An Unequal Caching Strategy for Shared-Memory Graph Analytics

YuAng Chen and Yeh-Ching Chung

Abstract—Recent advances in computer architecture significantly enhance the computational capacity of multicore systems. It allows large-scale graphs to be processed inside a single machine. Nevertheless, the irregular processing pattern of graph-structured data constrains the hardware resources from being productively utilized. In this paper, we investigate the constraints in two aspects: workload imbalance and parallel inefficiency. When a graph analytics algorithm is multithreaded, the thread time is highly diversified, indicating an uneven work distribution. Also, the intensive thread contention lowers the computing capacity of CPU cores, thereby hindering the effective utilization of CPU resources. To address these challenges, we present a proactive graph caching strategy that unequally segments graph components into cache-able subsets of varying sizes, namely Syze. First, the computational loads of cache-sized subgraphs are estimated. Then, the demanding subgraphs are further subdivided until certain threshold is met. Moreover, during the propagation of updates, a fraction of vertex ID (i.e., several bits) are encoded to facilitate the communication between subgraphs. As a result, Syze is able to balance the workloads amongst the logical cores by shortening the longest thread execution time. Meanwhile, it alleviates thread contention and thus elevates the parallel efficiency of multicores. Compared with well-optimized Ligra, Gemini and GPOP, Syze achieves accelerations by up to $17.76\times$, $11.67\times$ and $2.81\times$ respectively. Additionally, the side effects of Syze are evaluated, including raised cache misses and memory accesses. They play a trivial role in deciding the overall performance, as their costs are far outweighed by the gains from the even distribution of workloads and the improved utilization of multicores.

Index Terms—Graph analytics, multicore system, parallel computing

1 Introduction

The continuous evolution of CPU architecture and RAM technology has remarkably boosted the computing power of server-class multicore systems. Equipped with a huge volume of RAM, e.g., 1 Terabyte, a single multicore machine is able to process the largest publicly available graph within its own memory [1]. Without the necessity of offloading the workloads to external memory (i.e., disks) [2], [3] nor distributed memory (i.e., clusters) [4], [5], the in-memory graph processing avoids the overheads relating to disk I/O and network communication. Thus, it often delivers better performance compared with its external- and distributed-memory counterparts [5], [6].

However, the irregular data structure of graphs still imposes non-trivial challenges for the efficient utilization of multicore systems. The computation of graphs invokes a large volume of communication among the vertices along their edges. This leads to highly randomized memory access pattern [7]. Also, a race condition occurs when multiple vertices share a common source/destination vertex. Hence, many frameworks require heavy usage of atomics and mutexes for synchronization [8].

Moreover, many real-world graphs are featured by power-law degree distribution, as commonly observed in social, biological, citation and web networks [9], [10], [11], [12]. In a graph, a minority of vertices constitute a large portion of edges. By contrast, the majority of vertices are sparsely connected or even completely isolated with degrees equal to zero. The ”hot” vertices (i.e., densely connected ones) raise frequent message exchanges and create substantial workloads. During parallelized graph processing, the ”cold” vertices (i.e., sparsely connected ones) have to wait idly for the hot vertices to finish executions. The skewed distribution of vertex degrees not only causes imbalanced workloads for parallel graph processing, but also burdens the synchronization overhead.

Besides the graph data, the irregularity also exists in graph algorithms. When a graph is processed, in each iteration, updates are propagated along the active edges. The propagating path depends on the connectivity of root vertex (if needed), the structure of graph and the algorithm design. Hence, it is unpredictable. As a specialty, PageRank keeps all vertices active and traverses all edges, which controls itself as the most ”regular” graph algorithm.

To address aforementioned issues, a variety of optimizations are introduced by prior works. For example, Ligra maintains a frontier of active vertices during graph traversal to prevent unrelated vertices being cached [13]. Polymer leverages NUMA awareness in multi-CPU machines to reduce remote memory accesses [14]. GraphMat translates graph algorithms into generalized sparse matrix vector multiplication (SpMV) and then accelerates the computation from the angle of matrix [15]. Grazelle delicately refines the nested parallelization in pulling direction [16].

In addition to the above, a series of works based on *cache-able* subgraphs are proposed for aggressive graph caching [17], [18], [19]. They follow a core methodology: the vertex set of a graph,

- *The authors are with the Chinese University of Hong Kong, Shenzhen, Guangdong 518172, China. E-mail: yuangchen@link.cuhk.edu.cn, ychung@cuhk.edu.cn.*

Manuscript received 30 September 2021; revised 15 August 2022; accepted 27 October 2022. Date of publication 9 January 2023; date of current version 24 January 2023.
This work was supported in part by the National Key Research & Development Program of China under Grant 2018YFB1003505 and in part by the Large-scale Graph Pattern Query System and Its Optimization Strategy of Ali Damo Research Institute under Contract 2022E0018.
(Corresponding author: Yeh-Ching Chung.)
Recommended for acceptance by A. Randles.
Digital Object Identifier no. 10.1109/TPDS.2022.3218885

in the format of Compressed Sparse Row (CSR), is equally segmented by the size of caches. The disjoint vertex subsets are allocated to different threads for parallel execution. The access range per thread is confined within the cache-sized segment inside the cache. As a result, vertices are repeatedly accessed in nearby locations, thereby promoting high cache locality.

In general, prior shared-memory works focus on the optimization of memory-cache hierarchy in multicore systems. They aim to eliminate memory accesses and cache misses. In this paper, we investigate the optimization strategy from an often-ignored perspective: the CPU cores.

Due to the irregularity of graphs, the performance of multicores are limited by two issues: workload imbalance and parallel inefficiency. When a graph algorithm is parallelized, its overall processing time is bottlenecked by the longest thread execution time. Further, due to thread contention, multicores cannot be effectively utilized for parallelization. Full utilization of logical cores might even lead to a slowdown.

Motivated by these issues, we propose Syze, an unequal-sized caching strategy for shared-memory graph processing on multicore systems. It segments a graph into numerous subgraphs with the goals of alleviating uneven workloads and improving parallel efficiency (i.e., the effective utilization of available cores). In summary, our contributions can be generalized as follows:

- We showcase that, during the processing of graph, the completion time of threads is highly diversified, which reflects the imbalanced workloads. Moreover, if an excessive number of threads are employed, the thread pool would become overcrowded, thus hindering the effective parallelism of multicores.
- To tackle the above problems, we propose an unequal caching strategy, named as Syze. Syze splits the vertex set of a graph into cache-able subsets of varying sizes, so that the workload is fairly distributed and parallel efficiency is enhanced.
- Bitwise operations, namely Anchoring & Chaining, are designed to locate the subgraph of any given vertex according to its ID. Based on the manipulation of bits, a bit-aware propagating mechanism is built, which promotes efficient communication between subgraphs.
- Compared with the state of the art, Syze proceeds at a considerably faster speed. Comprehensive analyses are undertaken to explore Syze's effectiveness on diverse graphs as well as its effect on the multicore systems. Further, Syze is decomposed in order to find the root cause for its high performance.

After the introduction, Section 2 discusses background and motivation. Section 3 describes the design of Syze in details. The performance of Syze is evaluated in Section 4. Related research is provided in Section 5. Lastly, this paper is concluded in Section 6.

2 Background and Motivation

2.1 Preliminaries

This section introduces a fundamental graph format and an optimization methodology developed based on it.

2.1.1 Compressed Sparse Row

Compressed Sparse Row (CSR) represents a graph in a memory-efficient manner. CSR requires two arrays to encode a graph: Vertex and Edge. Fig. 1 depicts a directed graph and its CSR representation. For each vertex, the Vertex array registers the starting offset of its out-neighbors in the Edge array. The last element of the Vertex array stands for the number of edges. The Edge array stores all out-edges by consecutively recording the out-neighbor list of every vertex. Optionally, a Property array is attached. For example, in the context of PageRank, the Property array keeps the rank scores of vertices.

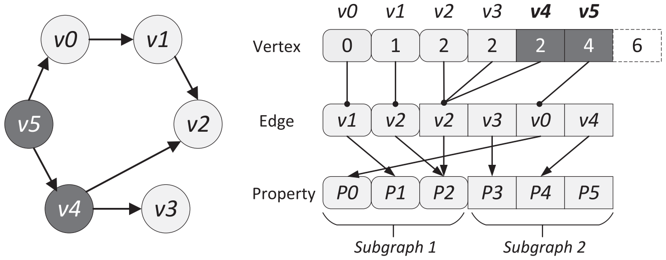


Fig. 1. CSR representation of a directed graph. Assuming a private cache can hold three vertices, the graph is divided into two subgraphs as the round and square boxes respectively. In particular, hot vertices $v4$ and $v5$ are highlighted because they are densely connected with two out-edges. They lead to workload imbalance between the two subgraphs.

Traversing the graph, elements in Vertex array and Edge vertex are sequentially accessed for only once. On the contrary, the access to PA exhibits irregularity. For instance, P2 is consecutively requested for twice, P5 is never reached, and P0 is visited at the end. The irregular access to the Property array is the root cause of random memory access on multicore systems.

2.1.2 Graph Analytics Based on CSR Segmentation

Based on the CSR format, a number of similar graph segmenting methods are proposed [17], [18], [19], [20]. In general, these methods split the vertex set of the graph (i.e., row array of CSR) into numerous subsets fitting in either Level 2 (L2) cache or Last Level Cache (LLC). The right part of Fig. 1 exemplifies the segmenting results with an assumption that a cache can accommodate three vertices.

After the segmentation of CSR, thread access to vertex is restricted within local caches, which significantly enhances the cache locality. Furthermore, a subgraph is exclusively processed by one thread. Hence, the race condition is diminished and atomic instructions are removed. As a result, the works (e.g., GPOP [7]) based on CSR segmentation significantly outperform conventional vertex-centric frameworks.

Integrated into the essential CSR segmenting methodology, various optimization techniques are designed. For example, the edges of subgraphs are sorted to ensure sequential memory access [18], [20]. Edges pointing out to multiple vertices inside the same destination subgraph are compressed into one inter-edge to reduce memory traffic [17]. Hierarchical frontier is implemented for both subgraphs and vertices to avoid caching unrelated graph components [7].

Additionally, some researchers refer to the graph processing model built on CSR segmentation as the *partition-centric* paradigm [7]. In following context, we use the terms ”partition-centric” and ”CSR-segmenting” exchangeably to describe the graph processing paradigm where CSR is segmented by caches.

2.2 Motivation

This section investigates the computing challenges in prior CSR-segmenting graph processing frameworks. We experimentally demonstrate that CPU cores severely suffer from

TABLE 1
Statistical Summary of GPOP-Based PageRank on Graph *Twitter*

Attribute	Max	Min	Aver
Thread Time (ms)	320.39	1.78	6.06
Ratio (Max/∗)	1.00×	180.16×	52.86×
Degree Sum (million)	212.52	0.63	2.31
Ratio (Max/∗)	1.00×	339.49×	92×

The execution time of threads (Thread Time) per iteration and the sum of vertex degrees (Degree Sum) per segment are presented in terms of the maximum (Max), minimum (Min) and average (Aver) results. The lower row (Ratio) within each attribute lists the ratios between the max result and respective results.

workload imbalance as well as parallel inefficiency. These issues motivate the proposal of our work.

2.2.1 Workload Imbalance

The contrast between hot vertices and cold ones causes workload imbalance when a graph algorithm is multithreaded. To tackle this issue, conventional shared-memory graph analytics frameworks dynamically parallelize graph algorithms using the work-stealing policy of Cilk [13], [21] or the dynamic scheduler of OpenMP [7], [18]. However, it is insufficient to solely rely on the programming tool to address the problem of imbalance.

In CSR-segmenting works, a subgraph is the basic unit for threads to execute. Since the number of vertices in all subgraphs are equated, the workload for each subgraph mainly depends on the edges. The local concentration of hot vertices leads to imbalance among subgraphs. In the example of Fig. 1, vertices $v4$ and $v5$ together contribute 4 out-edges in subgraph 2, while there are only 2 out-edges in subgraph 1. Thus, subgraph 2 tends to involve heavier communications as well as computations.

To validate our reasoning, we measure the execution time of parallel threads by running PageRank under a CSR-segmenting framework GPOP [7] on graph *twitter* [22]. PageRank is selected for its relatively regular pattern [23]. A Intel Xoen Silver machine is used, the configuration of which is detailed in Section 4.2. The application is parallelized by 20 threads, and the size of segments is 256 KB.

Table 1 lists the experimental results. It can be observed that the max thread costs considerably longer time than the average and min threads. Such difference results from the imbalanced distribution of hot vertices. Also, the vertex degrees of each segment are summed up. The densely connected vertices are highly concentrated in a few subgraphs, exhibiting the intrinsic irregularity of the graph.

For a parallel region, the overall processing time is decided by the longest thread time. The results in Table 1 suggests that the CSR-segmenting paradigm is hindered by the fluctuating thread execution time due to the concentration of hot vertices. This encourages us to develop a strategy that bridges the gaps between threads, i.e., shortening the longest thread time when multithreading.

2.2.2 Parallel Inefficiency

CSR segmentation leads to inefficient utilization of CPU cores. The efficiency of a program utilizing available cores is defined by *parallel efficiency (PE)*. This metric reports the percent of average utilization by all cores, which can be measured via Intel Vtune Profiler [24]. By multiplying PE with the number of available cores, the number of effective cores is acquired. For example, assuming a 20-core system, a parallel efficiency of 50% means that the system behaves as if $20 \times 50\% = 10$ cores are fully loaded by computations.

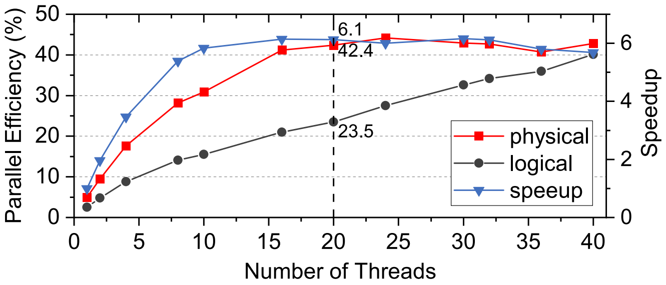


Fig. 2. Parallel efficiency of physical cores and logical cores and the speedup over the 1-threaded implementation, by performing GPOP-based PageRank on graph *twitter*.

Today's CPU is able to provide two logical cores (i.e., threads) per physical core by Hyper-Threading [25]. It is beneficial when two logical cores execute independently without resource contentions [26]. The parallel efficiencies of logic cores (PE_L) and physical cores (PE_P) can be measured simultaneously using Vtune Profiler. PageRank is tested for its dense computation [23], which heavily consumes resources. The same Intel Xoen Silver machine is used, offering 20 physical and 40 logical cores. Fig. 2 depicts the PE_L, PE_P and speedup with varied size of thread pool T.

PE_L scales almost linearly with increasing T, while the scaling behaviors of PE_P and speedup are both sub-linear. The later two achieve peak performance when $T = 20 =$ the number of physical cores. At this point, $PE_P = 42.4\%$ and $PE_L = 23.5\%$, and the system functions as only around 9 cores actively executing the application, though 20 threads are deployed. The low values of PE_P and PE_L signal that available CPU resources are underutilized. As T grows from 20, PE_P and speedup are saturated, and even deteriorated. Cache-friendly programs might suffer from Hyper-Threading due to thread contention [27], [28]. In GPOP, the cache-fitting subgraphs promote high cache efficiency. The logical cores sharing a physical core could compete for caches' accesses. Thereby, the computing capacity (e.g., effective L2 cache size and speed) of a logical core is halved. Compared with a physical core, two logical cores in parallel attain the same throughput, but double the PE as two units are counted. The contrast between PE_P and PE_L reflects the occurring of thread contention.

PE is the result of complex interactions among workload imbalance, thread contention and resource underutilization. It is rarely discussed in prior CSR-segmenting works [7], [17], [18], [20], which simply set the size of thread pool equals the number of physical cores. In this paper, in-depth investigations into PE are undertaken to maximize the performance.

3 Design of Syze

To ameliorate workload imbalance and enhance parallel efficiency, we propose Syze, an optimization strategy for graph processing based on CSR segmentation. The abstraction of Syze is visualized in Fig. 3 and the design goal of it is to dilute the hot subgraphs. In principle, the subgraph identified as "hot" are further subdivided into sub-units with smaller sizes to alleviate the concentration of hot vertices. The details of Syze are elaborated in following sections.

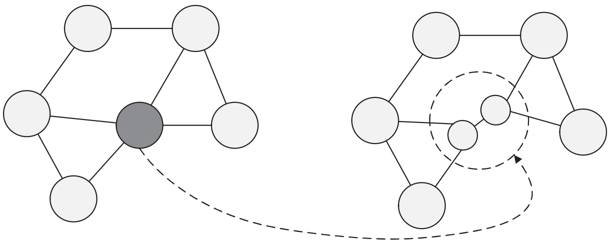


Fig. 3. Abstraction of unequal-sized caching. A graph is first segmented into cache-able subgraphs (grey cycles) with uniform vertex subset size. Then, the hot subgraph accommodating an excessive number of hot vertices (in dark grey) is further subdivided into smaller units.

3.1 Unequal Caching

Before describing the mechanism of unequal caching, we first formally define the terminologies with respect to the graph and subgraphs. A graph is represented by $G = (V, E)$, where V is the vertex set and E is the edge set. The capacity of cache is denoted as C = {size of cache}/{size of a vertex}. Further, *Subgraph degree* D^s is defined as the total sum of vertex degrees in a vertex subset. Based on the definitions, we unequally cache a graph in 3 steps:

Step 1: Initial Segmenting. First of all, the vertex set V is segmented into N cache-able subset V_i with identical sizes of C: $N = |V|/C$ and $|V_i| = C$, where $0 \leq i < N$ and $|\cdot|$ means the size of a set. A global operand $R = log_2\ C$ is used by any given vertex v to locate its belonging subgraph S_i, as formulated in Eq. (1)

$$i = v \gg R. \tag{1}$$

Here, $\gg$ is the right shift operator. The locating of a subgraph is a critical operation in partition-centric paradigm, because the destination subgraph (containing the destination vertex) is needed when an update is propagated across subgraphs. This bit-aware propagating mechanism is detailed in Section 3.3.

An example is offered as in Fig. 4. The *old* array lists the subgraphs constructed in the initial segmenting step. Assume 64 vertices are separated into 4 subsets in the order of vertex ID. Each subset contains 16 vertices, so the R equals 4. The vertex ID can be represented in 6-bit binary format.[1] To locate the belonging subgraph of a vertex, only the leftmost 2 bits are needed; the rightmost $R = 4$ bits are ignored due to right shift. In other words, the coverage of each subgraph is expressed by the leftmost 2 bits of vertex ID. This is exactly the same as prior CSR-segmenting graph processing, where the sizes of all vertex subsets are equally fixed [17].

Step 2: Identifying Hot Subgraphs. With all subgraphs segmented with a uniform size, we measure the hotness of subgraphs by the sum of vertex degrees per subgraph. The average subgraph degree $\overline{D^s}$ is obtained by multiplying the average vertex degree over the entire graph $\overline{D}$ with the size of vertex subset: $\overline{D^s} = \overline{D} \cdot C$. It serves as a threshold. The hotness of a subgraph is evaluated by simply comparing its D^s_i with $\overline{D^s}$ as in Eq. (2). When $hot_i \geq 2$, the corresponding subgraph is identified as hot

$$hot_i = D^s_i / \overline{D^s}. \tag{2}$$

Step 3: Subdividing Hot Subgraphs. According to the hot_i of hot subgraphs, further subdivision is conducted. The number of sub-units n_i split from each hot subgraph is calculated by performing binary exponentiation and binary logarithm together as in Eq. (3)

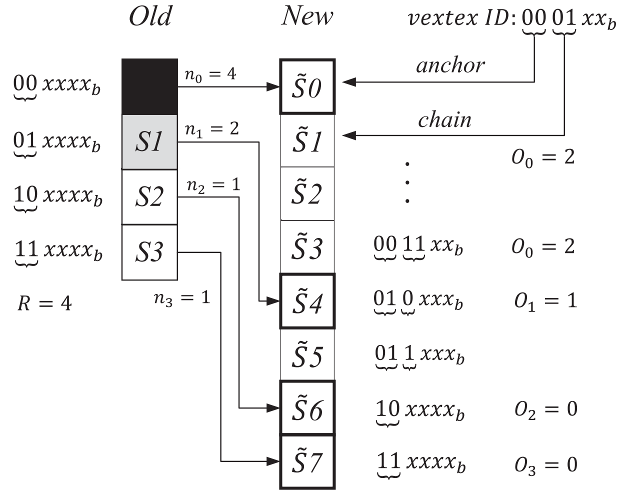

Old
New
$vextex\ ID: \underbrace{00}\,\underbrace{01}\,xx_b$
anchor
chain
$\underbrace{00}\,xxxx_b$
$\underbrace{01}\,xxxx_b$
$\underbrace{10}\,xxxx_b$
$\underbrace{11}\,xxxx_b$
$R = 4$
S1
S2
S3
$n_0 = 4$
$n_1 = 2$
$n_2 = 1$
$n_3 = 1$
$\tilde{S}0$
$\tilde{S}1$
$\tilde{S}2$
$\tilde{S}3$
$\tilde{S}4$
$\tilde{S}5$
$\tilde{S}6$
$\tilde{S}7$
$O_0 = 2$
$\underbrace{00}\,\underbrace{11}\,xx_b$
$O_0 = 2$
$\underbrace{01}\,\underbrace{0}\,xxx_b$
$O_1 = 1$
$\underbrace{01}\,\underbrace{1}\,xxx_b$
$\underbrace{10}\,xxxx_b$
$O_2 = 0$
$\underbrace{11}\,xxxx_b$
$O_3 = 0$

Fig. 4. An example of unequal-sized segmenting. Hot subgraphs in the old array are subdivided into smaller units in the new array. The anchored subgraphs are plotted with thick boxes and mapped to their original subgraphs. The precise location of a target subgraph in the new array depends on both *anchor* bits and *chain* bits. The coverage of each subgraph is annotated in binary format.

$$n_i = 2^{O_i} = 2^{\lfloor log_2(hot_i)\rceil}. \tag{3}$$

The $\lfloor\cdot\rceil$ stands for the rounding of a number, and the rounded value for each subgraph is recorded as local offset $O_i = \lfloor log_2(hot_i)\rceil$ later used in Section 3.2. Eq. (3) forces n_i to be an integer that is a power of 2, e.g., 1, 2, 4 and 8, where $n_i = 1$ means no subdivision. By contrast, hot_i can be any floating number. As a result, a hot subgraph is subdivided into n_i sub-units, the size of each which is C/n_i.

For the example of Fig. 4, we further assume that $S0$ and $S1$ are hot subgraphs and can be subdivided into 4 and 2 sub-units respectively. Hence, $S0$ is associated with $(n_0 = 4, O_0 = 2)$, $S1$ is with $(n_0 = 2, O_0 = 1)$, and both S_2 and S_3 are with $(n_0 = 1, O_0 = 0)$. After the subdivision, new subgraphs, including the sub-units, are constructed as in the *new* array of Fig. 4.

3.2 Locating a Subgraph

The subdivision of hot subgraphs leads to an increase in the number of subgraphs, containing the sub-units with sizes $<$ C. The varying size of subgraphs invalidates the bitwise operation for locating a subgraph by a given vertex ID, i.e., Eq. (1). One possible alternative is to use `if & else` or `switch & case` statements to exhaustively examine the coverage of each subgraphs until the correct one is found.

Nevertheless, those conditional statements are computationally inefficient. For example, they might cause a massive number of mispredicted branch instructions, which wastes the pipelines of CPU. Thereby, for computational efficiency, we advance the bitwise operation by *Anchoring* & *Chaining:*

Anchoring. A mapping table is maintained between the original subgraphs and the new subgraphs after subdivision. As in Fig. 4, every subgraph in the old array points to its first sub-units or itself (with new ID if not subdivided) in the new array. For the newly created subgraphs, we refer to the pointed elements as the *anchored* subgraphs and the

[1] The subscript b denotes a binary number

remaining ones as the *chained* subgraphs. Hence, we can still rely on the leftmost 2 bits, which we label as *anchor* bits, to locate an anchored subgraph. For instance, given vertex ID $11xxxx_b$, where $\forall x \in \{0,1\}$, shift it rightwards by $R = 4$ bits, and its anchor equals $11_b = 3$. Thus, it originally lies in $S3$, which is now pointing to $\tilde{S}7$, forming a mapping relationship: $map(S3, \tilde{S}7)$ and $map[S3] = \tilde{S}7$.

Nonetheless, the mapping table only allows us to recognize a fraction of subgraphs (i.e., anchored subgraphs) in the new array. The chained subgraphs stay out of reach. For a complete solution, one more step is planned.

Chaining. To find the chained subgraphs in the new array, extra bits in the vertex ID are utilized. Recall that, in Step 3, an offset O_i is affiliated to each subgraph in the old array. It indicates the number of extra bits, labeled as *chain*, needed to scan right after the anchor bits. The value of *chain* is the offset from current *anchored* subgraph to target subgraph.

For example, since $S0$ is attached with $O_0 = 2$, it requires 2-bit anchor and additional 2-bit chain, together the leftmost 4 bits, to locate its sub-units. Given vertex ID $\underline{00}\,\underline{01}xx_b$, by its anchor 00_b, we identify that it is initially contained in $S0$, which now links to the anchored subgraph $\tilde{S}0$. Then, according to the chain $01_b = 1$, we can locate the target subgraph by offsetting its anchored subgraph $\tilde{S}0$ by 1, which is $\tilde{S}1$.

In a compact form, Anchoring & Chaining can be mathematically described as following formulas

$$anchor = v \gg R \tag{4a}$$

$$anchored\_sub = map[anchor] \tag{4b}$$

$$chain = (1 \ll O_i - 1) \& (v \gg (R - O_i)) \tag{4c}$$

$$chained\_sub = anchored\_sub + chain \tag{4d}$$

Eq. (4a) first finds the old subgraph before the subdivision is performed. Eq. (4b) depicts the mapping procedure between the original subgraph and the anchored subgraph. Eq. (4c) shows the arithmetic operations of truncating the chain bits from a vertex ID. Finally, Eq. (4d) locates the chained subgraph after subdivision.

It is worthy to note that, in Step 1, the optimal initial segmenting size for CSR segmentation does not always equate to the size of caches, e.g., neither L2 nor LLC. The exploration regarding the proper initial segmenting size is discussed in Section 4.7.

3.3 Bit-Aware Propagation

The propagation of updates in Syze behaves as a variant of the classic Scatter-Gather model. A source (src) vertex first broadcasts its property data over its active neighboring subgraphs (Scatter). The data are stored in the buffers of these subgraphs. Then, the destination (dst) vertex collects all the received data from the local buffer of its belonging subgraph (Gather). The propagating mechanism of Syze exhibits the uniqueness in its bit-awareness of vertex ID. During the scattering phase, the leftmost bits of the dst vertex are extracted to locate the dst subgraph. Also, during the gathering phase, an extra bit is inserted in front of the dst vertex to reduce communication between subgraphs.

Fig. 5 explains the bit-aware propagation of Syze with a concrete example. The upper part of Fig. 5 presents a conventional message passing from one src vertex to two dst vertices. We assume the src vertex and dst vertices reside inside two different subgraphs (within two L2 caches) in the context of CSR-segmenting graph processing. Hence, the propagation of data along the edges would cross the private caches of cores, and traverse the LLC or even main memory, causing a high latency.

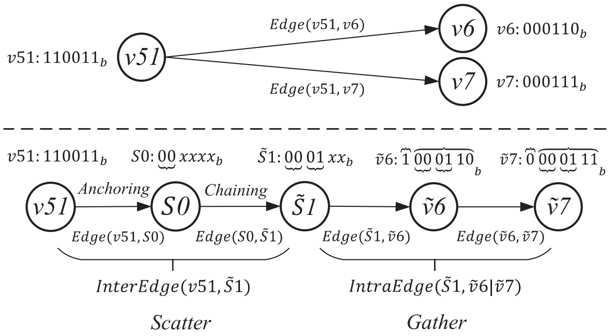

$v51: 110011_b$
$v51$
$Edge(v51, v6)$
$Edge(v51, v7)$
$v6$
$v7$
$v6: 000110_b$
$v7: 000111_b$
$v51: 110011_b$
$S0: \underbrace{00}xxxx_b$
$\tilde{S}1: \underbrace{00}\,\underbrace{01}\,xx_b$
$\tilde{v}6: \overbrace{1}\,\overbrace{\underbrace{00}\,\underbrace{01}\,10}_b$
$\tilde{v}7: \overbrace{0}\,\overbrace{\underbrace{00}\,\underbrace{01}\,11}_b$
Anchoring
Chaining
$v51$
$S0$
$\tilde{S}1$
$\tilde{v}6$
$\tilde{v}7$
$Edge(v51, S0)$
$Edge(S0, \tilde{S}1)$
$Edge(\tilde{S}1, \tilde{v}6)$
$Edge(\tilde{v}6, \tilde{v}7)$
$InterEdge(v51, \tilde{S}1)$
$IntraEdge(\tilde{S}1, \tilde{v}6|\tilde{v}7)$
Scatter
Gather

Fig. 5. Bit-aware propagation of Syze.

The lower part of Fig. 5 plots the data flow triggered by the same vertices in Syze. The Scatter and Gather phases are presented, which govern the transmission of updates. Notice that the dst vertices $v6$ and $v7$ only differ in one bit, so they are included in the same subgraph $\tilde{S}_1$.

In the Scatter phase, according to composition of $v6$ (i.e., anchor bits 00 and chain bits 01), the update reaches to the subgraph $\tilde{S}1$ via $S0$. The edges $(v51, S0)$ and $(S0, \tilde{S}1)$ are delineated conceptually. In actual implementation, they are enacted by Anchoring & Chaining, which form an inter-edge transmitting data between $v51$ and $\tilde{S}1$. The linking procedure is detailed in Section 3.2.

In the Gather phase, the data buffered in $\tilde{S}1$ is locally fetched by the dst vertices. The edge compression [17] is adopted for the reduction of memory traffic. Since $v6$ and $v7$ share the same src vertex $v51$, the data fetched by $v6$ is copied to $v7$, instead of transmitting again from $v51$ to $v7$. To achieve this, one compress bit is inserted in front of the dst vertices: $v6 \rightarrow \tilde{v}6$ and $v7 \rightarrow \tilde{v}7$. When compress equals one, the vertex (e.g., $\tilde{v}6$) is responsible for collecting the data from local buffer. If the compress of next vertex is zero, then the data assigned to the previous vertex (e.g., $\tilde{v}6$) is duplicated to the next vertex (e.g., $\tilde{v}7$).

Similarly, the edges $(\tilde{S}1, \tilde{v}6)$ and $(\tilde{v}6, \tilde{v}7)$ are conceptual designs. They are actually implemented by a simple bitwise operation: examining the compress bit of vertex ID. This bit compresses two edges into one intra-edge. Following prior works [17], we employ the most significant bit (MSB) of vertex ID (32-bit unsigned integer) as the compress bit. The encoding of compress bit is performed only once during the segmenting of graphs, hence leading to no overhead during the propagating of messages.

The cooperation between the inter-edge and the intra-edge effectively reduces memory traffic at the (negligible) cost of additional computation. In the original propagation process (upper part of Fig. 5), two cross-core messages are transmitted. In comparison, in the lower part, the propagation of Syze compresses two cross-core messages into one inter-edge, and then locally sends out the message via one intra-edge.

3.4 Program Execution

Like preceding CSR-segmenting works [17], [18], [29], the iterative execution of Syze are decomposed into two separate parallel regions as outlined in Algorithm 1. The first region (line 14-17) includes the Scatter phase; the second covers the Gather phase (line 18-21). Besides, inheriting from GPOP [7], the Reset phase and the Apply phase are provided at the end of each region. They convey similar semantics, both which can update vertex data and control vertex state in the frontier (i.e., active or inactive). The cooperation of Reset and Apply improves the flexibility of expressing graph algorithms and reduces programming efforts.

During the execution, each phase corresponds to an interface calling a specific user-defined function. The example of PageRank is illustrated in line 1-12. Function `PR.scatter` returns the scaled rank scores of vertices (line 2), which is passed to `Scatter` for broadcast from current subgraphs to neighboring subgraphs. Then, the rank scores of vertices are reset to zero and vertex states stay active with `PR.reset` (line 4) called by `Reset`. `PR.gather` (line 7) belonging to `Gather` accumulates the rank scores from the local buffers of active subgraphs. Finally, the rank scores are normalized by `PR.apply` (line 9) that is passed to `Apply`.

Apparently, the Scatter and Gather phases are the main source of workloads. They introduce sequential inner loops over all active subgraphs. Also, heavy memory traffic are invoked due to the data propagation across these subgraphs. On the contrary, the Reset and Apply phases merely perform on current subgraphs with basic arithmetic operations.

Meanwhile, Syze enjoys the hierarchic frontier of GPOP too, which maintains the activity states of graph components [7]. The frontier consists of two levels that record the states of subgraphs and vertices respectively. A subgraph turns active if it accommodates at least one active vertex. The active subgraphs and vertices are filtered out by the hierarchic frontier. Hence, for dynamic graph applications, such as BFS, the access to the inactive is avoided.

Algorithm 1. Iterative Execution of PageRank

```
 1: Structure PR {                                  //define a graph algorithm
 2: Function scatter ($v$)
 3:     return $rank[v]/deg[v]$
 4: Function reset ($v$)
 5:     $rank[v] = 0$
 6:     return $true$
 7: Function gather ($v, update$)
 8:     return $rank[v] += update$
 9: Function apply ($v$)
10:     $rank[v] = (1-d)/|V| + d \cdot rank[v]$
11:     return $true$
12: }
13: while $!converged \& iter < max\_iter$ do                  // run a task
14:   for all $curr \in active\_subs$ do in parallel           // region 1
15:     for $next \in active\_subs$ do
16:       Scatter(PR.scatter, $curr, next$)
17:     Reset(PR.reset, $curr$)
18:   for all $curr \in active\_subs$ do in parallel           // region 2
19:     for $next \in active\_subs$ do
20:       Gatter(PR.gather, $curr, next$)
21:     Apply(PR.apply, $curr$)
```

3.5 Benefits and Costs

The most prominent difference between Syze and prior CSR-segmenting works lies in the segmenting procedure of graphs. In Syze, a graph is unequally cached. Hot subgraphs are identified and further subdivided into smaller units. By contrast, previous methods only deploy a equal-sized segmenting strategy where the sizes of all segments are equivalent, which is just the first step of Syze's unequal caching strategy.

The benefits of unequal caching strategy involves two aspects. First, the workloads amongst threads are balanced, since hot subgraphs are split into smaller ones. This effectively shortens the longest threads execution time, and thus reduces the overall graph processing time (see Section 4.5.1). Second, the thread contention is alleviated. Thanks to the subdivision of hot subgraphs, threads (i.e., logical cores) are assigned with smaller data unit to process. As a result, the computing pressure of CPU cores are relieved, and therefore the exhaustion of resources are postponed (see Section 4.5.2).

Though Syze improves the performance of CPU cores and accelerates graph processing, it imposes side effect on the memory-cache hierarchy in multicore systems. The subdivision creates extra subgraphs, which leads to more inter-edges. Moreover, the Anchoring & Chaining procedure requires supplementary memory space for the mapping table and additional arithmetic operations. Consequently, memory traffic is raised and cache misses are increased (see Section 4.5.3).

Another distinct feature of Syze is that it employs more threads for multithreading than prior CSR-segmenting methods, e.g., 36 versus 20. Syze takes the advantage of relieved thread contention by increasing the size of thread pool. Therefore, the available CPU resources are better exploited. The benefit of adding more threads relies on the unequal caching strategy, and is relatively minor (see Section 4.6).

4 Evaluation

In this section, dual evaluations combining theoretical calculations and experimental observations are undertaken. First, the intrinsic characteristics of graphs are profiled with respect to skewness and imbalance, which theoretically affect the effectiveness of Syze.

Then, the performance of Syze is evaluated by comparing with state-of-the-art graph processing frameworks. We also investigate the impact of Syze on CPU cores as well as cache and memory. Furthermore, Syze is decomposed for a better understanding of its contribution.

4.1 Estimating Imbalance

The power-law degree distribution (i.e., skewness) of graphs does not necessarily guarantee the graph are imbalanced as well. It is possible that the hot vertices, though being a minority, are evenly distributed throughout the graph; that is, in the context of equal-sized CSR segmentation, subgraphs contain comparable hot vertices.

To estimate the imbalance of a graph, the subdivision factor n_i from Eq. (3) is utilized, which is the number of sub-units to be subdivided from a hot subgraph. It indicates the density of subgraphs before further subdivision. The larger the value of n_i is, the more hot vertices are concentrated in the corresponding subgraph, and hence the more imbalanced the graph is. For an imbalanced graph, it is expected to consist of a tiny portion of hot subgraphs with $n_i \gg 1$ and a large portion of cold subgraphs with $n_i = 1$. In opposition, when $n_i = 1$, a subgraph needs no subdivision. If all subgraphs of a subgraph are associated with $n_i = 1$, the graph is perfectly balanced.

Table 2 lists a diversity of graphs collected from the real world or generated by software tools. Graphs are equally segmented into subgraphs in which the vertex subsets are of size 256 KB. Their skewness is indicated by (1) the ratio of hot vertices over all vertices and (2) the percentage of hot vertices' edges over all edges. Their imbalance is portrayed by (1) the ratio of hot subgraphs over all subgraphs and (2) the maximum n_i of all subgraphs.

The first five graphs exhibit high skewness as well as high imbalance. A minority $(9 - 22\%)$ of vertices are responsible for constituting a majority $(63 - 94\%)$ of edges. Meanwhile, their hot subgraphs, in spite of being rare $(2 - 4\%)$, are crowded with hot vertices, delivering n_i up to 128.

The last three graphs *kron* and *urand* are well balanced (i.e., $n_i = 1, \forall i \in [0, N)$,). Though *kron* follows the power-law degree distribution (i.e., 9% vertices for 93% edges), its hot vertices are evenly distributed as none of its subgraphs requires subdivision. In *urand*, a connection is established by any two vertices with equal probability. Thereby, *urand* is absolutely unskewed and balanced. Graph *road* represents an important category of mesh-structured graphs that are characterized by low degrees and balanced distributions.

The statistical summary of graphs in Table 2 demonstrates that the skewness of a graph does not promise its imbalance. It is possible that a highly skewed graph has a uniform distribution of hot vertices across its subgraphs, such as *kron*. In such case, subgraphs will not be further subdivided, which cancels out the optimization of Syze. In theory, Syze cannot boost the performance of balanced graphs. Hence, in following sections, we only conduct experiments on the imbalanced graphs.

TABLE 2
Graph Features in Terms of the Hot Vertices Percentages (*V*), the Edge Coverage of Hot Vertices (*E*), the Hot Subgraphs Percentage (*H*), and the Maximum n_i

Graphs	Description	Skewness		Imbalance	
		V (%)	E (%)	H (%)	$\max(n_i)$
track	Web Tracker [30]	18	63	2	128
live	Live Journal [31]	22	91	2	32
wiki	Wiki Link [32]	19	94	4	32
twitter	Twitter Follower [22]	9	79	2	64
mpi	Twitter Influence [33]	11	81	3	128
kron	Synthetic Graph [34]	8	93	0	1
urand	Synthetic Graph [34]	51	59	0	1
road	USA Road [35]	50	66	0	1

4.2 Experiment Setup

The experiments are performed on a dual CPU machine. The model of CPU is Intel Xeon Silver 4210 with Skylake microarchitecture. Each CPU is composed by 10 physical cores and 20 threads with Hyper-Threading enabled. The volume of main memory, shared LLC, private L2, and private L1 are 256 GB, 13.75 MB, 1 MB and 64 KB respectively. The operating system is Ubuntu 20.04.

Syze[2] is implemented by modifying the source code of GPOP [7]. The code is written in C++ with g++ 9.3.0 and compiled with optimization level O3. The multithreading is enabled by the use of OpenMP [36]. The cache utility and the memory dynamics are monitored by `Perf` [37] and `Likwid` [38] respectively.

The initial segmenting size is tuned to 256 KB, a quarter of L2 cache (explained in Section 4.7). 36 threads are used for the parallelization of graph algorithms (explained in Section 4.5.2). Each experiment is repeated for 10 rounds, and the average result is reported. For each round of PageRank, program runs for 20 iterations.

Graph Frameworks. The performance of Syze is compared with four cutting-edge frameworks: Ligra [13], Polymer [14], GPOP [7] and Gemini [5]. To obtain the best results, Ligra is multithreaded by OpenCilk [39] with the optimization of `numactl`. The native Ligra parallelized via OpenMP yields substantially poorer performance and therefore is not demonstrated. Polymer enhances Ligra by exploiting the NUMA feature of modern multicore systems. GPOP is set with 20 threads and 256 KB partition size to attain optimal performance. Although developed for distributed systems, Gemini draws considerable inspiration from shared-memory graph processing, such as NUMA awareness and hybrid push-pull engine. It exhibits superior performance than many shared-memory frameworks on single machine [5].

Graph Algorithms. Under these graph frameworks, four graph algorithms are tested. PageRank (PR) ranks the importance of vertex based on their connectivity [40]. Breadth-first search (BFS) traverses a graph from a given root vertex in breadth-first order. Single source shortest path (SSSP) finds the shortest path between a given root vertex and all other reachable vertices. Connected component (CC) classifies connected vertices into components using label propagation.

It is reported that [7], GPOP exhibits dominant advantage in PR, CC, SSSP thanks to its aggressive caching technique, compared to state-of-the-art works (e.g, GraphMat [41] and Galios [15], Ligra [13]). Meanwhile, Ligra offers the fastest implementation for BFS. Also, Gemini oftentimes delivers better performance than Ligra and Galios [5] on a single machine.

Due to limited space, our discussions focus on PageRank and BFS. They behave oppositely as the two ends of a spectrum. In our implementation, PageRank is exempted from the deployment of frontier because every vertex is updated in every iteration. It enables a system to run at the peak performance. Also, the examination of convergence is removed in all frameworks, so the accurate performance per iteration is obtained.

In contrast to PR, BFS is completely driven by frontier. During the whole processing, a vertex is accessed for only once. Therefore, BFS introduces lightweight communication and computation. Additionally, for root-dependent algorithms, the vertex with the largest degree is selected as the input to start graph traversal.

Graph Datasets. Five power-law graph datasets are selected for performance evaluation. Graphs *wiki* is extracted from the hyperlinks of wikipedia [32]. *track* collects web domains and their embedded tracker [30]. *live*, *twitter* and *mpi* are social graphs modeling social networks [22], [31], [33]. Table 4 lists the numerical information of the graphs.

4.3 Execution Time

Table 3 lists the execution time for the four graph algorithms implemented under Syze and cutting-edge frameworks. Syze constantly outperforms other competitors in all analytics algorithms except for BFS. Syze achieves speedups over Ligra, Polymer, Gemini and GPOP by up to $17.76\times$, $26.45\times 11.67\times$ and $2.81\times$, and on average $3.34\times$, $7.35\times$ $2.55\times$ and $1.80\times$ respectively. Nevertheless, it worth mentioning that Syze is theoretically ineffective at the intrinsically balanced graphs as previously analyzed in Section 4.1.

For PR, CC and SSSP, Syze provides the fastest results on most of datasets, except SSSP on *track*. GPOP, which is also

[2] https://github.com/yuang-chen/Syze-TPDS-22

TABLE 3
Graph Processing Time (In Seconds) Excluding the Overhead for Graph Algorithms Under Different Graph Frameworks

	PageRank						Connected Component				
Frameworks	*live*	*mpi*	*track*	*twitter*	*wiki*	Frameworks	*live*	*mpi*	*track*	*twitter*	*wiki*
Syze	**0.866**	**7.990**	**2.105**	**5.562**	**0.888**	Syze	**0.231**	**0.679**	**0.466**	**1.153**	**0.174**
GPOP	2.193	17.915	2.753	9.897	1.542	GPOP	0.650	1.587	0.521	1.484	0.409
Ligra	15.369	92.885	10.949	40.895	7.240	Ligra	1.122	5.660	2.096	2.814	0.790
Polymer	7.540	63.000	7.650	40.200	4.520	Polymer	1.020	7.490	1.300	3.940	0.788
Gemini	1.602	22.573	2.826	10.767	2.677	Gemini	0.372	7.916	0.915	5.326	1.009
	Single Source Shortest Path						Breadth-First Search				
Frameworks	*live*	*mpi*	*track*	*twitter*	*wiki*	Frameworks	*live*	*mpi*	*track*	*twitter*	*wiki*
Syze	**0.249**	**1.688**	0.094	**1.410**	**0.266**	Syze	0.090	0.827	0.065	0.829	**0.143**
GPOP	0.652	2.906	**0.093**	1.661	0.497	GPOP	0.228	1.432	0.070	1.079	0.206
Ligra	0.269	3.037	0.123	3.002	0.501	Ligra	**0.054**	**0.526**	**0.050**	**0.408**	0.152
Polymer	0.252	2.140	0.208	1.730	0.340	Polymer	1.140	20.400	1.710	3.660	1.870
Gemini	0.305	3.259	0.461	1.813	1.149	Gemini	0.339	1.063	0.649	0.805	1.612

partition-centric but with equal segmenting size, performs as the second fastest framework. The partition-centric paradigm (e.g., Syze and GPOP) shows considerable advantages over the vertex-centric paradigm (e.g., Ligra, Polymer and Gemini) on shared-memory multicore systems. This attributes to the aggressive cache and memory optimizations in the former, including cache-aware graph partitioning [19] and edge compression [17].

For BFS, Ligra performs as the best framework. We deliberately fine-tune Ligra to maximize its performance. Cilk is employed by Ligra for multithreading, which is substantially faster than the OpenMP-based Ligra. The hybrid push-pull processing engine and the lightweight vertex frontier plus the NUMA-awareness significantly shortens the execution time for BFS. On the contrary, Syze is parallelized by OpenMP. It is neither optimized in the propagation direction nor aware of NUMA. Also, the hierarchic frontier adopted by Syze imposes an extra burden of filtering subgraphs. Nevertheless, Syze still acts as the second fastest frameworks for BFS, offering performances comparable to Ligra.

Syze far outperforms GPOP due to the unequal caching strategy deployed on the imbalanced graphs. The hot subgraphs are identified and subdivided into smaller units for workload balance, which imposes high overheads. Moreover, the subdivision inevitably induces more subgraphs and complicates the connectivity amongst them. Also, the bit-wise operations Anchoring & Chaining lead to additional computations that may affect the overall performance. In the next sections, the benefits and costs of Syze are elaborated.

4.4 Preprocessing Overhead

The preprocessing overhead is an important metric to evaluate the practicality of a given framework. It is not counted in the processing time in other sections. Table 5 presents the overheads of different graph frameworks, which excludes the loading of data from disks.

TABLE 4
Graph Information (K: Thousand, M: Million, B: Billion)

Graphs	Vertices	Edges	Max Deg.	Refer.
track	27.7 M	140.6 M	1.1 M	[30]
live	7.5 M	112.3 M	300	[31]
wiki	18.3 M	172.1 M	9300	[32]
twitter	41.7 M	1.5B	3.0 M	[22]
mpi	52.6 M	2.0B	779 K	[33]

The NUMA-aware frameworks Polymer and Gemini generate staggeringly high overheads, thus being impracticable for real-world deployment. After the graph data is loaded, these frameworks have to partition and duplicate the graph data in different NUMA nodes again. The partitioning strategy for load balance and re-allocation of data in memory inevitably causes high computational and memory cost.

GPOP incurs the lowest overhead. Syze is built on GPOP by enabling unequal-sized CSR segmentation. For imbalanced graphs, Syze creates a larger number of inter-edges to encode (see later Section 4.5.3), thus leading to higher overheads. The overhead of Syze is roughly the same as that of Ligra, but Syze significantly outperforms Ligra for most of graph applications, including PR, CC and SSSP.

Furthermore, in Syze, the results of the preprocessing step (i.e., inter-edges, intra-edges, and a mapping table) are used by all downstream graph applications. Therefore, it is possible to store them as offline data, and reload them whenever a graph application needs to be executed. The preprocessing of a graph is performed only once for multiple applications. In such way, the overheads of Syze can be further reduced. We consider this feature as future work.

4.5 Impact on Multicores

Following our motivation in Section 2, we investigate the impacts of Syze on the multicores in two aspects: (1) workloads balance and (2) parallel efficiency. We also analyze the side effect of Syze in terms of cache and memory utilities. Moreover, the cost of Anchoring & Chaining is measured. Syze is benchmarked against a degraded version of itself that is deprived of any optimizations. In other words, the comparing baseline does not allow subdivision of hot subgraphs and only utilizes 20 threads, which is the same as prior partition-centric works (e.g., GPOP).

4.5.1 Workload Balance

The workload distribution amongst multicores is profiled by the execution time of threads. The thread with the *longest* completion time is evaluated, since it is decisive in a parallel region. Also, the majority of threads is represented by the *average* execution time of all threads.

TABLE 5
Preprocessing Overheads of Different Graph Frameworks in Seconds

	Syze	GPOP	Ligra	Polymer	Gemini
live	0.194	0.140	0.503	8.450	14.700
mpi	12.501	7.443	9.635	242.000	350.850
track	0.608	0.455	0.772	10.600	20.910
twitter	10.758	7.582	8.920	148.000	268.000
wiki	0.978	0.658	1.098	15.900	21.880
PR Iter.	20.453	13.651	19.971	358.382	574.572

Bottom row presents the average number of PageRank iterations by Syze to amortize the overheads.

Fig. 6 illustrates the speedup of Syze over the baseline in terms of graph processing, corresponding longest thread and average thread. Both the overall process and the longest thread (i.e., the most imbalanced workload) achieve remarkable accelerations. The longest thread time is reduced by $1.02 - 4.07\times$ (on average $2.27\times$) over all trials. Meanwhile, the speedup of graph processing is $1.20 - 4.20\times$ (on average $2.70\times$). The reduction of the longest thread time demonstrates the effectiveness of Syze in balancing workloads.

By contrast, the majority of threads are decelerated since the average speedup is $0.51 - 1\times$ (below 1 indicting a slowdown). This validates that it is the longest-running executor determining the overall processing time of a parallel region, instead of the majority. Moreover, the slowdown of the average thread encapsulates the cost of Syze, including the compromised efficiencies of memory and cache.

4.5.2 Parallel Efficiency

Fig. 7 shows the parallel efficiencies of Syze and the baseline by running PageRank on graph *twitter*. The parallel efficiencies of physical cores PE_P and logical cores PE_L are presented with varied number of working threads. Also, the execution time for PageRank is normalized by the single-threaded PageRank. The normalized results demonstrate the scalability of Syze. Compared with the baseline, Syze improves the utilization of CPU resources in two aspects: higher parallel efficiency and better scalability.

Syze raises PE_P and PE_L by up to 20 percent in comparison with the baseline. The enhanced parallel efficiency attains a significant acceleration for Syze. Against the single-threaded implementation, Syze with 20 threads delivers speedups of $9.85\times$. Meanwhile, the baseline achieves the highest speedup by only $5.99\times$ with 20 threads.

Syze exhibits better scalability than the baseline. As in Fig. 7b, when thread number > 20, the baseline receives no improvement, which behaves the same as GPOP in Fig. 2. This is because excellent cache affinity invokes thread contention on cache resources and counteracts the benefits of Hyper-Threading [26], [27], [28].

With underutilized caches (see later Section 4.5.3), Syze is enable to reduce thread contention and hence to leverage Hyper-Threading. The saturation of performance is postponed for Syze: it achieves the highest PE_P at 40 threads and the highest speedup by $10.70\times$ at around 36 threads. Multithreading with 36 out 40 threads serves as a good heuristic for all tested graph algorithms. The leftover capacity (e.g., 4 threads) is considered to be occupied by the background processes initiated by OS.

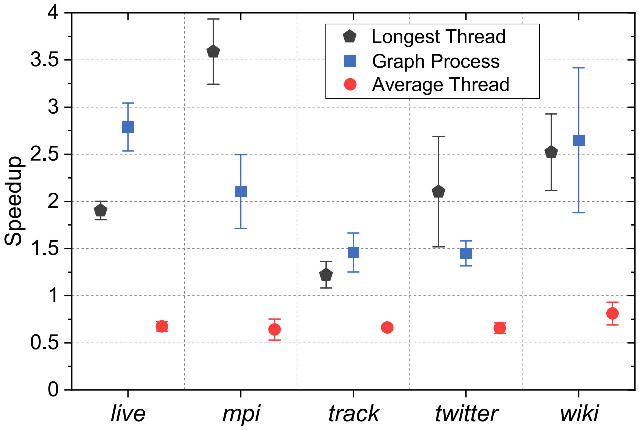


Fig. 6. The speedup of the graph processing, the longest thread and the average thread over the baseline. The results stand for the mean of different graph algorithms with 80 percent confidence interval.

Syze's high utilization of CPU cores essentially results from the subgraphs of varying sizes. The balanced workloads of subgraphs reduce the waiting time of parallel threads. This enables more active engagement of threads and thus facilitates higher parallel efficiency. Furthermore, since the caches can more easily accommodate small-sized subgraphs, the logical cores are less likely to compete for the cache resources, thereby leading to higher utilization of available cores.

4.5.3 Cost of Syze

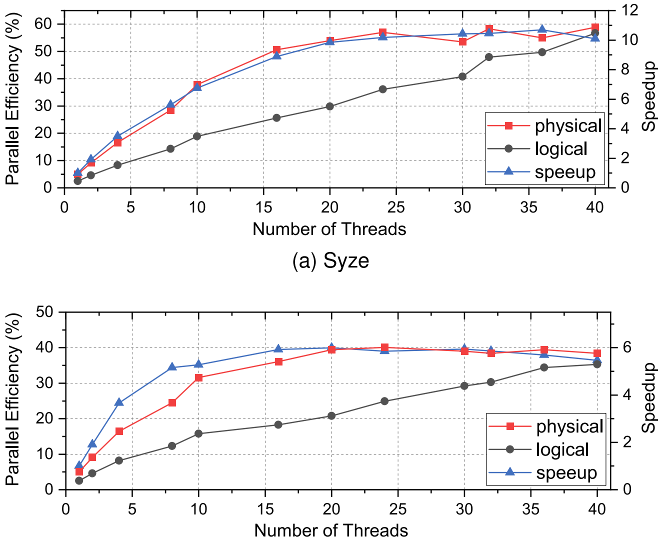
Though Syze brings substantial accelerations to graph applications, it also causes various side effects on the multicore systems. Hence, additional multicore activities are monitored, such as memory accesses and L2 cache misses. To better illustrate the cost of Syze, we report these metrics using *performance decline* = {Result of Syze}/{Result of Baseline}. The larger the value, the worse performance Syze incurs.

The performance deterioration related to memory and cache are plotted in Fig. 8. The average thread (defined in Section 4.5.1) is presented as well. Its performance decline equals the inverse of its speedup depicted in Fig. 6. The deployment of Syze lifts memory accesses as well cache misses, which results in the slowdown of the average thread.

The increased memory accesses and cache misses origin from the newly created subgraphs. As listed in Table 6, for Syze, the subdivision of hot subgraphs causes an upsurge of sub-units (i.e., smaller subgraphs) as well as inter-edges connecting these subgraphs. Hence, cross-subgraph communications are raised, which consequently leads to higher memory traffic. Also, although the small-sized subgraphs deteriorate the cache efficiency, they relieve thread contention on caches.

Also, Syze incurs additional cost in locating the subgraphs of vertices. For the baseline, the Chaining procedure is no more needed for locating the subgraphs due to its equal-sized segmenting; that is, Eqs. (4b), (4c) and (4d) are not performed by the baseline. In comparison, Syze requires complete Anchoring & Chaining for every vertex to locate its belonging subgraphs. Thus, Syze involves heavier computations and memory usage. Nonetheless, the overheads of locating the subgraphs are minor. As listed in Table 6, it costs less than 0.3 ms for any graphs to locate the subgraphs of all vertices.

Given the costs in the memory-cache hierarchy, Syze still delivers remarkable acceleration to the overall graph



(a) Syze

(b) Baseline

Fig. 7. The speedups of Syze and baseline and the parallel efficiencies of physical cores and logical cores, by performing PageRank on *twitter*.

processing, which is on average $2.70\times$. This is because the gain from workload balancing and effective parallelism far outweigh the loss from cache and memory inefficiencies. For instance, comparing Fig. 6 with Fig. 8, it can be observed that the speedup of the longest thread (on average $2.27\times$) is much higher than the performance decline of memory (on average $1.14\times$) and cache (on average $1.17\times$). In the next section, we explicitly quantify the respective contributions of balanced workload and improved parallelism.

4.6 Decomposition of Syze

The design of Syze differs from existing partition-centric paradigm in two distinct features. (1) Unequal-sized graph caching: the segment sizes are diversified due to the subdivision of hot subgraphs. Contrarily, prior works equally segment a graph into vertex subsets of the same size. (2) Optimized thread utilization: we exploit higher utilization of threads by deploying 36 out of 40 available threads. Other partition-centric methods, however, only employs half of available threads (e.g., 20) for task parallelization to prevent saturation.

To understand the contributions of the two features, the design of Syze is disintegrated and the components are evaluated separately. We employ Syze without unequal caching and with 20 threads as the plain baseline. Then, the baseline is compared with three variants of Syze. (1) Syze without unequal caching but with 36 threads, denoted as Syze-equal. (2) Syze with unequal caching and half-size thread pool, e.g, 20 out of 40 threads, denoted as Syze-half. (3) The fully functional Syze with unequal caching and 36 threads. The comparison results are presented in the form of speedup over the baseline as in Fig. 9.

Fig. 9 demonstrates that the unequal caching strategy acts as the main contributor to the effectiveness of Syze. Without the unequal solution, Syze-equal oftentimes causes a slowdown with speedup $< 1\times$. Simply adding more threads into the parallelization of a graph application is prone to deteriorate the processing speed. When the unequal caching is adopted by Syze-half, the performance immediately receives a substantial boost. Based on Syze-half, the addition of threads further accelerates graph processing, though the improvement from Syze-half to Syze is relatively minor.

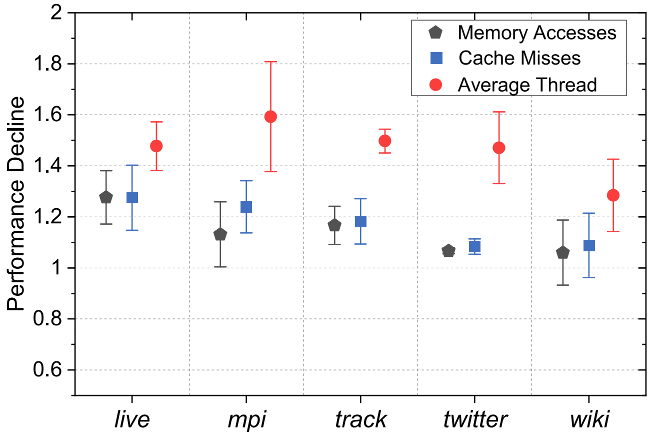


Fig. 8. The performance decline of the memory, the cache and the average thread compared with the baseline. The results stand for the mean of different graph algorithms with 80 percent confidence interval.

On the average of all graph algorithms and datasets, Syze-equal, Syze-half and Syze achieves speedups over the baseline by $1.06\times$, $1.76\times$ and $2.09\times$ respectively. The whole is greater than the sum of the parts. Also, the unequal caching strategy is ranked with higher importance than the large size of thread pool.

4.7 Initial Segmenting Size

To acquire the optimal size for the initial segmentation of graphs, a sensitivity analysis is conducted. The performance of Syze is examined by setting different initial segmenting size. As shown in Fig. 10, the results are normalized by the execution time of 16 KB.

The majority of graphs achieve the best performance when their initial segmenting sizes range *within* L2 cache: L1 cache size $<$ segmenting size $<$ L2 cache size. If the vertex subset is too small (i.e., fitting into L1 cache), there will be an upsurge in inter-edges and computational loads. Oppositely, if the size is too large, subgraph will spill over into LLC or even main memory, resulting in low cache locality and high memory traffic. In such case, the performance of Syze is drastically deteriorated.

Graph *live* behaves abnormally because its small size of vertex set. If segmented by 256 KB, *live* is subdivided into 149 subgraphs. On average, each thread is allocated with approximately 4 subgraphs. It is insufficient to fully exploit OpenMP's dynamic scheduler as threads might have to wait idly without new tasks.

Consequently, we choose 256 KB as the basic segmenting parameter at the stage of initial segmentation. Furthermore, a sufficient number of subgraphs N is required to facilitate dynamic scheduling policy. Thereby, we set $N \geq 8 \cdot T$, where T stands for the size of thread pool. For small-scale graphs, the initial segmenting size is halved consecutively until this condition is fulfilled.

5 Related Work

Graph Processing. A variety of high-performance graph analytics frameworks have been developed in recent decades. The vertex-centric programming model is introduced by the pioneering Google's Pregel [42] and then employed by Apache's Giraph [43] and Facebook for analyzing social networks [44]. However, these frameworks are operated on distributed systems and usually suffer from expensive network traffic.

Exempted from the network overhead, shared-memory graph frameworks are created for single-machine multicore

TABLE 6
The Numbers of Subgraphs and Inter-Edges (In Million), and the Time for Locating the Subgraphs of All Vertices (In Milliseconds) by Syze and the Baseline

		live	*mpi*	*track*	*twitter*	*wiki*
Subgraphs	Syze	140	1055	566	747	350
	Baseline	115	803	423	636	279
Inter-Edges	Syze	41	888	69	499	83
	Baseline	18	549	45	437	56
Time Cost	Syze	0.042	0.209	0.109	0.146	0.073
	Baseline	0.039	0.160	0.094	0.120	0.063

systems [1]. The vertex-centric paradigm is customized with diverse features, such as hybrid push-pull engine [13], [16] and NUMA-awareness [14], [45]. These techniques, in return, inspire the advancement of the distributed counterparts [5].

In the situation where the memory volume is limited, out-of-core frameworks are devised to process graphs by the usage of disks [3], [6]. X-Stream invents the edge-centric programming model to reduce random disk accesses [46]. GridGraph further promotes efficient streaming of edges on the disk-based systems by deploying graph 2-D partitioning [2]. Besides, there are a genre of subgraph-centric frameworks that focus on local processing of graph subsets for distributed systems [47], [48], [49]. A number of shared-memory graph works adopt the concept of subgraph by segmenting the vertex set of CSR [7], [17], [18], [19], which can be referred to as the partition-centric paradigm. Following above definitions, Syze can be categorized as a shared-memory partition-centric framework designed for multicore systems. Domain Specific Languages (DSLs) are another research field of high-performance graph analytics. The DSLs aim to provide simplicity and efficiency for graph analytics. Green-Marl compiler allows intuitive expression of graph algorithms and produces an optimized parallel C++ implementation [50]. GraphIt separates graph description and optimization with an algorithm language and a scheduling language respectively [51]. Users are able to explore well-known optimizations with minimum implementation effort. In the future, it is possible to embed the essence of Syze into the graph DSLs.

Graph Partitioning. The partitioning of graphs functions as a prerequisite for distributed graph frameworks. Improper partitioning results will lead to imbalanced computation and communication among the machines, which bottlenecks the parallel graph processing. The issue of workload imbalance is an important concern for graph partitioning, especially for skewed graphs [52], [53]. Hence, most of distributed graph frameworks have already taken the this issue into consideration when being designed.

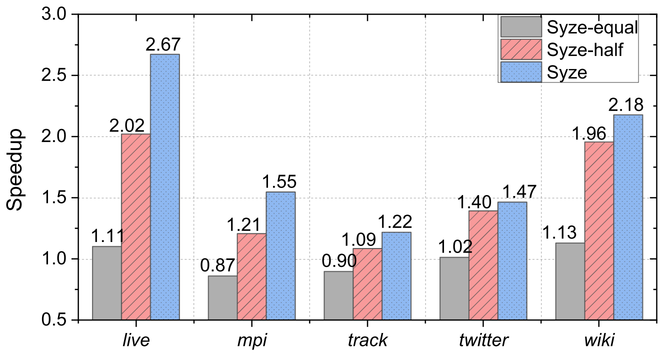


Fig. 9. The speedups delivered by Syze and its variants in comparison with the baseline, by performing BFS on *twitter*.

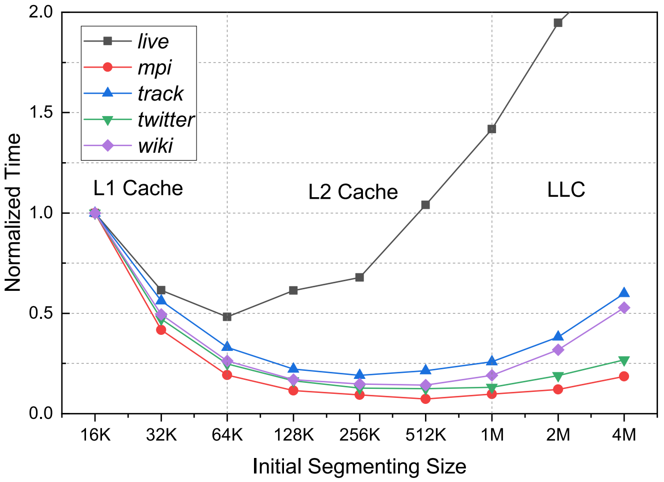


Fig. 10. The normalized execution time of Syze with different initial segmenting size, by performing BFS.

For instance, PowerGraph deploys vertex cut [54] that assigns machines with even number of edges and replicated vertices. PowerLyra applies hybrid cut [52], where low-degree and high-degree vertices are differentially partitioned and replicated to minimize replication overhead. Gemini ensures inter-machine balance by assigning each machine with balanced vertices and out-edges (similar to our method) [5]. Also, it employs work-stealing to balance the inter-core loads. Graphite proposes a 2-D partitioning and placement scheme is proposed to balance the computation/communication and to eliminate synchronization overheads [55].

Finding the optimal result of graph partitioning (i.e., min-cut) is proved to be NP-hard [56]. There have been growing attempts to reach an approximate solution to it, such as METIS [57], KaHIP [58] and PULP [59]. In order to obtain high-quality results, these partitioners undertake complicated calculations to analyze the internal structure of graphs (e.g., tree and community). However, despite the effectiveness in boosting the downstream graph applications, they tend to incur excessively high overheads. For high performance, their parallelized versions are proposed, including ParMETIS [60], ParHip [61], XtraPULP [62].

In most of shared-memory frameworks, a graph is globally accessible to all cores. Graph partitioning is unnecessary, unless a specific optimization is designed. Also, the issue of workload imbalance is poorly considered. Many frameworks simply rely on existing thread management policies, such work stealing or dynamic scheduling to handle this issue [7], [13], [18], [21]. As an exception, the shared-memory Polymer [14] adopts graph partitioning for NUMA machines and optimizes the workload distribution across NUMA nodes.

Syze splits a graph into subgraphs for cache locality. It takes proactive action to tackle the issue of imbalanced workloads by varying the sizes of subgraphs. Syze segments vertices according to edge information (i.e., vertex degrees). It leverages the very basic structural property of graphs (i.e., degrees), and thus causes a lightweight computational load. Moreover, the bit-aware propagation is customized for the unequal segmentation, together coalescing into the framework to facilitate the parallel execution of subgraphs.

6 Conclusion

In this paper, we propose an unequal caching strategy for high-performance shared-memory graph analytics, namely Syze. The proposal of Syze is motivated by the in-depth analyses of existing CSR-segmenting graph processing works. We experimentally demonstrate that, when the equal-sized CSR segmenting optimization is deployed on power-law graphs, CPU cores suffer from not only workload imbalance but also parallel inefficiency.

In order to address these issues, Syze identifies the hot subgraphs, and then subdivides them into smaller units. To locate the belonging subgraph of a vertex, the bitwise operations Anchoring & Chaining are designed, which efficiently encode the individual bits of vertex IDs. In this sense, updates of data are propagated with the awareness of vertex bits.

Syze is effective at graphs characterized by inherent imbalance, the feature of which can be theoretically calculated. Running on multicore systems, Syze is able to optimize the utilization of CPU resources by shortening the longest thread and alleviating thread contention. As a penalty, Syze places additional burdens on the cache and memory. Fortunately, the performance boost from balanced workloads far exceeds the cost from the memory-cache hierarchy. In consequence, Syze delivers substantial speedup over fine-tuned contemporary frameworks.

References

[1] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," *ACM Trans. Parallel Comput.*, vol. 8, no. 1, pp. 1–70, 2021.

[2] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 375–386.

[3] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a {PC}," in *Proc. 10th USENIX Symp. Oper. Syst. Des. Implementation*, 2012, pp. 31–46.

[4] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proc. 11th USENIX Symp. Oper. Syst. Des. Implementation*, 2014, pp. 599–613.

[5] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proc. 12th USENIX Symp. Oper. Syst. Des. Implementation*, 2016, pp. 301–316.

[6] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 527–543.

[7] K. Lakhotia, R. Kannan, S. Pati, and V. Prasanna, "Gpop: A scalable cache-and memory-efficient framework for graph processing over parts," *ACM Trans. Parallel Comput.*, vol. 7, no. 1, pp. 1–24, 2020.

[8] J. Malicevic, B. Lepers, and W. Zwaenepoel, "Everything you always wanted to know about multicore graph processing but were afraid to ask," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 631–643.

[9] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 251–262, 1999.

[10] A. T. Stephen and O. Toubia, "Explaining the power-law degree distribution in a social commerce network," *Soc. Netw.*, vol. 31, no. 4, pp. 262–270, 2009.

[11] M. Brzezinski, "Power laws in citation distributions: Evidence from scopus," *Scientometrics*, vol. 103, no. 1, pp. 213–228, 2015.

[12] E. Almaas and A.-L. Barabási, "Power laws in biological networks," in *Power Laws, Scale-Free Networks and Genome Biology*, Berlin, Germany: Springer, 2006, pp. 1–11.

[13] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. 18th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2013, pp. 135–146.

[14] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *Proc. 20th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2015, pp. 183–193.

[15] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. Twenty-Fourth ACM Symp. Oper. Syst. Princ.*, 2013, pp. 456–471.

[16] S. Grossman, H. Litz, and C. Kozyrakis, "Making pull-based graph processing performant," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 246–260, 2018.

[17] K. Lakhotia, R. Kannan, and V. Prasanna, "Accelerating pagerank using partition-centric processing," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 427–440.

[18] S. Beamer, K. Asanović, and D. Patterson, "Reducing pagerank communication via propagation blocking," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 820–831.

[19] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *Proc. IEEE Int. Conf. Big Data*, 2017, pp. 293–302.

[20] S. Zhou et al., "Design and implementation of parallel pagerank on multicore platforms," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2017, pp. 1–6.

[21] L. Dhulipala, G. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proc. 29th ACM Symp. Parallelism Algorithms Architect.*, 2017, pp. 293–304.

[22] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?," in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 591–600.

[23] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proc. 21st ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2016, pp. 1–12.

[24] "Intel® VTune$^{\text{TM}}$ profiler performance analysis cookbook," , 2011. Accessed: Apr. 22, 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top/tuning-recipes/os-thread-migration.html

[25] "Intel 64 and ia-32 architectures optimization reference manual," 2012, 2012. Accessed: Jan. 7, 2023. [Online]. Available: https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf

[26] W. Magro, P. Petersen, and S. Shah, "Hyper-threading technology: Impact on compute-intensive workloads," *Intel Technol. J.*, vol. 6, no. 1, pp. 1–9, 2002.

[27] R. A. Tau Leng, J. Hsieh, V. Mashayekhi, and R. Rooholamini, "An empirical study of hyper-threading in high performance computing clusters," *Linux HPC Revolution*, vol. 45, 2002.

[28] N. H. Qun et al., "Hyper-threading technology: Not a good choice for speeding up cpu-bound code," in *Proc. 3rd Int. Conf. Electron. Des.*, 2016, pp. 578–581.

[29] D. Buono et al., "Optimizing sparse matrix-vector multiplication for large-scale data analytics," in *Proc. Int. Conf. Supercomput.*, 2016, pp. 1–12.

[30] S. Schelter and J. Kunegis, "Tracking the trackers: A large-scale analysis of embedded web trackers," in *Proc. 10th Int. AAAI Conf. Web Soc. Media*, 2016, pp. 679–682.

[31] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proc. 7th ACM SIGCOMM Conf. Internet Meas.*, 2007, pp. 29–42.

[32] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "DBpedia: A nucleus for a web of open data," in *Proc. Int. Semantic Web Conf.*, 2007, pp. 722–735.

[33] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi, "Measuring user influence in twitter: The million follower fallacy," in *Proc. 4th Int. AAAI Conf. Weblogs Soc. Media*, 2010, pp. 10–17.

[34] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 2015, *arXiv:1508.03619*.

[35] J. Kunegis, "Konect: The koblenz network collection," in *Proc. 22nd Int. Conf. World Wide Web*, 2013, pp. 1343–1350.

[36] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 1998, pp. 212–223.

[37] "perf: Linux profiling with performance counters," 2022. Accessed Jan. 7, 2023. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[38] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proc. 39th Int. Conf. Parallel Process. Workshops*, 2010, pp. 207–216.

[39] "OpenCilk," Accessed: Sep. 21, 2021, 2011. [Online]. Available: https://cilkplus.github.io/

[40] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford InfoLab, Stanford, CA, USA, Tech. Rep. 442, 1999.

[41] N. Sundaram et al., "Graphmat: High performance graph analytics made productive," 2015, *arXiv:1503.07241*.

[42] G. Malewicz et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.

[43] C. Martella, R. Shaposhnik, D. Logothetis, and S. Harenberg, *Practical Graph Analytics With Apache Giraph*, vol. 1, Berlin, Germany: Springer, 2015.

[44] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[45] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "Graphgrind: Addressing load imbalance of graph partitioning," in *Proc. Int. Conf. Supercomput.*, 2017, pp. 1–10.

[46] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proc. 24th ACM Symp. Oper. Syst. Princ.*, 2013, pp. 472–488.

[47] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From," think like a vertex" to" think like a graph,"" *Proc. VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.

[48] Y. Simmhan et al., "Goffish: A sub-graph centric framework for large-scale graph analytics," in *Proc. Eur. Conf. Parallel Process.*, 2014, pp. 451–462.

[49] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1981–1992, 2014.

[50] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: A dsl for easy and efficient graph analysis," in *Proc. 17th Int. Conf. Architect. Support Program. Lang. Oper. Syst.*, 2012, pp. 349–362.

[51] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph DSL," *Proc. ACM Programm. Lang.*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[52] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1–15.

[53] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng, "Exploring the hidden dimension in graph processing," in *Proc. 12th USENIX Symp. Oper. Syst. Des. Implementation*, 2016, pp. 285–300.

[54] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Symp. Oper. Syst. Des. Implementation*, 2012, pp. 17–30.

[55] M. H. Mofrad, R. Melhem, Y. Ahmad, and M. Hammoud, "Graphite: A NUMA-aware HPC system for graph analytics based on a new MPI* X parallelism model," *Proc. VLDB Endowment*, vol. 13, no. 6, pp. 783–797, 2020.

[56] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," in *Algorithm Engineering*, Berlin, Germany: Springer, 2016, pp. 117–158.

[57] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *J. Parallel Distrib. Comput.*, vol. 48, no. 1, pp. 96–129, 1998.

[58] H. Meyerhenke, P. Sanders, and C. Schulz, "Partitioning complex networks via size-constrained clustering," in *Proc. Int. Symp. Exp. Algorithms*, 2014, pp. 351–363.

[59] G. M. Slota, K. Madduri, and S. Rajamanickam, "PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 481–490.

[60] G. Karypis and V. Kumar, "Parallel multilevel series k-way partitioning scheme for irregular graphs," *SIAM Rev.*, vol. 41, no. 2, pp. 278–300, 1999.

[61] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 9, pp. 2625–2638, Sep. 2017.

[62] G. M. Slota, C. Root, K. Devine, K. Madduri, and S. Rajamanickam, "Scalable, multi-constraint, complex-objective graph partitioning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 12, pp. 2789–2801, Dec. 2020.

YuAng Chen received the BS degree in electronic science and technology from the Huazhong University of Science and Technology, in 2015, and the dual MS degrees in embedded system from the Eindhoven University of Technology and Technische Universität Berlin, in 2018. Currently, he is working toward the PhD degree in computer science with the Chinese University of Hong Kong, Shenzhen. His research interests cover computer system & architecture, high performance computing and graph analytics.

Yeh-Ching Chung received the BS degree in computer science from Chung Yuan Christian University, in 1983, and the MS and PhD degrees in computer and information science from Syracuse University, in 1988 and 1992, respectively. Currently, he is a professor with the Chinese University of Hong Kong, Shenzhen. His research interests include parallel and distributed processing, cloud computing, Big Data, and embedded system.