ARTICLE IN PRESS

J. Parallel Distrib. Comput. ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

# Exploring big graph computing — An empirical study from architectural perspective

Lifeng Nai [a,*], Yinglong Xia [b], Ilie G. Tanase [c], Hyesoon Kim [a]

[a] Georgia Institute of Technology, Atlanta, GA, United States
[b] Huawei Research America, Santa Clara, CA, United States
[c] IBM T.J. Watson Research Center, Yorktown Heights, NY, United States

## HIGHLIGHTS

- We from the architectural perspective study behaviors of graph computing in real world use cases.
- We conduct comprehensive experiments to collect quantitative characteristics.
- Our characterizations include multiple architectural factors and data impact studies.
- Our explorations can deepen our understanding in graph computing.

## ARTICLE INFO

*Article history:*
Received 16 July 2015
Received in revised form
9 July 2016
Accepted 19 July 2016
Available online xxxx

*Keywords:*
Graph computing
Architectural characterization
Big data

## ABSTRACT

Graph computing is widely applied in a large number of big data applications. Despite its importance, high performance graph computing remains a challenge, especially for large-scale graphs. In this paper, by analyzing from the architectural perspective, we study computational behaviors of graph computing in real-world use cases. We benchmark a set of representative graph algorithms implemented on a unified framework and conduct experiments to analyze comprehensive performance characteristics. In the characterization, we observed multiple insights, including irregular memory patterns, significant diverse behavior across different computations, highly data dependent behaviors, etc., using large-scale synthetic and real-world graphs. To the best of our knowledge, this is the first comprehensive architectural study on the full-scope of graph computing. It can improve our understanding on graph computing and help high performance computing research for graph-based big data applications.



## 1. Introduction

In many big data scenarios, information from various entities is typically linked with others and forms a large-scale graph. Graph computing has become one of the most important techniques for processing, analyzing, and visualizing linked big data [31,37].

Graph computing explores graph topology and/or the attributes associated with the vertices and edges. It leads to many research topics, ranging from graph-oriented computer architecture design to massive graph analytics and visualization. There are numerous research efforts across multiple communities invested in this discipline [26]. Although graph theory and graph analytic algorithms are well studied in prior literature, much less attention is paid to the performance of graph computing [49,47]. To the best of our knowledge, other than high performance implementations of specific graph algorithms, the performance characteristics of a wide selection of graph computing workloads are insufficiently discussed [34,50]. Unlike many prior work that focus only on graph structures, the attributes (a.k.a. properties) of graph vertices and/or edges should be equally addressed, especially when rich/dynamic properties are involved [5]. Besides, the computing platform is becoming heterogeneous. More than just parallel graph computing on CPUs, there is a growing impact of graph computing on Graphic Processing Units (GPUs).

To understand the characteristics of graph computing, we must investigate multiple key performance factors, such as frameworks, data representations, computation types, and data sources [5]. First, in a graph computing system, elementary graph operations, such as find-vertex and add-edge, shall be supported via a unified underlying framework because of programmability and

* Corresponding author.
*E-mail address:* nailifeng@gmail.com (L. Nai).

http://dx.doi.org/10.1016/j.jpdc.2016.07.006
0743-7315/

usability considerations. The separation of user program and graph framework simplifies complexity of user programs and ensures graph applications to be independent from framework changes. These graph operations contribute to a large portion of the total execution time and thus, their implementation significantly impacts the performance. Similarly, the data representations can also affect multiple architectural features, especially the memory sub-system behavior. Second, although graph traversals are considered as representative graph applications, graph computing has a much broader scope. Graph applications can involve computations not only on graph structures, but also on the rich vertex/edge properties or dynamic graphs (refer to Section 2 for further details). Third, graph processing systems are also data-dependent. Data sources may significantly impact its behavior.

There are some graph workloads in existing architectural benchmark suites, demonstrating the inefficiency of graph computing on conventional CPU/GPU architectures [12,45,32,4]. However, most of them target generic benchmarking purposes and have limited graph workload number. Besides, multiple recent research efforts are ongoing for benchmarking existing graph processing systems, such as LDBC [23], Graphalytics [7], and GraphBench [14]. They target system-level comparison and evaluation, and thus, are less applicable when it comes to architecture-level analysis. Little is known about the architectural characteristics of graph processing systems with realistic frameworks.

To understand the behavioral characteristics of a wide scope of graph computing applications, we propose a comprehensive exploration of our previously proposed benchmark suite, *GraphBIG*, on contemporary hardware. GraphBIG is mainly inspired by the IBM System G framework, which is a comprehensive set of industrial graph computing toolkits used by many commercial use case scenarios [17,43], but also partially considers Dato's GraphLab [27], Google's Pregel [29], and Microsoft's Trinity [40]. By utilizing the middleware design of System G, we ensure a realistic framework and data representation in GraphBIG. Meanwhile, the workloads are selected comprehensively from real-world use cases of multiple application domains. In addition, GraphBIG provides real-world datasets to cover major graph data sources and a synthetic dataset for characterization purposes.

In our characterization, we observe the following behavior. (1) On average, modern processors show extremely low IPC for graph computing workloads because of significant inefficiencies in memory sub-systems. (2) Diverse behavior is also observed in different workloads and different computation types. Such diversity exists in cache hit rate, DTLB miss, branch miss, and overall performance. (3) Although low hit rates are observed in L2 and L3 caches, the L1D cache still can get a significant amount of hits from the meta data with small sizes. (4) Despite the performance differences introduced by computation models, they all share the same inefficiencies in the cache hierarchy. (5) Graph computing with big graphs also shows similar behavioral trends as medium-size graphs. (6) Graph data size, density, and connectivity all have significant impact on multiple architecture features. Such impact has complex correlations with workloads.

The main contributions of this paper are as follows:

- We present the first comprehensive architectural study on the behavioral characteristics of a wide selection of graph computing workloads using industrial graph frameworks.
- We analyze the irregularity of graph computing. Our results indicate high L2/L3 cache miss rates. However, L1D cache and ICache both show a relatively low miss rate because of the locality of non-graph data and the flat code hierarchy of the underlying framework respectively.
- We investigate the workload diversity and observe a significantly diverse architectural behavior across different graph computation types.
- We explore the data sensitivity and observe that graph workloads exhibit varying but consistently high degree of data sensitivity.

The rest of this paper is organized as follows. In Section 2, we discuss and summarize the key performance factors of graph computing behavior. Section 3 introduces previous related work. In Section 4, we characterize the workloads from multiple perspectives on CPU. Then, in Section 5, we analyze the impact of graph data. Finally, in Section 6, we conclude our work.

## 2. Graph computing: key performance factors

In real-world practices, graph computing contains a broad scope of use cases, from cognitive analytics to data exploration. The wide range of use cases introduces not only unique, but also diverse features of graph computing, including frameworks, data representations, computation types, and data sources. To understand graph computing in a holistic way, we first analyze these key performance factors of graph computing behavior in this section.

**Framework**: Unlike standalone prototypes of graph algorithms, graph computing systems largely rely on specific frameworks to achieve various functionalities because of programmability and usability concerns. By hiding the details of managing graph data and requests, the graph frameworks provide users primitives for elementary graph operations. The separation of user program and graph framework simplifies complexity of user programs and ensures graph applications independent from framework changes. The examples of graph computing frameworks include GraphLab [27], Pregel [29], Apache Giraph [2], and IBM System G. Although they have different management and synchronization mechanisms, they share significant similarity in their computation models and user primitives. First, unlike simplified algorithm prototypes, graph systems represent graph data as a *property graph*, which associates user-defined properties with each vertex and edge. The properties can include meta-data (e.g., user profiles), program state (e.g., vertex status in BFS or graph coloring), and even complex probability table (e.g., Bayesian inference). Second, instead of directly operating on graph data, the user defined applications achieve their algorithms via framework-defined primitives, which usually include find/delete/add vertices/edges, traverse neighbors, update properties, etc.

To estimate the framework's impact on the graph system performance, we performed profiling experiments on a series of typical graph workloads with IBM System G framework. As shown in Fig. 1, a significant portion of time is contributed by the framework for most workloads, especially for graph traversal based ones. On average, the in-framework time is as high as 76%. This is because the key component of most workloads is graph data accesses, which are supported by the framework-enabled primitives and are implemented within the framework. Fig. 1 shows that the heavy reliance on the framework indeed results in a large portion of in-framework execution time. It therefore can bring significant impact on the architecture behavior of the upper layer graph workloads.

**Data representation**: Within the graph frameworks, various data representations can be incorporated for organizing in-memory graph data. The differences between in-memory data representations can significantly affect the architectural behavior, especially memory sub-system related features, and eventually impact the overall performance.

One of the most popular data representation structures is Compressed Sparse Row (CSR). As illustrated in Fig. 2(a) (b), CSR organizes vertices, edges, and properties of graph G in separate compact arrays. (Variants of CSR also exist. For example,

Please cite this article in press as: L. Nai, et al., Exploring big graph computing — An empirical study from architectural perspective, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.07.006

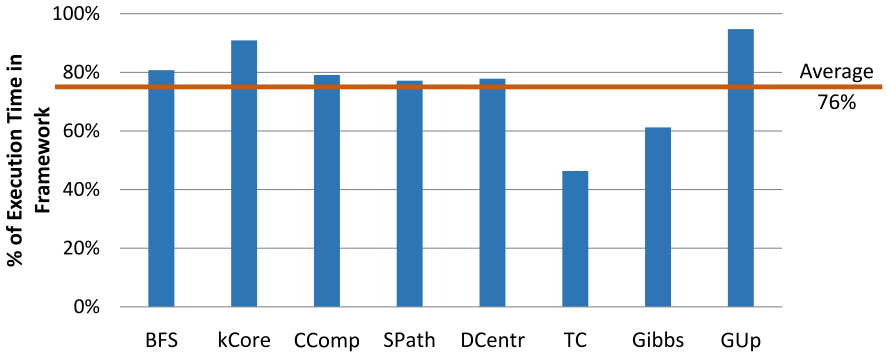



**Fig. 1.** Execution time of framework.

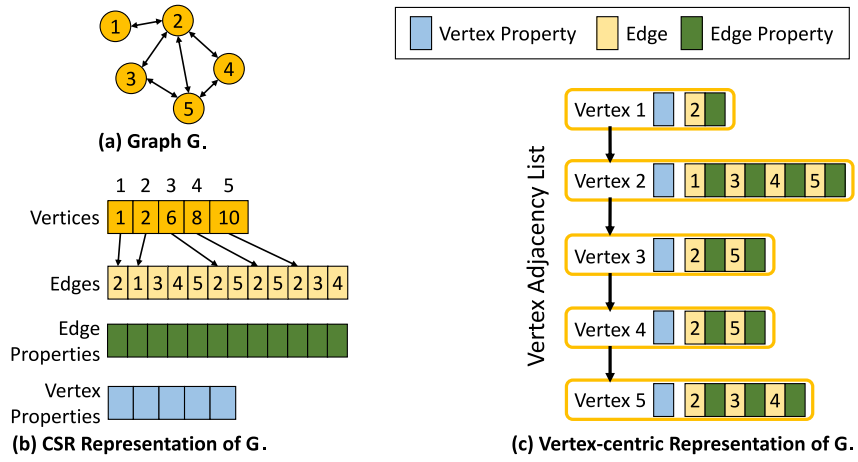



**Fig. 2.** Illustration of data representations. (a) graph G, (b) its CSR representation, and (c) its vertex-centric representation.

**Table 1**
Graph computation type summary.

| Graph computation type | Feature | Example |
|---|---|---|
| Computation on graph structures (CompStruct) | Irregular access pattern, heavy read accesses | BFS traversal |
| Computation on graphs with rich properties (CompProp) | Heavy numeric operations on properties | Belief propagation |
| Computation on dynamic graphs (CompDyn) | Dynamic graph structure, dynamic memory footprint | Streaming graph [10] |

Coordinate List (COO) format replaces the vertex array in CSR with an array of source vertices of each edge.) Comparing other dynamic data structures with indices, the compact format of CSR saves memory size and simplifies graph build/copy/transfer complexity. It is widely used in multiple systems. However, the compact data structure of CSR also brings significant data movement overhead if structural updates happen. Many graph applications incorporate only static graph structures. Nevertheless, in real-world graph systems, there are also many cases that require high dynamicity in both topologies and properties. Thus, flexible data representations are more desirable in many graph systems. For example, IBM System G, as well as multiple other frameworks, is using a vertex-centric structure, in which a vertex is the basic unit of a graph. As shown in Fig. 2(c), the vertex property and the outgoing edges stay within the same vertex structure. Meanwhile, all vertices form up an adjacency list with indices. Other graph representations also exist in academic research literature, though not widely adopted in industrial systems. Examples include the subgraph-centric [42] and edge-centric framework [38].

**Computation types**: Numerous graph applications exist in prior literature and real-world practices. Despite the variance of their implementation details, generally graph computing applications can be classified into a few computation types [48]. As shown in Table 1, we summarize the applications into three categories according to their different computation targets: graph structures, graph properties, and dynamic graphs. Because of the differences in computation targets, they have diverse features. (1) Computation on the graph structure traverses the graph through structures, such as the Breadth-first Search. It incorporates a large number of memory accesses and limited numeric operations. Their irregular memory access pattern leads to extremely poor spatial locality. (2) On the contrary, computation on graphs with rich properties performs computation within each vertex/edge's property, which is the attributes attached to vertex/edge that can be even as rich as a full stochastic table, such as in Gibbs Inference. Triangle Count is also an example because in its typical implementation, neighbor list is handled just as vertex property and computation is only based on the properties with non-traversal operations. This computation type introduces lots of numeric computations on properties, which leads to behavior similar as conventional applications. (3) For computation on dynamic graphs, it involves dynamic graph structure updates as well as graph traversals. Thus, it also shows an irregular pattern as the first computation type. However, the updates of graph structures lead to high write intensity and dynamic memory footprint.

**Graph data sources**: As a data-centric computing tool, graph processing systems heavily rely on data inputs. As shown in

Please cite this article in press as: L. Nai, et al., Exploring big graph computing — An empirical study from architectural perspective, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.07.006

**Table 2**
Graph data source summary.

| No. | Graph data source | Example | Feature |
|---|---|---|---|
| 1 | Social(/economic/political) network | Twitter graph | Large connected components, small shortest path lengths |
| 2 | Information(/knowledge) network | Knowledge graph | Large vertex degrees, large small-hop neighborhoods |
| 3 | Nature(/bio/cognitive) network | Gene network | Complex properties, structured topology |
| 4 | Man-made technology network | Road network | Regular topology, small vertex degrees |

Table 2, we summarize graph data into four sources [48]. The social network represents the interactions between individuals/organizations. The key features of social networks include high degree variances, small shortest path lengths, and large connected components [33]. On the contrary, an information network is a structure, in which the dominant interaction is the dissemination of information along edges. It usually shows large vertex degrees, and large two-hop neighborhoods. The nature network is used for learning and interacting naturally with people. Examples include deep belief network (DBN) [3] and biological network [8]. They typically incorporate structured topologies and rich properties addressing different targets. Man-made technology networks are formed by specific man-made technologies. A typical example is a road network, which usually maintains small vertex degrees and a regular topology.

## 3. Related work

Several graph computing frameworks have been proposed previously. Examples include Pregel [29], Giraph [2], Trinity [40], PEGASUS [20], GraphLab [27], and Cassovary [16]. There are also multiple academic research efforts, such as GraphChi [22], X-stream [38], Cusha [21], and Mapgraph [13]. They incorporate various techniques to achieve different optimization targets on specific platforms. For example, GraphChi utilizes a Parallel Sliding Window (PSW) technique to optimize disk IO performance. Cusha extends the similar technique on GPU platforms to improve data access locality. For similar purposes, X-stream proposes a graph system using an edge-centric programming model.

Multiple system-level benchmarking efforts are also ongoing for evaluating and comparing existing graph systems. Examples include LDBC benchmark, GraphBench, G. Yong's characterization work [15], and A. L. Varbanescu's study [44]. To understand the architectural behavior of graph computing, multiple graph benchmarks exist. For example, as one of the most famous graph benchmarks, Graph 500 [32] was proposed for system performance ranking purposes. Reference codes as graph generators exist in Graph 500. However, because of its special purpose and small workload number, it is less feasible for benchmarking graph computing. CloudSuite [12] and BigDataBench [45] target cloud computing and big data computing respectively. Graph workloads are included in their packages. Similarly, PBBS [41] targets evaluations of parallel programming methodologies. Graph computing is also one of its components.

As one of the representative industrial graph solutions, IBM System G is another example. It is a comprehensive set of graph computing software systems for Big Data portfolio. It includes a wide range of toolkits, use cases, and graph data sources. Meanwhile, by utilizing System G's framework and use case resources, a comprehensive benchmarking effort, GraphBIG [35], was also proposed. It covers the whole scope of graph computing and provides a rich support for architectural studies.

Most prior benchmarking and characterization efforts focus on the system level analysis, in which only system performance factors are analyzed, lacking in-depth study from architectural perspective. Besides, the architectural benchmarks with graph workloads target generic benchmarking purposes. They are less focused on revealing the comprehensive characteristics of graph computing. Moreover, most of existing benchmarks are biased to graph traversal related workloads (CompStruct). The other two graph computation types, computation on dynamic graphs and on rich properties, are less emphasized. Comparing with prior efforts, our proposed study provides a comprehensive analysis on the architectural behavior of graph computing. It covers the wide scope of graph computing unbiasedly and further incorporates the analysis of graph data impact. The study enables an in-depth understanding of graph computing from architectural perspective.

**Table 3**
Test machine configurations.

| | | |
|---|---|---|
| Processor | Type | Xeon E5-2670 |
| | Frequency | 2.6 GHz |
| | Core # | 2 sockets × 8 cores × 2 threads |
| | Cache | 32 kB L1, 256 kB L2, 20 MB L3 |
| | MemoryBW | 51.2 GB/s (DDR3) |
| System | Memory | 192 GB |
| | Disk | 2 TB HDD |
| | OS | Red Hat Enterprise Linux 6 |

## 4. Graph computing characterization

### 4.1. Characterization methodology

In this study, our exploration focuses on single node architectural behavior. Although real-world big graphs can contain enormous amount of data on a cluster of machines, the graph structure, which is the main focus of most graph applications, still mostly stays within the capacity of one single node [22]. Meanwhile, the memory capacity of a single node keeps increasing over the past years. For an instance, the newly released graph system from Neo4j with IBM POWER8 incorporates 56 TB memory in one node [36]. Moreover, as illustrated in GraphChi [22], because of the significant communication overhead, the performance of hundreds of nodes can be even worse than one optimized single node because of the communication overhead. Therefore, even in a distributed computing environment, it is still critical to scale-up the single node performance before scale-out computations.

**Hardware configurations**: In our characterization, we perform our experiments on a single node Intel Xeon machine with 2 sockets and 8 cores in each. The hardware and OS details are shown in Table 3. Our target framework, GraphBIG (further explained in the later subsection), is following Bulk Synchronous Parallel (BSP) with symmetric parallel threads on one single node. To avoid the context switch overhead introduced by OS thread scheduling, the threads are pinned to different hardware cores with Simultaneous Multithreading (SMT) disabled. In addition, the thread-pinning will prioritize the full-utilization of cores in one socket to reduce cross-socket communication.

**Workloads**: To address the key factors of graph computing behavior in our characterization, we use benchmarks from GraphBIG [35], which is a comprehensive benchmark suite with both CPU and GPU workloads. By utilizing an industrial framework design, GraphBIG incorporates a modern graph framework and data representation. Moreover, to ensure representativeness and coverage, GraphBIG selects workloads from real-world use cases and covers three major graph computation types.

Please cite this article in press as: L. Nai, et al., Exploring big graph computing — An empirical study from architectural perspective, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.07.006

**Table 4**
Characterization workloads.

| Category | Workload | Computation type |
|---|---|---|
| Graph traversal | Breadth-first search (BFS) | CompStruct |
| | Depth-first search (DFS) | CompStruct |
| Graph update | Graph construction (GCons) | CompDyn |
| | Graph update (GUp) | CompDyn |
| | Topology morphing (TMorph) | CompDyn |
| Graph analytics | Shortest path (SPath) | CompStruct |
| | K-core decomposition (kCore) | CompStruct |
| | Connected component (CComp) | CompStruct |
| | Triangle count (TC) | CompProp |
| | Gibbs inference (GI) | CompProp |
| Social analysis | Degree centrality (DCentr) | CompStruct |
| | Betweenness centrality (BCentr) | CompStruct |

In our experiments, we select 12 CPU workloads from GraphBIG. The workload details are shown in Table 4. They cover multiple usage types, from conventional graph traversal to graph update, and three graph computation types. Most workloads, except for Graph update workloads, are performing parallel processing with fully-utilization of hardware platform resources. In addition, for iterative workloads with dynamic working set size, GraphBIG incorporates a task queue for each thread. The task queue of next iteration will be generated dynamically according to the hashing outcome of new task vertices' id.

As summarized in Table 4, the workloads are grouped into four categories according to their high level usage. The details are further explained below.

- *Graph traversal*: Two workloads—Breadth-first Search (BFS) and Depth-first Search (DFS) are selected. Both are widely-used graph traversal operations.
- *Graph construction/update*: Graph update workloads are performing computations on dynamic graphs. Three workloads are included as follows. Graph construction (GCons) constructs a directed graph with a given number of vertices and edges. Graph update (GUp) deletes a given list of vertices and related edges from an existing graph. Both GCons and GUp achieve the vertex/edge add/delete operations via the underlying framework, in which a vertex-centric data representation with extra indexes is used. Topology morphing (TMorph) generates an undirected moral graph from a directed-acyclic graph (DAG) [6]. It involves graph construction, graph traversal, and graph update operations.
- *Graph analytics*: There are three groups of graph analytics, including topological analysis, graph search/match, and graph path/flow. Since basic graph traversal workloads already cover graph search behavior, here we focus on topological analysis and graph path/flow. As shown in Table 4, five chosen workloads cover the two major graph analytic types and two computation types. In their implementations, the shortest path is following parallel Bellman–Ford algorithm. The k-core decomposition is using Matula & Beck's algorithm [30]. The connected component is implemented with BFS traversals. The triangle count is based on T. Schank's algorithm [39]. Besides, the Gibbs inference is performing Gibbs sampling for approximate inference in Bayesian networks.
- *Social analysis*: Due to its importance, social analysis is listed as a separate category in our work, although generally social analysis can be considered as a special case of generic graph analytics. We select graph centrality to represent social analysis workloads. Since closeness centrality shares significant similarity with shortest path, we include the betweenness centrality with Brandes' algorithm [28] and degree centrality [19].

**Table 5**
Graph data in the experiments.

| Experiment dataset | Vertex # | Edge # |
|---|---|---|
| Bitcoin graph | 72M | 182M |
| LDBC synthetic graph | 1M | 28.82M |
| Twitter graph | 11M | 85M |
| IBM knowledge Repo | 154k | 1.72M |
| IBM Watson gene graph | 2M | 12.2M |
| CA road network | 1.9M | 2.8M |
| MUNIN graph | 1041 | 1397 |

**Datasets**: In the characterization experiments, we first use synthetic graph data to enable in-depth analysis for multiple architectural features. The LDBC synthetic graph is selected because it represents social network behavior and offers flexibility in graph size. Four other datasets are then included for real-world data studies (see Table 5). To analyze the large graph behavior, the Bitcoin graph is included in characterization. In addition, because of the special computation requirement of Gibbs Inference workload, the Bayesian network MUNIN [1] is used. It includes 1041 vertices, 1397 edges, and 80 592 parameters.

**Profiling method**: In our experiments, the hardware performance counters are used for measuring detailed hardware statistics [51,12,45]. In total, around 30 hardware counters are collected by following the guideline of Intel Manual [18]. We designed our own profiling tool embedded within the benchmarks instead of using existing profiling tools. This is because instead of profiling the whole workload with sampling, we want to profile only the particular code sections of interest and avoid inaccurate results caused by sampling error or multiplexing of performance counters. Our profiling tool is utilizing the perf_event interface of Linux kernel [46] for accessing hardware counters and the libpfm library for encoding hardware counters from event names [11].

**Metrics**: In the experiments, we are following a hierarchical profiling strategy. Multiple metrics are utilized to analyze the architectural behavior as shown in Table 6. *Execution cycle breakdown* is first analyzed to understand the bottleneck of workloads. The breakdown categories include *frontend* stall, *backend* stall, *retiring*, and *bad speculation* [18]. The frontend represents the processor stalls because of frontend issues, which include instruction fetch, decode, and allocate. Similarly, the backend represents the stalls because of backend issues, which include instruction rename, schedule, execution, and commit. The bad speculation in the wasted cycles is because of wrong branch prediction. Hence, except for retiring, which is the actual efficient execution time, the other three categories are all corresponding to wasted processor cycles. *Cache MPKI* (miss per kilo instructions) is then analyzed to understand memory sub-system behavior. It reflects the cache hierarchy performance. We estimated the MPKI values of L1D, L2, and LLC (last-level cache). In addition, we also measured multiple other metrics, including *IPC* (instruction per cycle), *branch miss rate, ICache miss rate, and DTLB penalty* [51,12,45]. These metrics cover major architectural features of modern processors.

### 4.2. Workload characterization

In this section, we characterize GraphBIG workloads with a top-down characterization strategy. The results are explained as follows.

**Execution time breakdown**: The execution time breakdown is shown in Fig. 3 and grouped by computation types. As explained in previous section, the Frontend and Backend represent the frontend bound and backend bound stall cycles respectively. The BadSpeculation is the cycles spent on wrong speculations, while the Retiring is the cycles of successfully retired instructions. It is a

Please cite this article in press as: L. Nai, et al., Exploring big graph computing — An empirical study from architectural perspective, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.07.006

**Table 6**
Architectural metric summary.

| Metric | Description |
|---|---|
| Frontend | Frontend stall cycles caused by instruction fetch/decode/allocate |
| Backend | Backend stall cycles caused by instruction rename/schedule/execution/commit |
| BadSpeculation | Wasted cycles because of wrong branch prediction |
| Retiring | Actually efficient execution cycles |
| Cache MPKI | Cache miss per kilo instructions.<br>Higher MPKI brings more memory related backend stalls |
| ICache miss | Instruction cache miss.<br>Higher ICache Miss brings more frontend stalls |
| DTLB penalty | Penalty cycles caused by DTLB misses. It also includes page-table walking and page-fault handling time. |
| IPC | Instruction per cycle. It represents the efficiency of processor execution |
| Branch miss | Wrong prediction of branches. It brings wasted execution cycles on wrong branch path. |

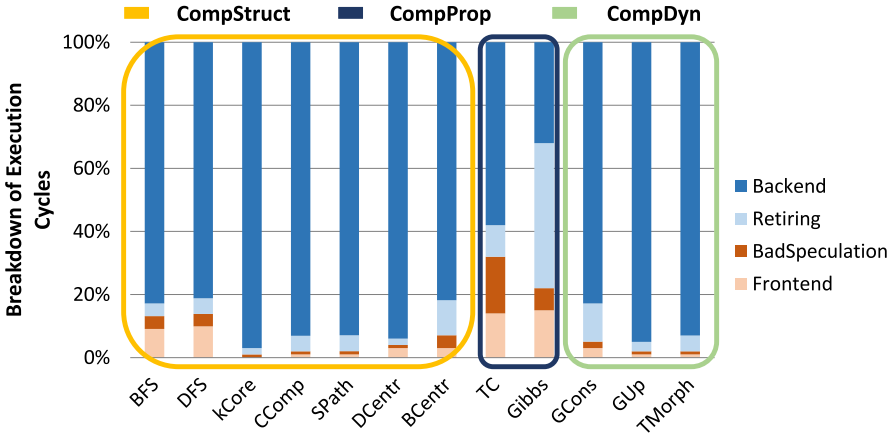



**Fig. 3.** Execution time breakdown of GraphBIG CPU workloads.

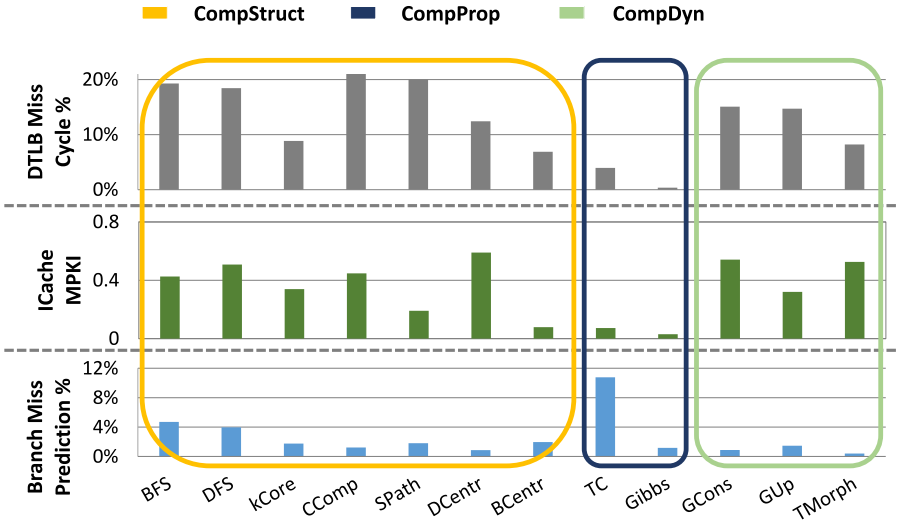



**Fig. 4.** DTLB penalty, ICache MPKI, and branch miss rate of GraphBIG CPU workloads.

common intuition that irregular data accesses are the major source of inefficiencies of graph computing. The breakdown of execution time also supports such intuition. It is shown that the backend indeed takes dominant time for most workloads. In extreme cases, such as kCore and GUp, the backend stall percentage can be even higher than 90%. However, different from the simple intuition, the outliers also exist. For example, the workloads of computation on rich properties (CompProp) category show only around 50% cycles on backend stalls. The variances between computation types further demonstrate the necessity of covering different computation types.

**Core analysis**: Although execution stall can be triggered by multiple components in the core architecture, instruction fetch and branch prediction are usually the key inefficiency sources. In previous literature, it was reported that many big data workloads, including graph applications, suffer from high ICache miss rate [12,45]. However, in our experiments, we observe different outcomes. As shown in Fig. 4, the ICache MPKI of each workload all show below 0.7 values, though small variances still exist. We believe the different ICache performance values result from the design differences of the underlying frameworks. Open-source big data frameworks typically incorporate multiple external libraries and tools. Meanwhile, the included libraries may further utilize other libraries. Thus, it eventually results in deep software stacks, which lead to complex code structures and high ICache MPKI. However, such behavior is highly implementation dependent. In

Please cite this article in press as: L. Nai, et al., Exploring big graph computing — An empirical study from architectural perspective, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.07.006

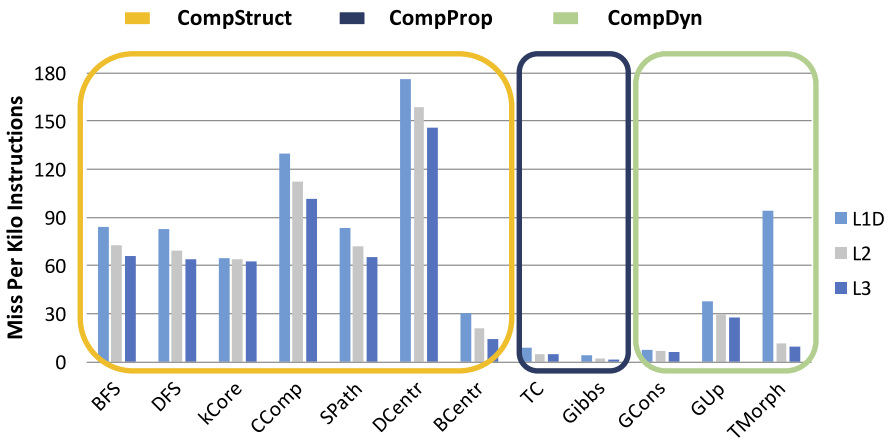



**Fig. 5.** Cache MPKI of GraphBIG CPU workloads.

GraphBIG, the underlying framework is following the design of IBM System G, in which minimum external libraries are included and a flat software hierarchy is incorporated. Hence, a low ICache MPKI is observed.

The branch prediction also shows low miss prediction rate in most workloads except for TC, which reaches as high as 10.7%. The workloads from other computation types show a miss prediction rate below 5%. The difference comes from the special intersection operations in TC workload. It is also in accordance with the above breakdown result, in which TC consumes a significant amount of cycles in BadSpeculation.

The DTLB miss penalty is shown in Fig. 4. The cycles wasted on DTLB misses are more than 15% of total execution cycles for most workloads. On average, it still takes 12.4%. The high penalty is caused by two sources. One is the large memory footprint of graph computing applications, which cover a large number of memory pages. Another is the irregular access pattern, which incorporates extremely low page locality. Diversity among workloads also exists. The DTLB miss penalty reaches as high as 21.1% for Connected Component and as low as 3.9% for TC and 1% for Gibbs. This is because for computation on properties, the memory accesses are centralized within the vertices. Thus, low DTLB-miss penalty time is observed.

**Cache performance**: As shown in previous sections, cache plays a crucial role in graph computing performance. In Fig. 5, the MPKI of different levels of caches are shown. On average, a high L3 MPKI is shown, reaching as high as 48.77. Degree Centrality and Connected Component show even higher MPKI, which are 145.9 and 101.3 respectively. For computations on the graph structures (CompStruct), a generally high MPKI is observed. On the contrary, CompProp shows an extremely small MPKI value compared with other workloads. This is in accordance with its computation features, in which memory accesses happen mostly inside properties with a regular pattern. The workloads of computation on dynamic graphs (CompDyn) introduce diverse results, ranging from 6.3 to 27.5 in L3 MPKI. This is because of the diverse operations of each workload. The GraphConstruct adds new vertices/edges, while the GraphUpdate mostly deletes them. The TMorph involves both operations. Meanwhile, unlike other workloads, TMorph includes no small size local queues/stacks, leading to a high MPKI in L1D cache. However, its graph traversal pattern results in relatively good locality in L2 and L3.

**Computation type behavior**: By averaging the architectural behavior from Figs. 4 and 5 by computation types, we can observe significant diversity across computation types as shown in Fig. 6. Although variances exist within each computation type, the average results demonstrate their diverse features. The CompStruct shows significantly higher MPKI and DTLB miss penalty values because of its irregular access pattern when traversing through graph structures. Low and medium MPKI and DTLB values are shown in CompProp and CompDyn respectively. Similarly, the CompProp suffers from a high branch miss rate while other two types do not. In the IPC results, CompStruct achieves the lowest IPC value due to the penalty from cache misses. On the contrary, CompProp shows the highest IPC value. The IPC value of CompDyn stays between them. Such feature is in accordance with their access patterns and computation types.

**Real-world dataset characterization**: To compare the characteristics of different input datasets, we performed experiments on four real-world datasets from different types of sources and the LDBC synthetic data. (We excluded the workloads that cannot take all input datasets and redundant workloads that show same architectural behavior.)

Despite the extremely low L2/L3 hit rates, Fig. 7 shows relatively higher L1D hit rates for almost all workloads and datasets. This is because graph computing applications all incorporate multiple small size structures, such as task queues and temporal local variables. The frequently accessed meta data, such as small-size local variables and task queue, introduces a large amount of L1D cache hits except for DCentr, in which there is only a limited number of meta data accesses. From the results in Fig. 7, we can also see that twitter data shows highest DTLB miss penalty in most workloads. Such behavior eventually turns into lowest IPC values in most workloads except SPath, in which higher L1D cache hit rate of the twitter graph helps performance significantly. Triangle Count (TC) achieves highest IPC with the knowledge dataset, because of its high L2/L3 hit rate and low TLB penalty. The high L3 hit rate of the Watson data also results in a high IPC value. However, the twitter graph's high L3 hit rate is offsetted by its extremely high DTLB miss cycles, leading to the lowest IPC value. The diversity is caused by the different topological and property features of the real-world data sets. It is clearly shown that significant impact is introduced by the graph data on overall performance and other architectural features.

**Framework Comparison**: As illustrated in Fig. 1, the underlying framework of graph system can bring significant impact on the overall performance. In our study, we focus on the GraphBIG benchmark suite, which is using an IBM System G-like framework. To understand and compare the impact of different computation models of various frameworks, we also performed a series of experiments to compare architectural features of multiple computation models, including GraphBIG, Pregel, and GAS. Pregel is a BSP-based framework and programming model proposed by Google [29]. As one of the first graph frameworks, it is widely adopted in multiple industrial and academic systems. GAS is a graph framework first utilized in GraphLab [27] project. It follows

Please cite this article in press as: L. Nai, et al., Exploring big graph computing — An empirical study from architectural perspective, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.07.006

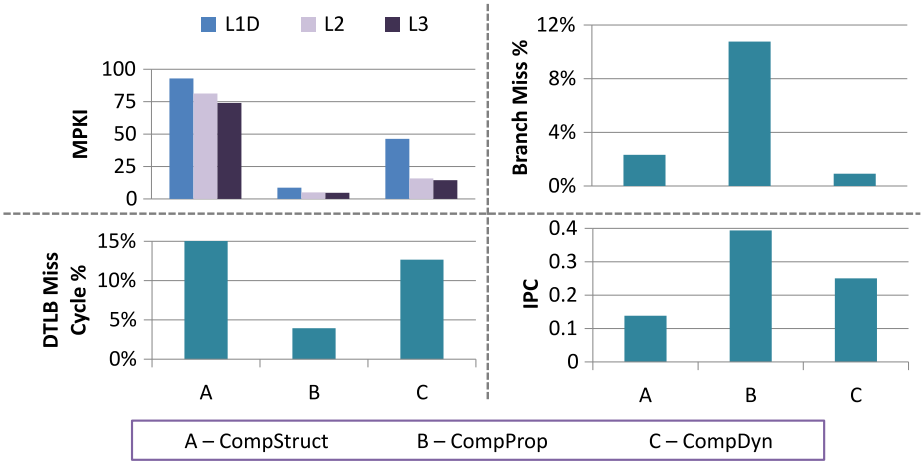



**Fig. 6.** Average behaviors of GraphBIG CPU workloads by computation types.

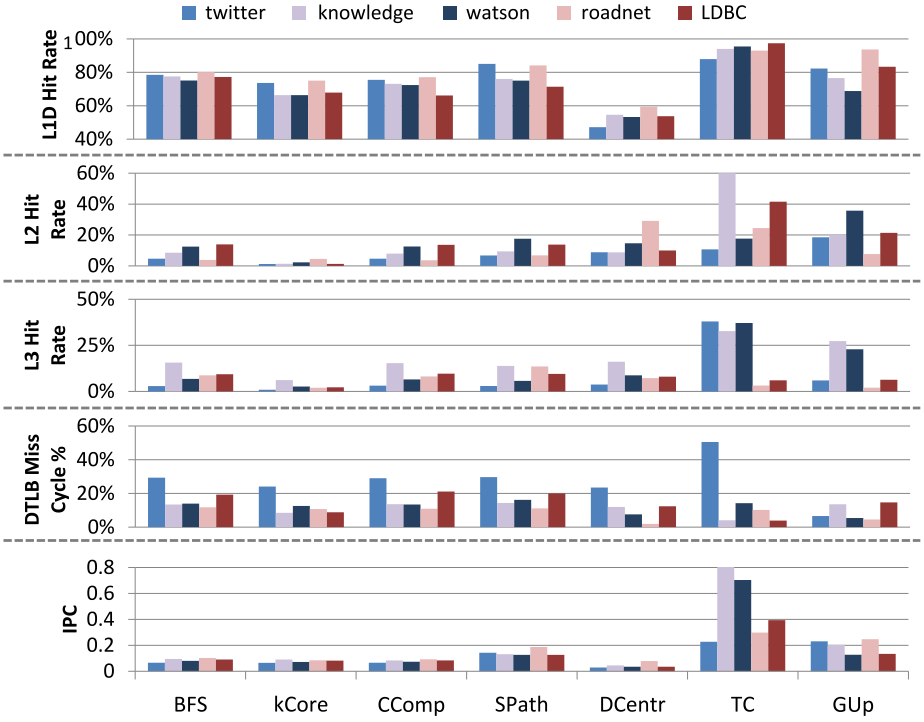



**Fig. 7.** Cache hit rate, DTLB penalty, and IPC of GraphBIG CPU workloads with different data sets.

a gather-apply-scatter model, which supports asynchronous graph processing. Since our focus in this comparison is to understand the impact of each computation model's design methodology, not implementation details or graph data layouts, we choose to implement both Pregel and GAS models by utilizing the GraphBIG data representation. Meanwhile, by implementing all models on the same supporting data management library from GraphBIG, we can better control other sources of variances, such as language, compiler, system library, and data layout.

The comparison results are shown in Fig. 8. We analyzed three selected workloads, Breadth-first Search (BFS), Single-source Shortest Path (SPath), and Triangle Counting (TC). This is because computation model variations can be better illustrated by graph traversal workloads with dynamic working-set size. BFS and SPath are both representative ones among such workloads. On the other hand, TC contains static working-set size and can illustrate the architectural behaviors when computation model brings limited impact. In the experiments, we characterized multiple architectural metrics, including instruction per cycle (IPC), execution cycle breakdown, cache hit rate, and execution time. For TC, we can observe that it is insensitive to computation model variations. Both the performance and architectural behaviors show similar results across all computation models. From the results of BFS and SPath, we can see that the GAS model is utilizing the hardware resources more efficiently. It is showing higher IPC values than both GraphBIG and Pregel, even though the IPC value is low in general. However, in the execution time analysis, we can observe an interestingly opposite result, in which the GAS consumes more execution time than GraphBIG. This is because the asynchronous programming model of GAS brings complex task scheduling and fine-grained lock management. Although the scheduling part can be executed more efficiently comparing with graph traversal operations, it increases overall execution time significantly. Thus, we can observe that GAS shows higher IPC, while longer execution time at the same time. From the execution cycle breakdown result, we can see that despite the differences in computation models, they all share the similar bottleneck in backend. Meanwhile,

Please cite this article in press as: L. Nai, et al., Exploring big graph computing — An empirical study from architectural perspective, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.07.006

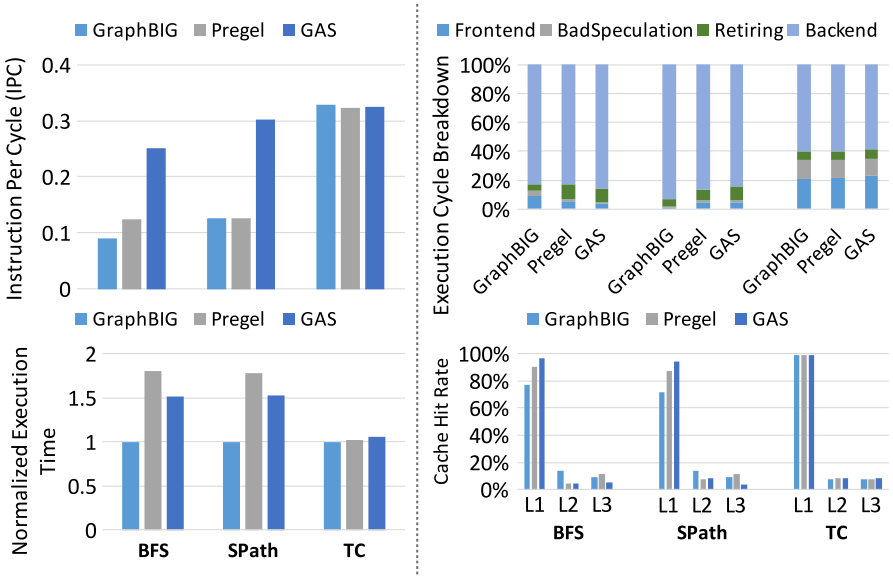



**Fig. 8.** Comparison between multiple graph frameworks.

as illustrated in the cache hit rate results, it can be inferred that the backend bottleneck is caused by the memory sub-system. Similar as previous GraphBIG analysis, the extremely low L2/L3 cache hit rate leads to inefficient memory sub-system processing, which then brings the high stall cycles in processor backend. Therefore, from the results, we can conclude that the inefficiency in memory sub-system remains one of the key bottlenecks for graph computing workloads, no matter for GraphBIG, Pregel, or GAS computation models. Such bottleneck is because of the inherent irregular graph connectivity, which brings poor data locality during graph traversals.

### 4.3. Big graph characterization

To further explore graph computing with big graphs, we perform characterization experiments on Bitcoin graph. Bitcoin a peer-to-peer payment system, in which users can transact directly without needing an intermediary. The Bitcoin graph is a network of Bitcoin accounts and transactions. In Bitcoin graph, vertices represent Bitcoin accounts, while edges represent Bitcoin transactions. It contains 72M vertices and 182M edges, representing 72M accounts and 182M transactions between them. Because of the nature of currency accounts, Bitcoin graph shows highly unbalanced degree distributions similar as power graphs.

Because of the huge size of Bitcoin graph and the long workload execution time, we select four representative workloads, which are BFS, SSSP, kCore, and TC. We exclude computations on dynamic graphs here because those workloads are mostly populating or modifying graphs, and therefore are less sensitive to graph data changes.

The performance results as well as execution time breakdown are shown in Fig. 9. As same as previous analysis, backend is still the dominant factor for BFS, SSSP, and kCore. Because of the inefficiencies in memory sub-systems, extremely low IPC values are observed. Meanwhile, TC shows quite different breakdown behavior and has relatively better performance. Differences with previous LDBC experiments also exist. First, BFS, SSSP, and kCore all show performance degradation comparing with previous LDBC graph experiments. This is because of the more irregular access pattern introduced by the huge data size of Bitcoin graph. However, TC's IPC value shows only negligible changes. It is in accordance with its computation pattern, which involves mostly graph property accesses. The memory sub-system features are shown in Fig. 10. Similarly, the results are also of consistent trends with previous characterization. All workloads have relatively high L1D cache hit rates because of their meta data accesses and extremely low L2/L3 cache hit rates because of the irregular graph structures. Because of the huge graph size, decrements in L2/L3 cache hit rates are observed. However, large graph size also brings less irregularities in task queue scheduling, leading to significant performance improvement in L1D caches. For Bitcoin graph, intuitively, its huge memory footprint size makes DTLB issue even more severe. However, our experiments show only similar DTLB penalty cycles. This is because the irregular access pattern of graph workloads already achieves a large number of DTLB misses. Therefore, even when data size increases, the DTLB miss penalty cycle does not increase as much as expected.

### 4.4. Observations

In the characterization experiments, by measuring several architectural factors, we observed multiple graph computing features. The key observations are summarized as follows.

- Backend is the major bottleneck for most graph computing workloads, except CompProp category. Such bottleneck is caused by cache performance issues. Hence, it is important to optimize data access pattern of graph computing in framework and architecture designs.
- The ICache miss rate of GraphBIG is as low as conventional applications, even though many big data applications are known to have high ICache miss rate. This is because of the flat code hierarchy of the GraphBIG framework. High ICache miss rate brings significant frontend stalls that cannot be hidden via out-of-order execution mechanisms in modern architectures. Thus, the design of frameworks should not only consider usability, but also the code complexity and the ICache performance.
- Graph computing is usually considered to be cache-unfriendly. L2 and L3 caches indeed show extremely low hit rates in GraphBIG. However, L1D cache hit rate is comparable as conventional applications. This is because of the locality of non-graph data, such as temporal local variables and task queues. The results demonstrate that despite the irregular pattern of graph data, it is infeasible to simply disable cache for the whole graph applications because of the significant amount of L1D cache hits.

Please cite this article in press as: L. Nai, et al., Exploring big graph computing — An empirical study from architectural perspective, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.07.006

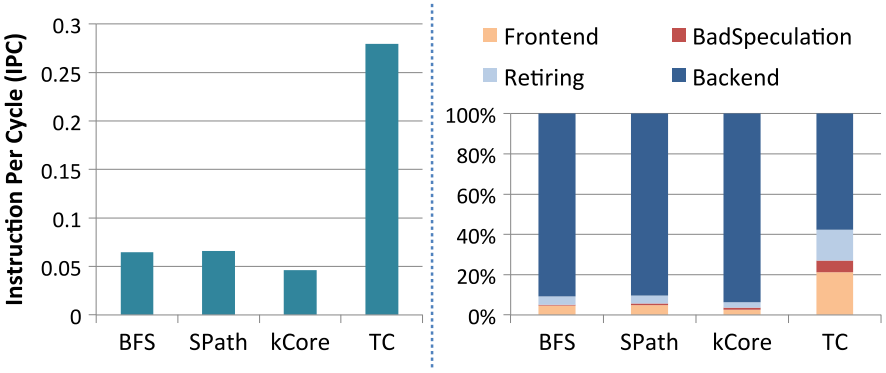



**Fig. 9.** Performance and execution time breakdown of Bitcoin graph.

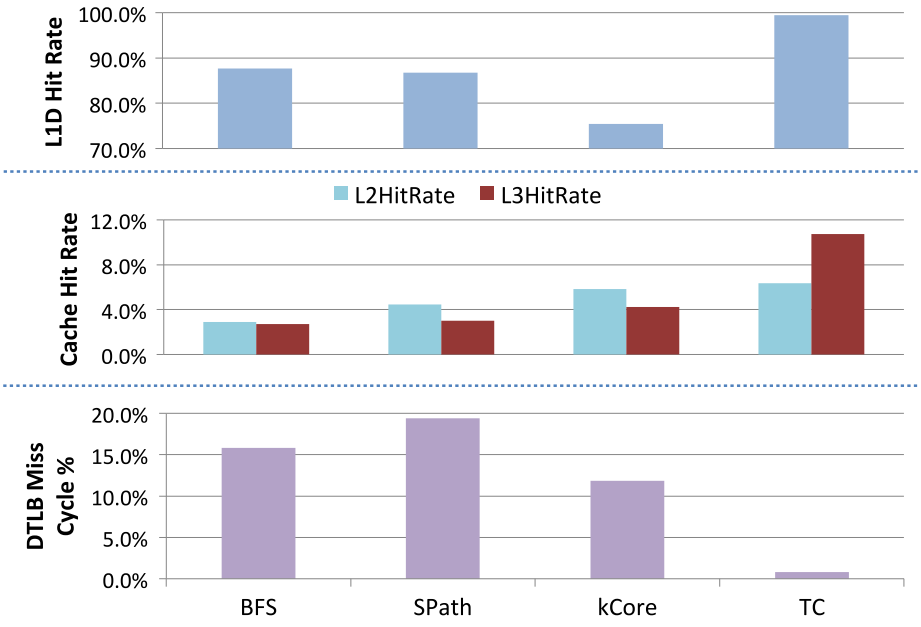



**Fig. 10.** Memory sub-system characteristics of Bitcoin graph.

- Although typically DTLB is not an issue for conventional applications, it is a significant source of inefficiencies for graph computing. In GraphBIG, a high DTLB miss penalty is observed because of the large memory footprint and low page locality. This result motivates the utilization of page optimization techniques, such as superpage.
- Graph workloads from different computation types show significant diversity in multiple architectural features. The study on graph computing should consider not only graph traversals, but also the other computation types.
- Input graph data has significant impact on memory sub-systems and the overall performance. The impact is from both the data volume and the graph topology.
- Computation models bring impact on multiple architectural characteristics and overall performance. Despite the differences, a general cache performance issue is shown in all computation models because of the irregular data pattern introduced by graph connectivity.
- The experiments with big graph show similar trends in architectural behavior as medium size graphs. It has lower overall performance and cache hit rates, but the similar DTLB patterns.

The unique and diverse features of graph workloads and data sets together form the broad scope of graph computing, leading to multiple complex challenges for future CPU architecture research.

### 4.5. Discussion

The optimization of graph computing performance can be achieved via multiple ways, such as algorithm innovations, faster system implementations, and better hardware platforms. Although algorithm usually is the most important factor for performance, its optimization usually is application-specific and therefore difficult to be generalized. For more generic approaches, efforts should be invested in software and hardware platform optimization. By examining a rich set of architectural features of graph computing applications, we have demonstrated a series of key observations, which can help both software and hardware optimization for graph computing.

First, significant inefficiencies exist in graph computing systems on contemporary hardware architectures. Thus, unlike highly optimized scientific computing applications, graph computing still contains promising opportunities for further architectural optimization. Second, since the key bottleneck of graph computing lies within memory sub-system, it is critical to optimize the cache performance first. The optimization can be achieved via software methods and hardware methods. For software implementation, the graph system needs a cache-conscious task scheduling and data management technique. Although the accesses of graph traversal are commonly considered as irregular, locality still may exist across multiple tasks or in non-graph data. From hardware

Please cite this article in press as: L. Nai, et al., Exploring big graph computing — An empirical study from architectural perspective, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.07.006

**Table 7**
Generated synthetic graphs.

| Feature | Configuration |
|---|---|
| Graph type | Kronecker graph |
| Graph vertex number | 64k, 128k, 256k, 512k, 1M, 2M, 4M, 8M |
| Graph density | 4, 8, 16, 32, 64, 128, 256, 512 |
| Graph connectivity | 120 random parameter matrices, Facebook-like (1, 0.589, 0.625, 0.368) |

perspective, a hardware platform that can tolerate a large number of cache misses is more desirable. For example, the cache miss tolerance can be contributed by the rich hardware thread resources, such as many-core/many-thread system and GPGPU. Third, because of the diverse behavior of different computation types, the evaluation of design choices should be correlated with the target applications. For a generic graph platform, it is important to consider all graph computation types in a holistic way. Moreover, the library and runtime code structure can significantly affect not only software design productivity, but also system performance. Graph systems should use a flat code hierarchy with limited external dependencies.

In this work, we performed experiments on a real machine by collecting hardware performance counters to characterize graph computing workloads. Although the discussion of general characterization techniques using performance counters is beyond the scope of this paper, our characterization methodology can also be applied on other graph applications. In our experiments, we follow a top-down analysis strategy, in which we first analyze the breakdown of execution cycles to understand the bottleneck components in the architecture. After determining that backend is the major issue, we further analyze the memory sub-system by collecting relevant architecture metrics, such as cache MPKI, cache hit rate, and DTLB penalty. The same workflow can also be utilized for the characterization and optimization of other graph applications.

## 5. Understanding data impact

### 5.1. Methodology

**Workloads**: We select four representative workloads from GraphBIG, including BFS, SPath, kCore, and TC. They cover two major computation types, computation on graph structure and graph properties. We exclude computation on dynamic graphs because it is highly related to graph structure updates and therefore is less feasible for data impact evaluations.

**Dataset generation**: To study the impact of input datasets on graph and computing, we performed experiments with generated synthetic graphs. In the experiments, we use Kronecker graph, which is a widely used graph generation technique [24]. We exclude the previous LDBC dataset or real-world datasets because our experiments require the flexibility of various graph size, density, and connectivity features. Kronecker graph can generate synthetic graphs with given vertex/edge number and connectivity structure according to the $2 \times 2$ parameter matrix. Its model recursively sub-divides the adjacency matrix of the graph into four equal-sized partitions and distributes edges within these partitions with the probabilities determined by the parameter matrix.

We generated Kronecker graphs with various graph vertex numbers, graph densities (a.k.a. average vertex degree), and graph connectivities via a Kronecker graph generator from SNAP library [25]. As shown in Table 7, the generated graphs cover 8 different sizes, 8 different densities, and 120 sets of connectivity parameters. To study the impact of graph data, we perform three groups of experiments. For graph density analysis, we select graphs with the same size and connectivity parameters to mimic the social graphs, but with different densities. Similarly for graph size study, we vary graph vertex numbers, while keep other factors the same. In both cases, the connectivity parameters were set to mimic the Facebook graph. For the analysis of graph connectivity, we generated 120 parameter matrices by performing uniform random sampling on the parameter hyper-plane.

**Architectural characteristics**: Similar as Section 4.1, we collected around 30 hardware performance counters to generate 11 architectural characteristics, which include IPC, execution time breakdown, L1D/L2/L3 cache hit rate, DTLB miss cycle, ICache MPKI, and branch miss prediction rate.

**Statistical data analysis**: For connectivity impact studies, our experiments involve 11 architectural characteristics, multiple workloads, and a large number of datasets. The architectural characteristics of different combinations of workloads and datasets form up a large number of high dimensional feature vectors. It is unfeasible to manually analyze the data, especially when studying connectivity impact. Therefore, we utilize multivariate statistical data analysis techniques, including Principal Component Analysis and Kmeans Clustering, to process the data.

Principal components analysis (PCA) [9] is a conventional data analysis technique that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables. We use PCA to remove the correlation between architectural characteristics and reduce the dimensionality of the experiment data.

After PCA processing, we perform Kmeans clustering on the generated reduced-dimension data to find the correlation between various graph configurations. The feature vectors will be grouped into multiple clusters. Each cluster represents graphs with similar architectural behaviors.

### 5.2. Impact of graph density

To analyze the impact of graph density, we perform experiments with different graph densities (a.k.a. average vertex degree). As summarized in Table 7, the density values vary from 4 to 512, while the vertex number remains to be 1 million. Meanwhile, each graph also has the same connectivity parameters, which are selected specifically to mimic the connectivity features of social graphs. As shown in Fig. 11, the graph workloads show diverse correlations between overall performance and graph densities. For example, BFS and SPath both show close to 2x performance degradation when graph density increases from 4 to 512. On the contrary, TC's performance increases with higher density. Its speedup with density-512 can be as high as 4.4. kCore generally has an increasing trend except for density-32, which shows a significant performance drop.

As explained in previous section, the major bottleneck of graph computing workloads comes from inefficiencies in memory sub-systems. To estimate the impact of memory sub-system, we analyzed the hit rate of each cache level and the percentage of DTLB miss time. The results are shown in Fig. 12. From the cache performance results, we can observe diverse behavior of different cache levels. As an example, with the increment of graph density, BFS and SPath both show decreasing hit rate in L1D and L2. However, their L3 hit rates are increasing instead. Similar

Please cite this article in press as: L. Nai, et al., Exploring big graph computing — An empirical study from architectural perspective, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.07.006

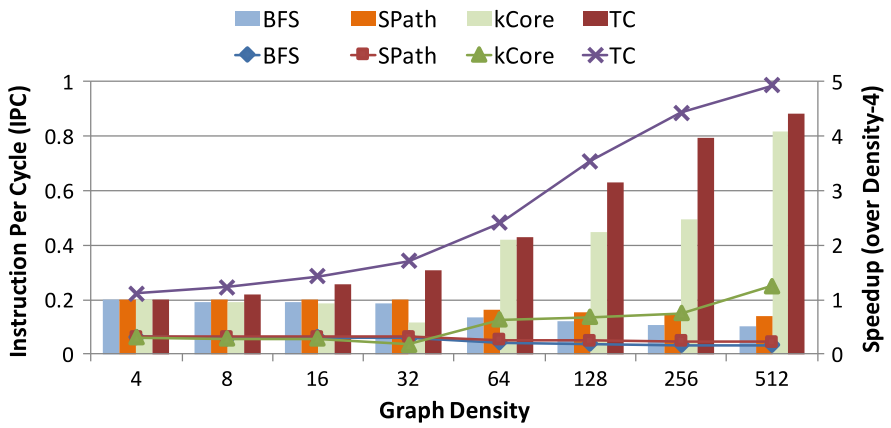



**Fig. 11.** Performance of different graph densities: Lines (left *y*-axis) represent the instruction per cycle (IPC); Bars (right *y*-axis) represent the speedup over the density-4 case.

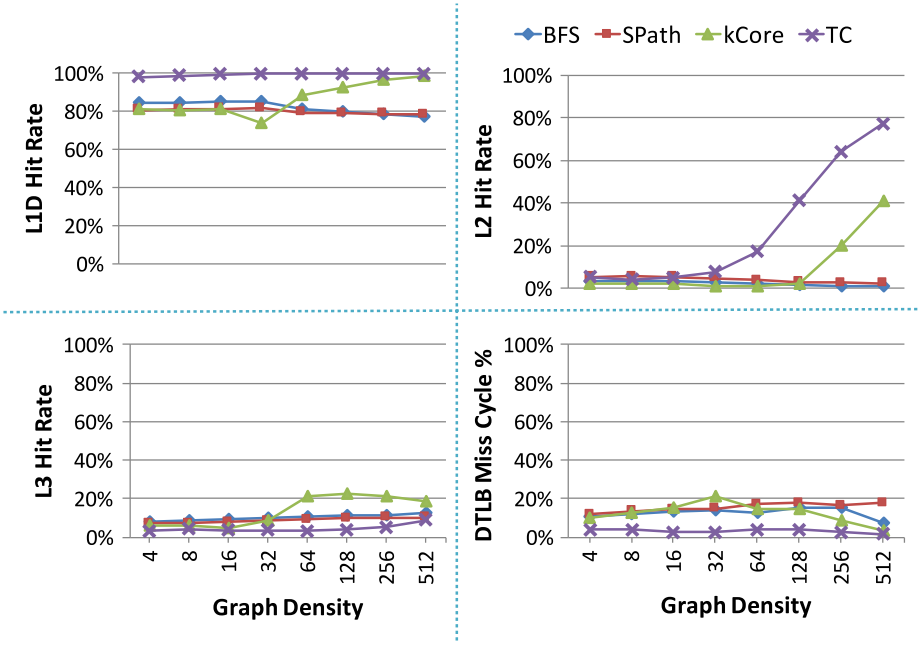



**Fig. 12.** Memory sub-system behavior of different graph densities, including L1D/L2/L3 hit rate and DTLB miss rate.

behavior also exists in kCore. In kCore, the cache hit rate outlier happens at density-32 in L1D, while in L3, the outlier point shows up at density-16. Meanwhile, DTLB miss time also shows diverse behavior. TC stays at a low penalty level with a small variance for all graph densities, while significant variances are observed in the other three workloads.

The diverse cache behavior is caused by each workload's memory access patterns. In graph workloads, because of the huge size of graph data and irregular graph structure, L1D cache hits are mostly generated by the accesses of meta data, such as task queues. Meanwhile, local graph properties may be cached by L2 cache. L3 cache likely holds graph structure data. In BFS and SPath, the task queues are formed up dynamically in each iteration. The irregular task allocation process may bring more L1D misses with larger vertex degrees. Thus, the L1D hit rate of BFS and SPath shows a decreasing trend. Meanwhile, denser graph usually leads to shorter re-reference distances of graph data. Hence, L3 cache shows an increasing trend for BFS and SPath. On the contrary, TC has static task queues and performs computations on the neighbor sets in vertex properties. Higher vertex degree brings more accesses within vertices. Such behavior exists in all cache levels, leading to significant hit rate increments. Unlike BFS and TC, kCore does not show monotonic trends in L1D/L3 cache or DTLB penalty time. This is because its computation tasks are highly sensitive to vertex degrees. Such sensitivity is reflected in meta data and graph data accesses, leading to high variances in L1D and L3 hit rates.

From the cache hit rate profiling results, we can also observe significant correlations between L1D cache hit rate and overall IPC value. This is because graph computing workloads generally have extremely low L2/L3 cache hit rate, leading to a high average L1D miss penalty time. The large performance penalty of L1D misses makes IPC value highly sensitive with L1D hit rates.

### 5.3. Impact of graph size

In general, it is a usual case that increment of data footprint hurts system overall performance. The experiments of graph workloads also support such intuition. The experiment results with different graph vertex numbers are shown in Fig. 13. From the results, we can see that although variances exist, BFS, SPath, and kCore indeed show performance degradation with larger graphs. However, TC's performance stays at the same level, showing an interestingly weak correlation with graph vertex numbers.

Please cite this article in press as: L. Nai, et al., Exploring big graph computing — An empirical study from architectural perspective, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.07.006

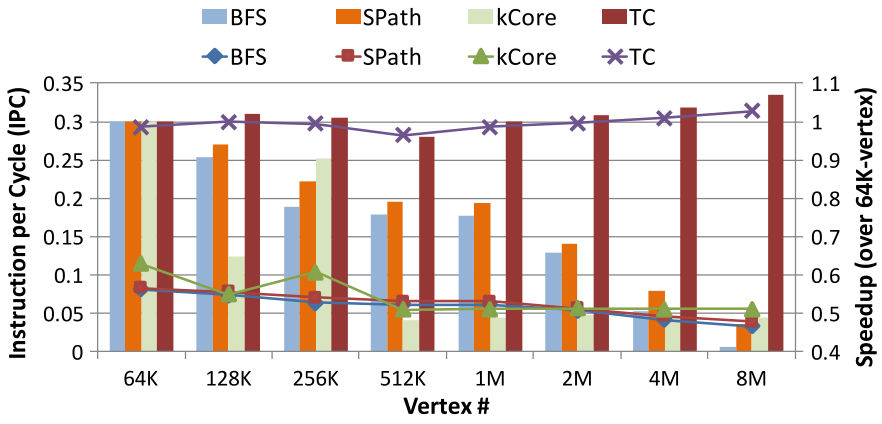



**Fig. 13.** Performance of different graph vertex numbers: Lines (left *y*-axis) represent the instruction per cycle (IPC); Bars (right *y*-axis) represent the speedup over the vertex-64k case.

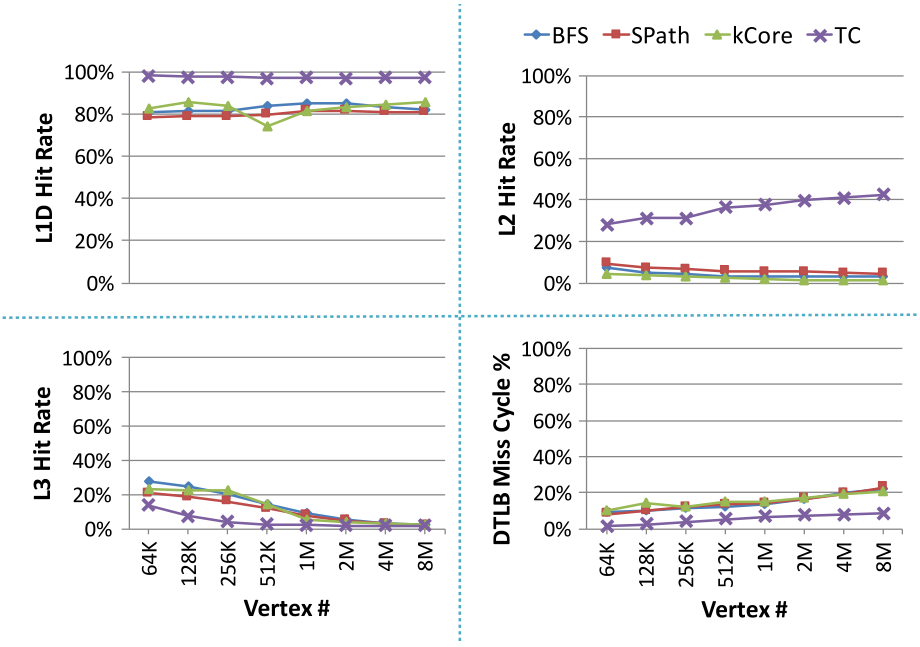



**Fig. 14.** Memory sub-system behavior of different graph vertex numbers.

Similar as previous graph density studies, we also perform analysis on memory sub-system behavior. As shown in Fig. 14, TC's L1D hit rate is stable at almost the same value. This is in accordance with its computation type, which has static work-set and involves mostly property accesses, not structure traversals. Other workloads all show significant variances in L1D cache hit rates because of their dynamic task queues, which are affected by graph sizes. Most workloads' L2 cache hit rate decreases slightly except for TC. In L3 cache, all workloads have significant decrements in hit rates. This is because of the change of graph sizes. With the increment of graph size, only a small portion of the full graph can fit into L3 cache. Thus, L3 cache accesses have much smaller chance to get re-referenced in a short period. Similarly, the DTLB accesses show more irregular pattern with the increment of graph size. The percentage of DTLB miss cycles increases from less than 10% to more than 20% for BFS, SPath, and kCore. For TC, the DTLB penalty varies between 1.4% and 8.2%.

### 5.4. Impact of graph connectivity

In this subsection, we analyze the impact of graph connectivity on architectural behavior of graph workloads. Unlike previous graph size/density studies, graph connectivity is a complex feature that cannot be represented as one simple parameter. Therefore, to analyze the impact of graph connectivity, we perform experiments on a large set of connectivity samples. With the architectural results of each sample, we then try to cluster them in the architectural feature space. The clustering outcomes indicate: (1) the correlation between connectivity feature and architectural behavior. (2) the representative connectivity samples that can be used for analyzing the architectural behavior of particular workloads on various graph connectivity.

In our experiments, we use 120 Kronecker graphs. As explained previously, the Kronecker graphs are generated by recursively sub-dividing the graph into four equal-sized partitions and distributing edges within each partition with the possibility given by the connectivity parameters. All of the generated graphs have the same size and density, but different connectivity parameters. The Kronecker graph parameters are generated by performing uniform random sampling on the hyper-plane of parameters. In each experiment, we collect 20 hardware counters and then generate 11 architectural features from them. To reduce feature dimension number and remove redundancies, we utilize PCA technique to process the results. The eigenvalue outcomes of each workload

Please cite this article in press as: L. Nai, et al., Exploring big graph computing — An empirical study from architectural perspective, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.07.006

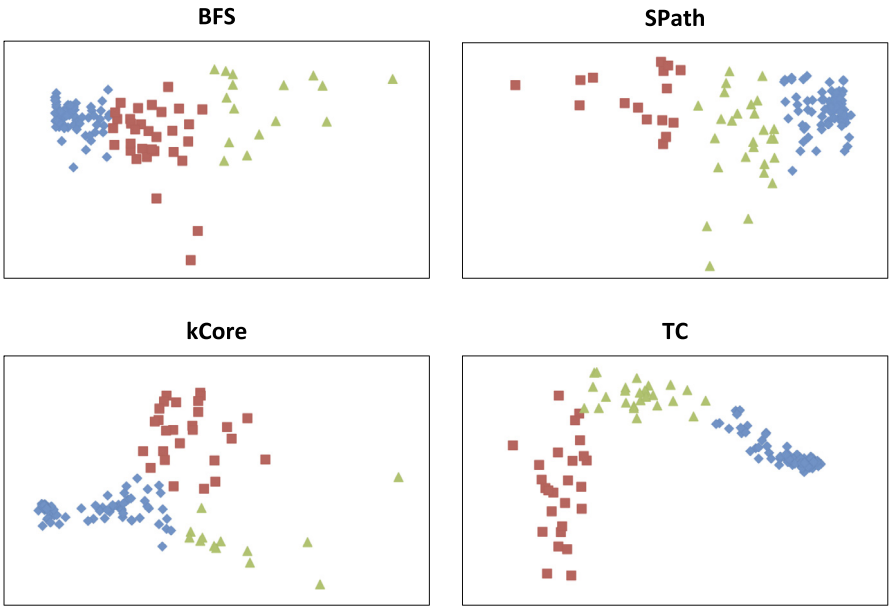



**Fig. 15.** Clustering of 120 graphs' experiment results according to the two-dimensional PCA outcomes of architectural features: Each dot represents the experiment of one graph; Three different dot shapes/colors represent three different clusters. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

all show a highly coalesced distribution, which demonstrates that two dimensional data vectors can already cover most necessary information. With the two-dimensional data after PCA process, we perform kmeans clustering on the data points. For illustration purposes, we use 3 clusters in the clustering processing. In further analysis, the cluster number can be changed according to dataset knowledge or determined by techniques like AIC and BIC.

Because the feature vector after PCA has only two dimensions, we are able to visualize them in 2D charts. The experiment data after PCA process and kmeans clustering is shown in Fig. 15. Different colors and dot shapes represent different kmeans clusters. From the results, we can see that BFS and SPath have similar non-uniform distributions. Both of them have much higher density in blue areas. kCore shows less biased distribution and TC has a unique correlation with specific feature dimensions. Our experiment results demonstrate that although we generate parameter matrices randomly with a uniform distribution, different parameter samples have different sensitivity for architectural features, leading to a non-uniform distribution in feature space. Moreover, the correlation is highly workload dependent. Therefore, to understand graph connectivity's impact for specific platforms, we should select representative connectivity parameter matrices according to its distribution in architectural feature space.

By utilizing the clustering results, we can select the representative graphs from each cluster according to their size and sparseness. For new workloads, the selection procedure can be done via the same profiling workflow. The steps include connectivity parameter sampling, architectural profiling, PCA dimension reduction, and kmeans clustering.

### 5.5. Observations

The key observations of our data impact analysis can be summarized as follows.

- Graph density, size, and connectivity all have significant impact on graph computing behaviors. The impact can be reflected in overall performance, cache hit rate, and DTLB miss penalty.
- The impact of graph data is workload dependent. Different workloads show diverse correlations between graph data factors and architectural features.
- L1D cache performance is more sensitive to graph density, while L3 cache is more sensitive to graph size.
- DTLB miss penalty increases with the increment of graph size. Graph density has much more complex impact on DTLB misses.
- Graphs with randomly sampled connectivity parameters show non-uniform distribution in architectural feature space. The selection of representative graphs can be achieved via the PCA and clustering process on experiment results.

### 5.6. Discussion

A variety of graph computing features shows significant dependence on the input graph. It is a common case that one specific optimization technique may become unsuitable when applied on a different graph. Therefore, in general, it is crucial to understand the potential target graph data before performing specific optimization techniques. Besides, because DTLB miss penalty increases with larger graphs, it is necessary to incorporate DTLB optimization methods, such as superpage, for big graph processing. Moreover, the impact of graph size and density is workload dependent. Thus, workload-specific optimization should be adopted for different input graphs. For example, BFS shows performance degradation with higher graph density. Techniques like vertex-cut [27] can be helpful. To optimize graph system implementation, we should perform tests on a set of representative input graph samples. The sample graphs can be selected from randomly sampled connectivity parameters based on kmeans clustering on the architectural feature space after PCA processing.

## 6. Conclusion

In this paper, we discussed and summarized the key performance factors of graph computing, including frameworks, data representations, graph computation types, and graph data sources. We analyzed real-world use cases to summarize the computation

Please cite this article in press as: L. Nai, et al., Exploring big graph computing — An empirical study from architectural perspective, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.07.006

types, and graph data sources. We also demonstrated the impact of framework and data representation.

To understand graph computing, we utilized our proposed benchmark suite, *GraphBIG*. GraphBIG addressed all key factors simultaneously by utilizing System G framework design and following a comprehensive workload selection procedure. With the summary of computation types and graph data sources, we selected representative workloads from key use cases to cover all computation types. In addition, we provided real-world datasets from different source types and synthetic social network data for characterization purposes.

By performing experiments on a real machine, we characterized GraphBIG workloads comprehensively. From the experiments, we observed the following behavior. (1) Significant inefficiencies in contemporary architectures are observed for graph computing. The major bottleneck is coming from memory sub-systems and is affected by multiple factors, such as data representation and workload algorithm. (2) Significant diverse behavior is shown across different workloads and different computation types. Such diversity involves multiple architectural features, from cache performance to branch prediction. (3) Graph computing is highly data sensitive. Graph size, density, and connectivity all have significant impact on multiple architecture features. The data-workload correlation is complex and diverse. Based on the architectural observations from our characterization, we also discussed and proposed suggestions for potential software and hardware optimization for graph computing.

As a comprehensive architectural study on graph computing, our work can help researchers achieve in-depth understanding of graph computing from the architectural perspective. It can also be served for future architecture and system research of graph computing.

## Acknowledgments

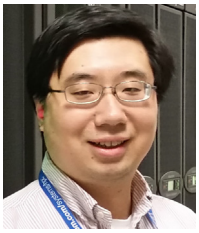
We gratefully acknowledge the support of National Science Foundation (NSF) CCF 1533767 and CCF 1337177. We would also like to thank the anonymous reviewers for their comments and suggestions. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF.

## References

[1] S. Andreassen, F.V. Jensen, S.K. Andersen, B. Falck, U. Kjærulff, M. Woldbye, A.R. Sørensen, A. Rosenfalck, F. Jensen, MUNIN — an expert EMG assistant, in: Computer-Aided Electromyography and Expert Systems, 1989.
[2] Apache, Apache Giraph, 2014. URL http://giraph.apache.org/.
[3] Y. Bengio, Learning deep architectures for ai, Found. Trends Mach. Learn. 2 (1) (2009).
[4] M. Burtscher, R. Nasre, K. Pingali, A quantitative study of irregular programs on gpus, in: IISWC'12.
[5] M. Canim, Y.-C. Chang, System G data store: Big, rich graph data analytics in the cloud, in: IC2E'13.
[6] A. Cano, S. Moral, Heuristic algorithms for the triangulation of graphs, 1995.
[7] M. Capotă, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, P. Boncz, Graphalytics: A big data benchmark for graph-processing platforms, in: Proceedings of the GRADES'15, GRADES'15, ACM, New York, NY, USA, 2015, pp. 7:1–7:6. http://dx.doi.org/10.1145/2764947.2764954, URL http://doi.acm.org/10.1145/2764947.2764954.
[8] E. Chesler, M. Haendel, Bioinformatics of Behavior:, no. pt. 2, Elsevier Science, 2012.
[9] G.H. Dunteman, Principal Components Analysis, Quantitative Applications in the Social Sciences, SAGE Publications, Inc., 1989.
[10] D. Ediger, K. Jiang, J. Riedy, D.A. Bader, Massive streaming data analytics: A case study with clustering coefficients, in: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, (IPDPSW), IEEE, 2010, pp. 1–8.
[11] S. Eranian, Libpfm4 Documentation, Perfmon2.
[12] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A.D. Popescu, A. Ailamaki, B. Falsafi, Clearing the clouds: A study of emerging scale-out workloads on modern hardware, in: ASPLOS XVII, ACM, New York, NY, USA, 2012.
[13] Z. Fu, M. Personick, B. Thompson, Mapgraph: A high level api for fast development of high performance graph analytics on gpus, in: GRADES'14, 2014.
[14] GraphBench, GraphBench, 2016. URL https://github.com/uwsampa/graphbench.
[15] Y. Guo, A.L. Varbanescu, A. Iosup, C. Martella, T.L. Willke, Benchmarking graph-processing platforms: A vision, in: ICPE'14, 2014.
[16] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, R. Zadeh, Wtf: The who to follow service at twitter, in: WWW'13, 2013.
[17] IBM, IBM System G, 2014. URL http://systemg.research.ibm.com.
[18] Intel, Intel 64 and IA-32 Architectures Optimization Reference Manual, Intel Corporation.
[19] U. Kang, S. Papadimitriou, J. Sun, H. Tong, Centralities in large networks: Algorithms and observations, in: SDM'11, 2011.
[20] U. Kang, C.E. Tsourakakis, C. Faloutsos, Pegasus: A peta-scale graph mining system implementation and observations, in: ICDM'09, Washington, DC, USA, 2009.
[21] F. Khorasani, K. Vora, R. Gupta, L.N. Bhuyan, Cusha: Vertex-centric graph processing on gpus, in: HPDC'14, 2014.
[22] A. Kyrola, G. Blelloch, C. Guestrin, Graphchi: Large-scale graph computation on just a pc, in: OSDI'12, Hollywood, CA, 2012.
[23] LDBC, LDBC Benchmarks, 2016. URL http://ldbcouncil.org/benchmarks.
[24] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, Z. Ghahramani, Kronecker graphs: An approach to modeling networks, J. Mach. Learn. Res. 11 (2010) 985–1042.
[25] J. Leskovec, A. Krevl, SNAP Datasets: Stanford large network dataset collection (Jun. 2014).
[26] C.-Y. Lin, L. Wu, Z. Wen, H. Tong, V. Griffiths-Fisher, L. Shi, D. Lubensky, Social network analysis in enterprise, Proc. IEEE 100 (9) (2012).
[27] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J.M. Hellerstein, Distributed graphlab: A framework for machine learning and data mining in the cloud, Proc. VLDB Endow. 5 (8) (2012).
[28] K. Madduri, D. Ediger, K. Jiang, D. Bader, D. Chavarria-Miranda, A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets, in: IPDPS'09, 2009.
[29] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: A system for large-scale graph processing, in: SIGMOD'10, 2010.
[30] D.W. Matula, L.L. Beck, Smallest-last ordering and clustering and graph coloring algorithms, J. ACM 30 (3) (1983).
[31] J. Mondal, A. Deshpande, Managing large dynamic graphs efficiently, in: SIGMOD'12, 2012.
[32] R.C. Murphy, K.B. Wheeler, B.W. Barrett, J.A. Ang, Introducing the graph 500, in: Cray User's Group, CUG, 2010.
[33] S.A. Myers, A. Sharma, P. Gupta, J. Lin, Information network or social network?: The structure of the twitter follow graph, in: WWW Companion'14, 2014.
[34] L. Nai, Y. Xia, C.-Y. Lin, B. Hong, H.-H.S. Lee, Cache-conscious graph collaborative filtering on multi-socket multicore systems, in: CF'14, 2014.
[35] L. Nai, Y. Xia, I.G. Tanase, H. Kim, C.-Y. Lin, Graphbig: Understanding graph computing in the context of industrial solutions, in: SC'15, 2015.
[36] Neo4j, Neo4j on POWER 8 Solution Data Sheet, Neo4j.
[37] M.J. Quinn, N. Deo, Parallel graph algorithms, ACM Comput. Surv. 16 (3) (1984).
[38] A. Roy, I. Mihailovic, W. Zwaenepoel, X-stream: Edge-centric graph processing using streaming partitions, in: SOSP'13, 2013.
[39] T. Schank, D. Wagner, Finding, counting and listing all triangles in large graphs, an experimental study, in: WEA'05, Springer-Verlag, Berlin, Heidelberg, 2005.
[40] B. Shao, H. Wang, Y. Li, Trinity: A distributed graph engine on a memory cloud, in: SIGMOD'13, 2013.
[41] J. Shun, G.E. Blelloch, J.T. Fineman, P.B. Gibbons, A. Kyrola, H.V. Simhadri, K. Tangwongsan, Brief announcement: The problem based benchmark suite, in: SPAA'12, 2012.
[42] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, V. Prasanna, Goffish: A sub-graph centric framework for large-scale graph analytics, in: Euro-Par 2014 Parallel Processing, Springer, 2014, pp. 451–462.
[43] I. Tanase, Y. Xia, L. Nai, Y. Liu, W. Tan, J. Crawford, C.-Y. Lin, A highly efficient runtime and graph library for large scale graph analytics, in: GRADES'14.
[44] A.L. Varbanescu, M. Verstraaten, C. de Laat, A. Penders, A. Iosup, H. Sips, Can portability improve performance?: An empirical study of parallel graph analytics, in: ICPE'15, 2015.
[45] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, B. Qiu, Bigdatabench: A big data benchmark suite from Internet services, in: HPCA'14.
[46] V. Weaver, perf_event Programming Guide, Linux man page project.
[47] Z. Wen, C.-Y. Lin, How accurately can one's interests be inferred from friends, in: WWW'10, ACM, New York, NY, USA, 2010.
[48] Y. Xia, System G: Graph analytics, storage and runtimes, in: Tutorial on the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2014.
[49] Y. Xia, J.-H. Lai, L. Nai, C.-Y. Lin, Concurrent image query using local random walk with restart on large scale graphs, in: ICMEW'14, 2014.
[50] Y. Xia, V.K. Prasanna, Topologically adaptive parallel breadth-first search on multicore processors, in: PDC'09, 2009.
[51] D. Ye, J. Ray, C. Harle, D. Kaeli, Performance characterization of spec cpu2006 integer benchmarks on x86-64 architecture, in: 2006 IEEE International Symposium on Workload Characterization, 2006, pp. 120–127.

Please cite this article in press as: L. Nai, et al., Exploring big graph computing — An empirical study from architectural perspective, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.07.006

**Lifeng Nai** received his M.S. and B.S. degrees in electrical engineering from Shanghai Jiao Tong University China, in 2008 and 2011 respectively, and is currently pursuing the Ph.D. degree in electrical and computer engineering from the Georgia Institute of Technology, Atlanta. His current research interests include architectural and system support for high performance large-scale graph computing, near-data processing, emerging memory technology, and heterogeneous computing architectures.



**Yinglong Xia** is currently a Senior Data Scientist in Huawei Research America, working on the next generation of big data analytics platforms. Before that, he was a Technical Lead of Graph Database and Reasoning Frameworks in the IBM T.J. Watson Research Center, establishing the native graph database and runtime for IBM System G. He received the Ph.D. in Computer Science from the University of Southern California (USC) in 2010, M.S. in Statistical Machine Learning from Tsinghua University in 2006, and B.S. from University of Electronic Sciences and Technology of China (UESTC) in 2003. Yinglong has multi-disciplinary background in Big Data, HPC, NoSQL Data Management, and Business Analytics. Yinglong is active in professional communities. He is a director since 2014 in the Linked Data Benchmark Council (LDBC). He is in the IEEE Big Data Standardization Committee and co-chair of the group on the big data management and metadata. He was a CCC/CRA Computing Innovation Fellow (CIFellow) in 2010–2012. He publishes extensively 40+ technical papers and 10+ patents. He is a general vice chair of HiPC'16, the industry program co-chair of IEEE Big Data'16, a program co-chair of CBDCom'16, and a TPC member of SC'16.

**Ilie G. Tanase** is a Research Staff Member at IBM T.J. Watson Research Center where he works on systems for large scale graph analytics and run time systems for parallel programming languages. He graduated with a Ph.D. in Computer Science from Texas A&M University. His Ph.D. work is on parallel data structures in the context of a C++ parallel programming library called STAPL. He received his Bachelor of Science from the Polytechnic University of Bucharest, Romania in 1999 and Master of Science from the same University in 2000. His research interests are in the area of graph databases, high performance computing, parallel programming languages and libraries, parallel algorithms and generic programming.

**Hyesoon Kim** is an associate professor in the School of Computer Science at the Georgia Institute of Technology. Her research interests include high-performance, energy-efficient heterogeneous architectures; interaction between programmers, compilers, and microarchitectures; and developing tools to help parallel programming. Kim has a Ph.D. in computer engineering from the University of Texas at Austin. She is a member of IEEE and the ACM.

Please cite this article in press as: L. Nai, et al., Exploring big graph computing — An empirical study from architectural perspective, J. Parallel Distrib. Comput. (2016), http://dx.doi.org/10.1016/j.jpdc.2016.07.006