J Supercomput (2015) 71:1563–1586
DOI 10.1007/s11227-015-1378-z

Efficient graph computation on hybrid CPU and GPU systems

Tao Zhang · Jingjie Zhang · Wei Shu · Min-You Wu · Xiaoyao Liang

Published online: 21 January 2015

Abstract Graphs are used to model many real objects such as social networks and web graphs. Many real applications in various fields require efficient and effective management of large-scale, graph-structured data. Although distributed graph engines such as GBase and Pregel handle billion-scale graphs, users need to be skilled at managing and tuning a distributed system in a cluster, which is a non-trivial job for ordinary users. Furthermore, these distributed systems need many machines in a cluster in order to provide reasonable performance. Several recent works proposed non-distributed graph processing platforms as complements to distributed platforms. In fact, efficient non-distributed platforms require less hardware resource and can achieve better energy efficiency than distributed ones. GraphChi is a representative non-distributed platform that is disk-based and can process billions of edges on CPUs in a single PC. However, the design drawbacks of GraphChi on I/O and computation model have limited its parallelism and performance. In this paper, we propose a general, disk-based graph engine called gGraph to process billion-scale graphs efficiently by utilizing both

T. Zhang (✉) · W. Shu · M.-Y. Wu · X. Liang
Department of Computer Science and Engineering, Shanghai Jiao Tong University, 800 Dong Chuan Road, Min Hang District, Shanghai 200240, China
e-mail: tao.zhang@sjtu.edu.cn

W. Shu
e-mail: shu@sjtu.edu.cn

M.-Y. Wu
e-mail: mwu@sjtu.edu.cn

X. Liang
e-mail: liang-xy@sjtu.edu.cn

J. Zhang
Department of Electrical Engineering, Fudan University, Shanghai, China
e-mail: 13307130068@fudan.edu.cn

CPUs and GPUs in a single PC. GGraph exploits full parallelism and full overlap of computation and I/O processing as much as possible. Experiment results show that gGraph outperforms GraphChi and PowerGraph. In addition, gGraph achieves the best energy efficiency among all evaluated platforms.

Keywords Graph · Graph algorithm · Graph processing platform · GPU

1 Introduction

In recent years we have witnessed an explosive growth of graph data. For example, the World Wide Web graph currently has over 4.34 billion pages and one trillion URLs [18]. Also, the social network of Facebook has over 1,310 million users and 140 billion social links [9]. The volume to store only the topology of such a graph is beyond TeraBytes (TB), letting alone rich metadata on vertices and edges. Graphs with billions of vertices resident in memory require hundreds of gigabytes of main memory, which is only possible in very expensive servers [14].

Processing large-scale real-world graphs has become significantly important for mining valuable information and learning knowledge in many areas, such as data analytics, web search and recommendation systems. The most frequently used algorithmic kernels, including path exploration (e.g., traversal, shortest paths computation) and topology-based iteration (e.g., page rank, clustering), are driven by graph structures. Many scalable systems have been recently proposed to handle big graphs efficiently by exploiting distributed computing. For example, GBase [20] is a recent graph engine using MapReduce. It shows that, if the graph is represented as a compressed matrix, matrix-vector computation solves many representative graph queries including global queries such as page rank and targeted queries such as induced subgraph and k-step neighbor queries. However, distributed systems based on MapReduce are generally slow unless there are sufficient machines in a cluster. For example, GBase used 100 machines to answer a two-step out-neighbor query from a given vertex in the Yahoo Web graph in about 265 s [20].

To solve the inherent performance problem of MapReduce, many distributed systems based on the vertex-centric model have been proposed, including Pregel [27], GraphLab [26], PowerGraph [14] and GPS [33]. In this model, users only need to write a function for each graph query type, which is invoked for each vertex by underlying systems. However, efficient graph partitioning in a distributed environment for all types of graph operations is very difficult [24]. Furthermore, users need to be skilled at managing and tuning a distributed system in a cluster, which is a non-trivial job for ordinary users. These distributed systems still need many machines in order to provide good performance.

Recently, several non-distributed graph processing platforms have been proposed as complements and more energy-efficient and economical alternatives to distributed graph processing platforms. Ligra [37] is a lightweight graph processing framework that is specific for shared-memory parallel/multicore machines, which makes graph traversal algorithms easy to write. It is designed to compute graphs purely in memory, which requires a large-sized memory in a single PC. GPUs are advantageous in good

energy efficiency, massive parallelism and high memory access bandwidth [12]. Totem [13] is a processing engine that provides a convenient environment to implement graph algorithms on hybrid CPU and GPU platforms. It integrates optimized graph partitioning strategies as well as other GPU-specific optimizations. Totem utilizes GPUs for additional computing power and improved energy efficiency. The author demonstrated that Totem outperformed state-of-the art CPU-based graph-processing systems in terms of both performance and energy efficiency [12]. However, a common drawback of Neo4j, Ligra and Totem is that they can not process large-scale graphs exceeding the PC's memory capacity.

GraphChi [24] is the first disk-based platform that can process large-scale graphs on CPUs in a single PC. GraphChi exploits the novel concept of parallel sliding windows (PSW) for handling billion-scale disk-based graphs. Like Pregel [27], GraphChi also follows the vertex-centric model. Since GraphChi is a disk-based system rather than a distributed system, message passing through edges is implemented as updating values to the edges. PSW divides the vertices into P execution intervals, and each execution interval contains a shard file which stores all edges that have target vertices in that interval. The edges in each shard file are ordered by their source vertices. PSW processes one shard file at a time. Processing each shard consists of three separate sub-steps: (1) loading a subgraph from a disk, and (2) updating the vertices and edges, and (3) writing the updated parts of the subgraph to the disk. GraphChi can use less hardware resource to provide comparable performance with representative distributed disk-based platforms [24].

We observe that GraphChi has several problems. First, it adopts the asynchronous computation model and enforces race condition detection for data updating, which limits the parallelism. Second, the PSW method loads both in-edges and out-edges and processes a shard at a time, which further decreases the parallelism and performance. Third, the disk I/O is not fully overlapped with the computation; the computation threads have to stall to wait for data, which degrades the performance. Fourth, it maintains an edge value vector and uses edges' value to communicate between vertices, which incurs large memory footprint. Overall, these issues result in a poor scalability in terms of the number of threads and a poor utilization in terms of total hardware resources.

We present a general, disk-based graph processing platform called gGraph which can process billion-scale graphs very efficiently by exploiting modern hardware in a single PC. GGraph exploits: (1) full parallelism including flash SSD I/O parallelism, multi-core CPU parallelism and many-core GPU parallelism and (2) full overlap of computation and I/O processing as much as possible—data are pre-fetched prior to the computation. Note that multi-core CPUs and many-core GPUs can process multiple jobs at the same time, and flash SSDs can process multiple I/O requests in parallel by using the underlying multiple flash memory packages. Our experiments show that gGraph outperforms GraphChi [24] and PowerGraph [14]. In addition, gGraph achieves the best energy efficiency among all evaluated platforms.

This work is an exploration on three important aspects of supercomputing: large-scale graph processing, performance, and energy efficiency. Although distributed computing will be the ultimate choice for processing very large-scale graphs in terms of performance and scalability, we believe that non-distributed platforms (e.g., SSD-based,

hybrid CPU and GPU graph processing platforms) are valuable complements, instead of replacements, to distributed graph processing platforms. These non-distributed platforms are eligible candidates for some energy-critical or economy-critical cases in supercomputing.

Our contributions are as follows: First, we propose a general, I/O efficient and scalable graph processing platform called gGraph which exploits the parallelism of multi-core CPUs, many-core GPUs and SSDs. GGraph is the first disk-based, hybrid CPU and GPU platform for large-scale graph processing on a single PC. Second, we propose an adaptive load balancing method and a load and store unit for data prefetching to improve performance. Third, we present a comparative evaluation of gGraph with three state-of-the-art graph processing platforms, and propose to use energy efficiency in addition to performance as metrics for a fair comparison.

2 Related work

Previous work has demonstrated the benefit of utilizing GPUs to accelerate graph processing. Harish et al. [17], Vineet et al. [39], and Merrill et al. [29] implemented several graph algorithms including Breadth First Search (BFS), parallel Boruvka and Minimum Spanning Tree(MST) on GPUs and gained various speedups. These past works assumed that the GPU memory can hold the entire graph. However, even the high-end GPUs today (e.g., Nvidia K20 series) have only up to 20 GB memory each, which is far from enough to hold a large-scale graph. For example, a snapshot of the current Twitter follower network has over 500 million vertices and 100 billion edges, and requires at least 0.5 TB of memory. Recent work by Gharaibeh et al. [13] utilizes both the host (CPU) memory and the device (GPU) memory to hold graphs and partitions workloads between CPUs and GPUs for processing in parallel. Still, the host memory and device memory of a single machine are inadequate to hold large-scale graphs. The communication and workload distribution strategies between compute nodes have been addressed by Arabnia et al. [2,3], and we will investigate this matter in the context of CPUs and GPUs.

There are numerous algorithms for various types of graph queries, e.g., finding neighborhoods [28], community detection [22], finding induced subgraphs [1], computing the number of triangles [19], finding connected components [36], computing subgraph isomorphism [25], and page rank [31]. Most of them are based on an in-memory computing model that limits their ability to handle large-scale, disk-resident graphs. Thus, they do not tend to scale well for web-scale graphs with billions of vertices and edges. To efficiently handle web-scale graphs and reduce the redundant effort of developing an algorithm for each query, many scalable and high-level graph systems [14,20,21,26,27,33,34,37] have recently been proposed. They support various kinds of graph queries instead of a specific graph query and also can handle web-scale graphs with billions of vertices and edges. They can be classified into distributed systems and non-distributed systems depending on the number of computing nodes. Distributed systems can be further categorized into synchronous systems and asynchronous systems.

Distributed synchronous systems: PEGASUS [21] and GBase [20] are based on MapReduce and support matrix-vector multiplication using compressed matrices. Pregel [27] is not based on MapReduce but on the vertex-centric model where a vertex kernel is executed in parallel on each vertex. In this model, the user only needs to write a function for each graph query type, which is invoked for each vertex by the underlying system. Pregel follows the Bulk-Synchronous Parallel (BSP) message passing model in which all vertex kernels run simultaneously in a sequence of super-steps. Within a super-step, each kernel receives all messages from the previous super-step and sends them to its neighbors in the next super-step. A barrier is imposed between super-steps to ensure that all kernels finish processing messages. All synchronous approaches above could suffer from costly performance penalties since the runtime of each step is determined by the slowest machine in the cluster. Such an imbalance in runtime may be caused by a lot of factors including hardware variability, network imbalances, and power-law degree distributions of natural graphs. GPS [33] is a complete open-source system developed for scalable, fault-tolerant, and easy-to-program execution of algorithms on extremely large graphs. GPS implements an extended API to make global computations more easily expressed and more efficient. It includes a dynamic repartitioning scheme that reassigns vertices to different workers during the computation based on messaging patterns. In addition, it has an optimization that distributes adjacency lists of high-degree vertices across all compute nodes to improve performance.

Distributed asynchronous systems: GraphLab [26] is also based on the vertex-centric model but a vertex kernel is executed in asynchronous parallel on each vertex. In GraphLab, each vertex reads and writes data on adjacent vertices and edges through shared-memory instead of messages. Since asynchronous systems update parameters using the most recent parameter values as input, they can make many kinds of queries converge faster than synchronous systems do. However, some algorithms based on asynchronous computation require serializability for correctness, and GraphLab allows a user to choose the level of consistency needed for correctness. PowerGraph [14] is basically similar to GraphLab, but it partitions and stores graphs by exploiting the properties of real-world graphs of highly skewed power-law degree distributions. Even though the above asynchronous approaches have the algorithmic benefits of converging faster, efficient graph partitioning in a distributed environment for all types of graph operations is an inherently hard problem [24]. Furthermore, the user must be skilled at managing and tuning a distributed system in a cluster, which is a non-trivial job for ordinary users.

Non-distributed systems: Neo4j [30] is one of the popular open-source graph databases. Neo4j stores data in graphs rather than in tables. Every stored graph in Neo4j consists of relationships and vertices annotated with properties. Neo4j can execute graph-processing algorithms efficiently on just a single machine, because of its optimization techniques that favor response time. Ligra [37] is a lightweight graph processing framework that is specific for shared-memory parallel/multicore machines, which makes graph traversal algorithms easy to write. Ligra requires large memory in a single PC to hold and compute the entire graph in memory, which is infeasible for large-scale graphs. GraphChi [24] is a disk-based single machine system following the asynchronous vertex-centric model. GraphChi exploits the novel approach called

Parallel Sliding Windows (PSW) for handling web-scale disk-based graphs. Since it is a disk-based system rather than a distributed system, message passing through edges is implemented as updating values to the edges. PSW divides the vertices into P execution intervals, and each execution interval contains a shard file which stores all edges that have the target vertices in that interval. The edges in each shard file are ordered by their source vertices. PSW processes one shard file at a time. When processing each shard, there are three separate sub-steps: (1) loading a subgraph from disk, (2) updating the vertices and edges, and (3) writing the updated parts of the subgraph to disk. Even though GraphChi is very efficient, and thus able to significantly outperform large Hadoop deployments on some graph problems while using only a single machine, there are still several serious problems which we discussed in Sect. 1. As a consequence, it results in poor scalability in terms of the number of threads and poor utilization in terms of total hardware resources.

3 Opportunities and challenges

Large-scale graph processing faces two major difficulties. First, large memory footprint: efficient graph processing requires the whole graph to be loaded in memory. However, large real-world graphs can occupy few gigabytes to terabytes of memory space. Second, high memory access latency combined with a random memory access pattern: during graph processing, the value of vertices and edges will be loaded and stored for one or more times. Because of poor locality and the scale of the workload, caches often miss and most accesses are served by the main memory. Therefore, industries and academia are seeking for solutions in various directions.

The opportunities: While distributed graph processing is the mainstream and can scale up and provide highest performance, non-distributed platforms have the potential as complements to achieve better energy efficiency. GPU-accelerated, disk-based graph processing has the potential to offer a viable solution. Compared to CPUs, GPUs have much higher parallelism and memory bandwidth. Today's commodity GPUs support thousands of hardware threads and in-flight memory requests through light-weight hardware multi-threading. Besides, the memory access latency on GPUs can be effectively hid by other active computing threads. In addition, properly partitioning and mapping the algorithmic computing tasks between the CPUs and GPUs holds the promise to utilize both computing resources best: CPUs are optimized for latency while GPUs are optimized for throughput. Therefore we could offload many light computing tasks onto GPUs while keeping a few computing intensive tasks on CPUs.

The challenges: Large-scale graph processing poses several major challenges to hybrid CPU and GPU systems. First, the scale of large-scale graphs exceeds the overall system memory resource (on the host and the device). This motivates us to design systems using hard disks as secondary storage for graphs, and efficient data loading mechanism to improve performance. Experiments show that the runtime of the graph algorithms on GraphChi is dominated by the I/O time [24]. Therefore we need to explore better designs to reduce the impact of I/O on performance. Second, to achieve good performance on hybrid CPU and GPU systems, we need to balance their workload and improve their utilization. Finally, mapping high-level abstractions and APIs to

facilitate application development to the low level engine while limiting the efficiency loss, is an additional challenge.

4 Overview of graph processing on hybrid CPU and GPU systems

In this section, we give a brief overview on graphs, graph algorithms, the computation model of graph processing, the programming model for hybrid CPU and GPU systems and the characteristics of SSD I/O.

4.1 Graph algorithms

We distinguish between two main graph algorithm classes: traversal algorithms and analytical iterative algorithms.

Traversal algorithms involve iterating through vertices of the graph in a graph dependent ordering. Vertices can be traversed one time or multiple times. This class includes search algorithms (such as breadth-first search, depth-first search), single source shortest paths, minimum spanning tree algorithm, connectivity algorithms and so on.

Analytically iterative algorithms involve iterating over the entire graph multiple times until a convergence condition is reached. This class of algorithms can be efficiently implemented using the Bulk Synchronous Parallel (BSP) model. Algorithms in this class include page rank, connected components, community detection, triangle counting and so on.

We focus on this two classes of graph algorithms. Parallel implementations of these algorithms typically require some form of graph partition, concurrency control, and thread-level optimizations for optimal performance.

4.2 Graph representation

GGraph processes graphs stored in the Compressed Sparse Row (CSR) format. The CSR format is aimed at efficient storage of sparse matrices. In our case we use it to store the sparse adjacency matrix of the graph (A_G). This format is widely used in many graph processing systems such as Totem [13] and the work of Pearce et al. [32]. The basic principles of our system could be applied to systems using other storage formats; however, we chose CSR for the sake of comparability. The CSR format consists of two components: the row index and the column index. The row index of A_G is a vector R_G of size $|V|$ with $R_G[i]$ being the index of the first non-zero element of row i in the column index. The column index of A_G is a vector C_G of size $|E|$ which is a row-wise listing of the column numbers of those elements in A_G, which are non-zero. Depending on the specific algorithm's needs, an additional $O(V)$ vector may be allocated to store the vertices' value (e.g., the rank in page rank algorithm).

In graphChi [24], it always loads an $O(E)$ sized edge value vector in memory and stores it on disk after updating. In addition, graphChi uses the edge value to communicate between vertices implicitly. This significantly enlarges memory footprint as well

as the I/O cost. However, many algorithms such as BFS, page rank and connected component do not reply on edge weight. Therefore, our system uses an edge value vector only if an algorithm needs edge value/weight for computing.

During the computation in our system, the $O(V)$ sized row index R_G will be fully loaded into the host memory while only a portion of the column index will be loaded into the host memory at a time.

4.3 Computation model

Our system adopts the vertex-centric model of computation, as introduced by Pregel [27]. A problem is encoded as a directed graph, $G = (V, E)$. We associate a value with each vertex $v \in V$ and each edge $e = (\text{source}, \text{destination}) \in E$. To perform computation of an algorithm, a programmer specifies an **update-function**. The update-function is executed for each of the vertices, iteratively, until a termination condition is satisfied [24].

There are two widely used model to execute the update function: the Bulk-Synchronous Parallel (BSP) [38] model and the asynchronous model. The BSP model is adopted by Totem [13], Giraph [4], GPS [33] and many more graph processing platforms. On the other hand, graphChi [24] and graphLab [26] utilize the asynchronous model.

The asynchronous model is designed for fast convergence of algorithms instead of massive parallelism. In fact, it enforces race condition detection for value updating and a sequential order to load graph partitions from disk, which significantly reduces the parallelism. Therefore, our system adopts the BSP model.

With the BSP model, parallel systems execute the update function in lock-step, and synchronize after each iteration. BSP model is simpler to implement and has good parallelism. Our system adopts the (BSP) computation model and divides processing into rounds (*super-steps* in the BSP terminology). Each super-step consists of three phases executed in order: computation, communication and synchronization. In the computation phase, each processor (in our case a CPU or a GPU) executes computations asynchronously based on values stored in their local memories. In the communication phase, the processors exchange messages that are necessary to update their statuses before the next computation phase. The synchronization phase guarantees the delivery of the messages. Specifically, a message sent at super-step i is guaranteed to be available in the local memory of the destination processor at super-step $i + 1$.

Adopting the BSP model allows to circumvent the fact that the GPUs are connected via the high-latency PCI-Express bus. In particular, batch communication matches well BSP, and this enables our system to hide (some of) the bus latency. In addition, using the BSP model and a double buffer mechanism guarantees that the output of algorithms is deterministic regardless of the order of processing graph partitions.

4.4 Programming model

In gGraph, OpenMP and CUDA programming models are adopted to create and manage the CPU threads and GPU threads, respectively. We chose them primarily for better performance. Alternatively, OpenCL can be used to implement both the CPU kernels

and GPU kernels. OpenCL has been proposed to tackle multi-/many-core diversity in a unified way. However, it is not fully mature. Work [35] reported the performance degradation incurred by immature OpenCL compilers. In addition, there seems to be a trade-off between its unified way and performance. Previous work [10] showed that the CUDA version of GPU kernels outperformed the OpenCL version. Other work [11,35] reported that the OpenMP version of CPU kernels achieved better performance than the OpenCL version. After manually optimizing codes, CPU kernels in OpenCL can reach similar performance with those in OpenMP [35]. In summary, OpenCL can be good alternatives to CUDA and OpenMP in the future.

There are several differences between OpenMP and OpenCL CPU kernels. First, OpenCL is advantageous for its good portability; there are only slightly differences between kernels implemented in OpenCL for different processors (e.g., CPUs and GPUs). Second, OpenMP provides locality-friendly coarse-grained parallelism while OpenCL provides fine-grained parallelism [35]. In OpenMP, each thread processes consecutive data elements. In OpenCL, one work-item processes one data element. Improper implementation of the fine-grained parallelism method in OpenCL can lead to poor CPU cache utilization. Third, depending on the version of OpenCL and OpenMP, they may handle the divergent data-dependent branches differently: OpenMP using the hardware branch prediction to choose a path to execute and OpenCL executing all branch paths.

In gGraph, the GPU threads are allocated based on the amount of data, as the applications in CUDA SDK do. There is no bank conflict problem since the GPU kernels use no shared memory. The allocation of GPU threads has negligible impact on memory access performance since the memory access patterns of many graph algorithms on GPUs are known to be irregular and unpredictable [8].

4.5 SSD I/O characteristics

Unlike rotational Hard Disk Drive (HDD), Solid State Drive (SSD) can serve multiple random access requests in parallel without suffering degradation in the latency of servicing an individual request. Therefore they are adopted as a better alternative to HDD in graph processing platforms [24]. The number of requests that can be handled concurrently is referred to as the queue depth in literature. In our system, we design an efficient load and store manager that can access multiple trunk of the graph data concurrently through multiple threads, and a mechanism to identify future vertices and retrieve the associated data from the SSD ahead of time.

5 Architecture of gGraph platform

Figure 1 shows the architecture of gGraph and the underlying hardware. GGraph consists of three layers. The top layer is the main controller of the platform that initializes and terminates the algorithms, and performs computation using the supporting functional modules. There are four functional modules in the middle layer: load and store unit, graph partitioner, message handler and memory manager. At the bottom layer

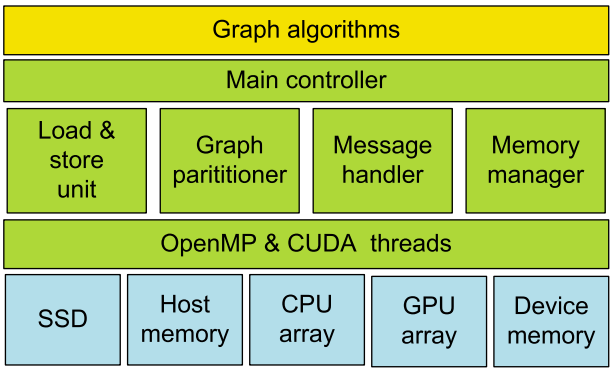

Fig. 1 Architecture diagram of gGraph platform

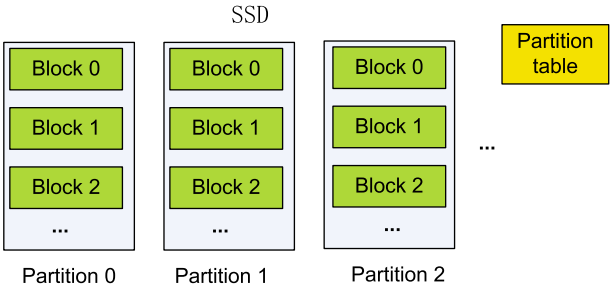

Fig. 2 Graph storage on the SSD

are the concurrent threads created by OpenMP and CUDA. Next we will elaborate the design and implementation details of the platform.

5.1 Graph storage

In this work, we consider large-scale graphs that can not fit into the host memory. In real-world graphs, the number of edges is often much larger than the number of vertices [13]. Therefore, we store the row index of the entire graph in a separate file and the column index of the entire graphs in P partitions to improve the bandwidth and latency to access the graph data on the SSD, as shown in Fig. 2. The P is determined such that each partition can fit into the host memory. Both the partition size and the block size are configurable in the platform configuration file. There is a partition table file that stores the mapping from each vertex to a ⟨partition_id, block_id⟩ pair.

5.2 Load and store unit

The Load and store unit (LSU) is in charge of the accesses to the SSD, whose architecture is shown in Fig. 3. Prior to the computation, LSU threads prefetch the vertices within the current loading window and their edges in CSR format from the SSD into

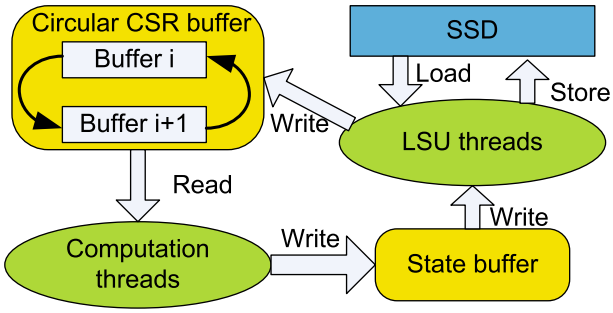

Fig. 3 Load and store unit

memory. We call the vertices and their edges as a *subgraph* since it is only a portion of the entire graph that fits into the host memory.

The current *loading window* is a set of vertices $\{v_i\}$ determined by the current progress and the data access pattern of the specific algorithm. There are two major type of access patterns: the random access pattern and the sequential access pattern.

- *Random access pattern:* traversal algorithms generally follow the random access pattern. The vertices to access (called frontier in literature [29]) in the next super-step of the BSP model are determined in the current super-step and maintained in a First In First Out (FIFO) queue in gGraph.
- *Sequential access pattern:* analytically iterative algorithms are generally in this pattern. The vertices to access in the next super-step are just the N vertices following the current loading window.

During the computing process, the loading window will be continuously updated. The updates will trigger a callback function which wakes up the LSU threads to make the prefetching. Therefore, the computation of super-step i is overlapped with the subgraph loading for super-step $i + 1$. The prefetched subgraph is put into a circular CSR buffer in the host memory. The circular CSR buffer has at least two buffers, one in use and another on hold. The buffer in use contains the subgraph for the computation of the current super-step, while the buffer on hold is used to store the subgraph for the next super-step.

In this paper, we do not consider dynamic graphs that evolve over time. In other words, the structures of graphs remain constant. Therefore, LSU loads but never stores the graph structure data. If there are interim results (e.g., the rank in page rank algorithm) or vertices' value to be saved onto the SSD, they are put into a state buffer by the computing threads and then stored onto the SSD by the LSU.

5.3 Graph partitioner

The graph partitioner distributes the computation workload among CPUs and GPUs and makes adjustment if necessary. Due to limited memory, gGraph only loads and computes a subgraph in memory at a time. For a system with M CPUs and N discrete GPUs, a subgraph will be cut into $N + 1$ sections, one section for all CPUs and one

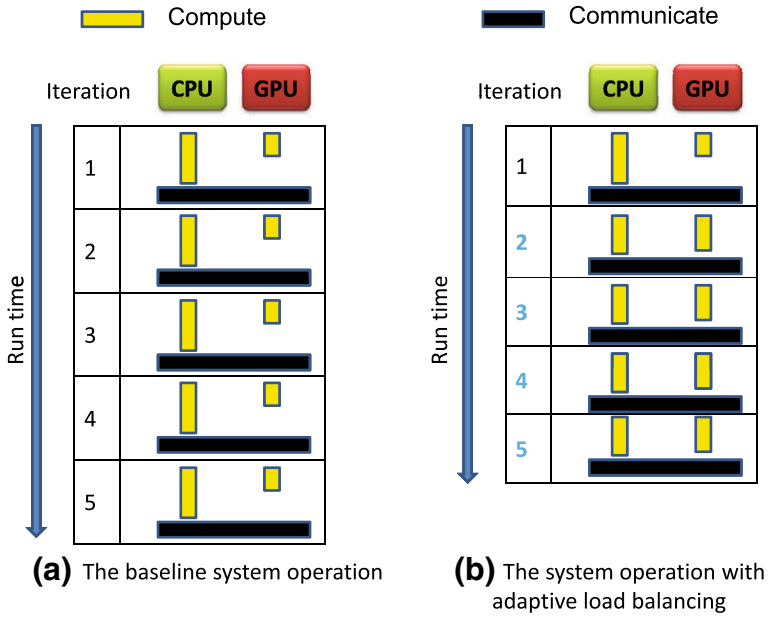

Fig. 4 An illustration of the adaptive load balancing method

section for each of the GPUs, respectively. The reason is that all CPUs share the same host memory and are homogeneous threads from the OpenMP's view, while each discrete GPU has its own device memory. Note that the processing power and memory capacity of GPUs can be different.

The graph partitioner cuts the subgraph in an effort such that the sections are proportional to the processing power of CPUs and GPUs and can fit into corresponding CPUs' or GPUs' memory. As shown in Fig. 4a, since the computation time of each super-step is determined by the slowest processor, load imbalance will prolong overall algorithm runtime. A static partition scheme can not guarantee optimal performance. We propose an analytical model-based method to adaptively balance the load among CPUs and GPUs at runtime.

we assume a heterogeneous platform P that consists of two processing elements $P = \{p_{\text{cpu}}, p_{\text{acc}}\}$, the CPU and an accelerator (e.g., a GPU). The model can be easily generalized to a mix of multiple CPUs and accelerators.

Let $G = (V, E)$ be a sub-graph, where V is the set of vertices and E is the set of edges. $|V|$ and $|E|$ represent the number of vertices and edges, respectively. The time it takes to process a section of G, $G_p = (V_p, E_p) \subseteq G$ on a processing element p, is given by:

$$t(G_p) = \frac{|E_p|}{R_p} \tag{1}$$

where R_p is the processing rate of processor p in Edges per second (E/s). Here we ignore the communication time since it is relatively small with the message aggregation mechanism introduced in Sect. 5.4.

We define the completion time of a subgraph G on a given platform P as follows:

$$T(G) = \max\{t(G_p)\}, \quad p \in P \tag{2}$$

For a system with a CPU and a GPU, the $T(G)$ can be expressed as:

$$T(G) = \max\{t(G_{\text{cpu}}), t(G_{\text{gpu}})\} \tag{3}$$

$$T(G) = \max\left\{\frac{|E_{\text{cpu}}|}{R_{\text{cpu}}}, \frac{|E_{\text{gpu}}|}{R_{\text{gpu}}}\right\} \tag{4}$$

We define a *CPU loading ratio* $\alpha = \frac{|E_{\text{cpu}}|}{|E|}$, so $|E_{\text{gpu}}| = (1-\alpha)|E|$. So we get:

$$T(G) = \max\left\{\frac{\alpha|E|}{R_{\text{cpu}}}, \frac{(1-\alpha)|E|}{R_{\text{gpu}}}\right\} \tag{5}$$

The parameters in Eq. (5) are unknown prior to the first execution. Depending on the specific hardware, R_{cpu} could be larger or smaller than R_{gpu}. α determines where to cut a sub-graph and consequently determines $|E_{\text{cpu}}|$ and $|E_{\text{gpu}}|$. So it is paramount to find an optimal α that minimizes $T(G)$.

Our system adopts the following method to find an optimal value for α:

- Before the first iteration, initializes α, R_{cpu} and R_{gpu} to 0.
- At the beginning of each iteration, if there are valid R_{cpu} and R_{gpu} ($R_{\text{cpu}} \neq 0$ and $R_{\text{gpu}} \neq 0$), find the optimal α that minimizes $T(G)$ in Eq. (5) which does not violate the capacity of the host and the device memory. Otherwise adopts the minimum α that does not violate the capacity of the host and the device memory.
- At the end of each iteration, calculates and updates R_{cpu} and R_{gpu}.

In practice, it may be better to derive the optimal α after the second iteration since some algorithms use the first iteration to assign initial value to vertices and edges. The real computation starts from the second iteration.

5.4 Message handler

The message handler maintains the message buffers and performs message aggregation. Since gGraph partitions a graph into subgraphs then sections, there are some boundary edges whose one end-vertex (called local vertex) is within a section while the other end-vertex (called remote neighbor) is not. So messages are sent via these boundary edges to remote neighbors to notify the value changes of vertices. The messages are temporarily buffered in the computation phase and then transferred in the communication phase.

The message handler maintains two buffers for each processing unit: an outbox buffer and an inbox buffer. For the same reason discussed in Sect. 5.3, each GPU has its own inbox and outbox while all CPUs share an inbox and an outbox. The outbox buffers have an entry for each remote neighbor, while the inbox buffers have an entry

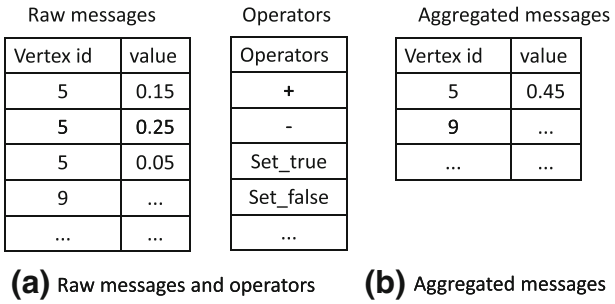

Fig. 5 An example of message aggregation

for each local vertex that is remote to another partition. An inbox or outbox buffer is composed of two arrays: one contains the vertex ID and the other stores the messages.

We enforce message aggregation to reduce the communication cost. In each processing unit, messages to a same destination vertex (from different source vertices) are aggregated into one message. Figure 5 presents an example for message aggregation. Figure 5a shows some raw messages as well as the operators. The choosing of the operator depends on the specific algorithm. For example, page rank takes the weighted sum of neighbouring vertices as the new value of a vertex, so the operator for this algorithm is plus. Figure 5b shows the aggregated messages with the plus operator. This optimization is based on the observation that real-world graphs exhibit power-law degree distribution [6], so vertices have lots of in-edges and out-edges.

5.5 Memory manager

Like GraphChi and PowerGraph, gGraph uses OpenMP only for threads management. The memory used in gGraph is managed by the memory manager module. The memory manager allocates, deallocates and recycles the memory used by the platform. It ensures that the memory allocated by the consumers in gGraph does not exceed a pre-configured limit. The graph data are stored in the host DDR memory and the GPU DDR memory, and accessed by the CPU/OpenMP threads and GPU/CUDA threads, respectively. Comparing with the host DDR memory, the GPU DDR memory is small in capacity but with much bigger bandwidth shared by many threads. There are several main memory consumers:

- *Circular CSR buffer:* an $O(E)$ buffer to store the CSR of subgraphs.
- *State buffer:* an $O(V)$ and sometimes $O(E)$ buffer that stores the interim algorithmic output data (e.g., rank in page rank algorithm or vertices' value). If algorithms need edges' weight for computing, the state buffer takes $O(E)$.
- *Loading window:* an $O(V)$ memory space to track the set of vertices (and their edges) to be loaded for the next super-step.
- *Miscellaneous buffer:* other small or temporary buffers such as the partition mapping table, the temporary buffer for I/O and so on.

Besides, the memory manager also handles the memory transfer between the host memory and the device memory. GGraph utilizes asynchronous memory copy to overlap the computation and memory transfer and applies some techniques to improve the performance of memory accesses on GPUs, including using the pinned memory, configuring the L1 cache to be 48 KB (maximum size allowed) in the 64 KB configurable memory and implementing GPU kernels to create coalesced memory accesses as many as possible.

6 Evaluation methodology

6.1 Graph algorithms

We select three graph algorithms for evaluation: connected component (CONN), single source shortest path (SSSP) and page rank (PR). This suite of algorithms has different per-vertex computation and communication intensity (based on the categorization in [33]) which can better evaluate the performance of the graph processing platforms. In addition, these algorithms have rich applications in social networks, recommendation systems, routing algorithms and other physical or biological simulations. Therefore, these algorithms are frequently adopted in performance evaluations. The characteristics of the algorithms are summarized in Table 1 and described as follows.

Connected component (CONN) is used to detect regions in graphs. A region or a connected component is a subset of vertices that can reach each other through edges. A vertex only belongs to a single connected component of the graph. This algorithm is crucial in many operations such as graph partition. Our implementation is based on the HCC algorithm implemented in PEGASUS [21]. Connected component has also been used to demonstrate GPS [33], CGMgraph [7], PBGL [16] and GraphChi [24].

Single source shortest path (SSSP) finds the shortest path from a single source vertex to all connected vertices, producing a shortest path tree. It is a well-known and long-studied problem with many practical applications. Our implementation is based on the SSSP algorithm implemented in Pregel [27]. Single source shortest path has also been used to demonstrate GPS [33] and PBGL [16].

Page rank (PR) is an algorithm used by Google to calculate probability distribution representing the likelihood that a person randomly clicking on links arrives at any particular page [31]. This algorithm works by counting the number and weight of edges to each vertex iteratively to determine a rough estimate of how important each vertex is. The underlying assumption is that more important vertices are likely to receive more weighted in-edges from other vertices. Given the large number of web pages on the World Wide Web, efficient computation of page rank becomes a challenging

Table 1 Graph algorithms

Abbr.	Algorithm	Computation	Communication
CONN	Connected component	Low	Medium
SSSP	Single source shortest path	Medium	Medium
PR	Page rank	Low	High

problem. In the evaluation, we run page rank for six iterations on each graph. Our implementation is based on the page rank algorithm implemented in Pregel [27]. Page rank has also been used to demonstrate GPS [33], Trinity [34], PEGASUS [21] and GraphChi [24].

6.2 Software and experiment configuration

We use three platforms GraphChi (version 0.2.1) [33], PowerGraph (version 2.2) [14] and GPS (version 1.0, rev.112) [33] to compare with gGraph. GraphChi and gGraph are non-distributed platforms running on a single-PC while PowerGraph and GPS are distributed platforms running on a cluster of PCs. The platforms evaluated have many configuration parameters that can potentially change their performance. We use common best-practices for tuning each of the platforms and configure all platforms to utilize all cores in multi-core CPUs. We run each algorithm with default parameter values on all platforms. Page rank will be executed for six iterations while connected components and single source shortest path will run to their completion. For single source shortest path, we use a pre-determined vertex (e.g., vertex 0) as the source vertex in each graph. We repeat each experiment ten times and report the average results.

6.3 Hardware platform

We use a single server computer for evaluating the performance of gGraph and graphChi [24]. The server is equipped with two Intel Xeon 2650 CPUs, each having eight cores at 2.0 GHz and a 20 MB cache. It has 64 GB DDR memory and a single 512 GB Samsung 830 SSD with around 250 MB/s read or write throughput. In addition, there are two Nvidia Tesla C2075 GPUs installed in the server. Each GPU has 14 Streaming Multiprocessors (SMs) clocked at 1.14 GHz with 2MB shared L2 cache, 6 GB DDR memory at 144 GB/s bandwidth. On one hand, GPUs have significantly larger number of hardware threads, higher memory access bandwidth, and support a larger number of in-flight memory requests. On the other hand, the CPU cores are clocked at around double the frequency, and have access to several times larger memory and cache.

Besides, we use a cluster of six workstation computers to evaluate powerGraph [14] and GPS [33] for comparison. Each workstation has two sockets, each containing an Intel Xeon E5-4603 CPU with quad-core at 2.0 GHz and 10 MB cache. There are 24 GB DDR memory and a 1 TB SATA hard disk in each computer. The workstation computers are connected by a 1 Gbit/s Ethernet network. Both the server and workstation computers are running Linux platform with version 3.1 kernel.

6.4 Workloads

We use both publicly available real-world graphs and synthetic Recursive MATrix (RMAT) [6] graphs to evaluate our system. The RMAT graphs are generated with the

Table 2 Summary of the workloads

Abbr.	Graph	|Vertices|	|Edges|	Direction	Type
G1	Twitter 2010 [23]	61.6 M	1.5 B	Undirected	Social
G2	Com-Friendster [40]	65.6 M	1.8 B	Undirected	Social
G3	Uk-2007-d [5]	106.0 M	3.7 B	Directed	Web
G4	RMAT29 [15]	512.0 M	8.0 B	Undirected	Synthetic
G5	RMAT30 [15]	1.0 B	16.0 B	Undirected	Synthetic

M million, *B* billion

parameters $(A, B, C) = (0.57, 0.19, 0.19)$ and an average degree of 16. The graphs are listed in Table 2.

6.5 Evaluation metrics

We evaluate the platforms with two metrics: performance and energy efficiency.

- *Performance:* the traversed edges per second (TEPS) is used as the performance metric. TEPS is proposed by the graph500 benchmark [15] to report the throughput of super-computers on graph processing.
- *Energy efficiency:* the efficiency in terms of energy expressed as traversed edges per joule. Since platforms use various hardware resource and consume different power, we divide their performance (in TEPS) by their power to get the energy efficiency for a fair comparison. Energy efficiency is one of the hottest research topics for energy-critical cases such as datacenter operations. There are bunches of architectural or software work focusing on improving system energy efficiency.

7 Results and analysis

In this section, we evaluate the performance of gGraph and compare it with GraphChi, PowerGraph and GPS. GraphChi and gGraph run on the single server computer but only gGraph utilizes the GPUs on it. PowerGraph and GPS run on a six-PC cluster. We first evaluate the effectiveness of the adaptive load balancing method of the graph partitioner. Then we investigate the scalability of gGraph by varying the hardware configurations and measuring the corresponding performance. To observe the performance of platforms under various conditions, we run three algorithms in all platforms on different graphs and observe the performance. Finally, since the four platforms utilize different hardware resource, we divide their performance by their power to get the energy efficiency for a fair comparison. The results shown in figures are the average results of ten runs. Performance across multiple runs varies by only a very small margin.

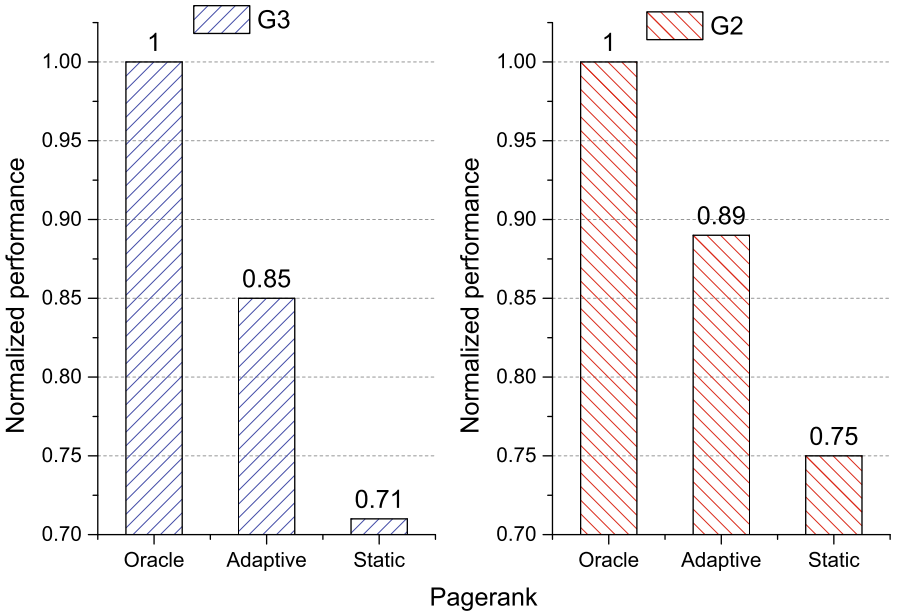

Fig. 6 Evaluation of the adaptive load balancing method

7.1 Evaluation of the adaptive load balancing method

Figure 6 shows the performance of the adaptive load balancing method. There are three bars for each graph. The "oracle" bar represents the performance under the optimal α, which can be found by exhaustive searches. The "adaptive" bar is the normalized performance with the adaptive load balancing method. Comparing with the "oracle", the "adaptive" has around 13 % performance degradation because it spends one or two un-optimized iterations to find out an optimal α to balance the load between CPUs and GPUs. The "static" bar represents the condition that the system uses the default static α in the entire application lifetime. The result shows that the performance with a static α is 14 % on average worse than the performance with the adaptive load balancing method. In summary, the adaptive load balancing method can provide better performance than using a static α.

7.2 Evaluation of the scalability

Figure 7 presents the performance of running page rank in gGraph with different hardware configurations including: one CPU, one GPU, two CPUs, one CPU and one GPU, two GPUs, two CPUs and one GPU, and two CPUs and two GPUs. The performance is expressed as million traversed edges per second (MTEPS). In the configurations of one GPU or two GPUs, each graph is processed by the GPU(s) in multiple partitions since the capacity of the GPU device memory is inadequate to hold the entire graph. On all three graphs, the performance increases proportionally with the growing of the

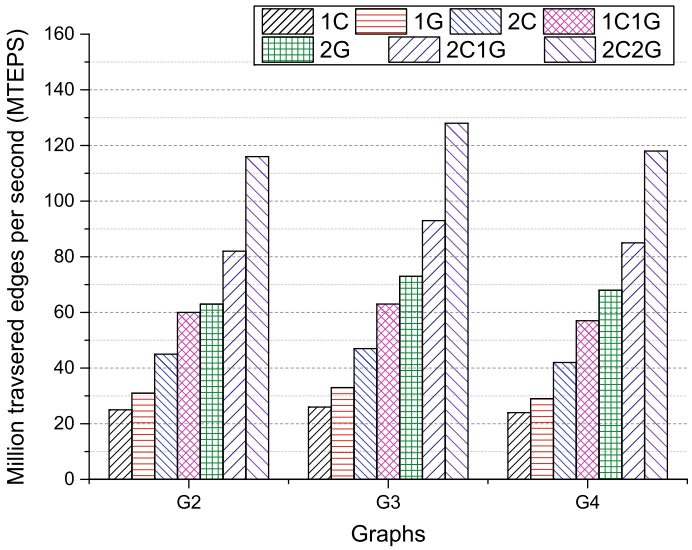

Fig. 7 Performance with different hardware configurations and graphs

hardware's computing power (e.g., adding one processing unit or replacing a CPU with a GPU). In addition, gGraph does not show significant performance degradation when the size of graphs increases from G2 (1.8 billion edges) to G3 (3.7 billion edges), and finally to G4 (8 billion edges). In summary, the experiments manifest the good scalability of gGraph in utilizing more hardware resources and in processing larger graphs.

7.3 Performance on different graphs

Figure 8 compares the performance of all platforms when running page rank on various graphs. Four platforms achieve different performance because they have different designs and use different hardware. As described in Sect. 6.3, GraphChi uses the two CPUs and gGraph uses the two CPUs and two GPUs in the server. PowerGraph and GPS use the twelve CPUs in the cluster.

GGraph outperforms GraphChi and PowerGraph. On average, gGraph achieves 5.11× and 1.23× performance of GraphChi and PowerGraph, respectively. GPS obtains the best performance among all platforms because of its good design that enables the effective utilization of the resources in the cluster. We can also compare the performance of GraphChi and gGraph on the same hardware by combining Figs. 7 and 8. Figure 8 and the "2C" bar in Fig. 7 show that the average performance of GraphChi and gGraph for graphs G2, G3 and G4 on two GPUs is 27.78 MTEPS and 44.66 MTEPS, respectively. Therefore, with the same hardware, gGraph achieves on average 1.61× performance of GraphChi. If given two more GPUs, gGraph can achieve larger performance improvement over GraphChi.

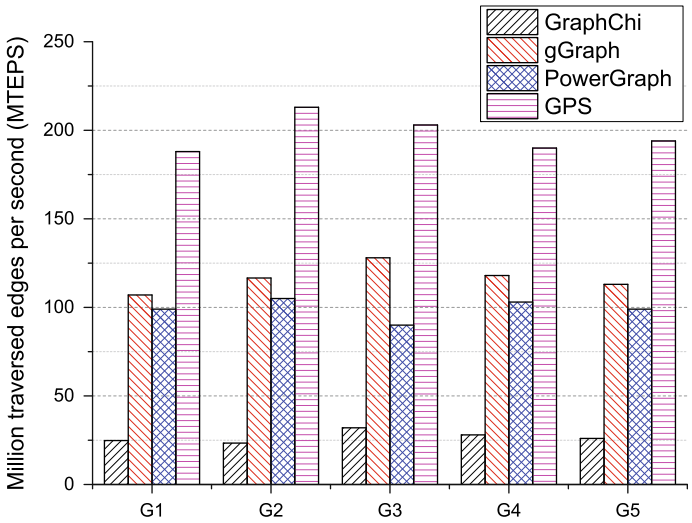

Fig. 8 Performance comparison on different graphs

GGraph outperforms GraphChi and PowerGraph for several reasons. First, gGraph can utilize the processor cores more effectively than GraphChi and PowerGraph. GGraph enables full parallelism by adopting the BSP model and proposing the adaptive load balancing method to remove the performance degradation in the BSP model due to imbalanced load among heterogeneous processing units (i.e. CPUs and GPUs). In contrast, the parallelism of GraphChi and PowerGraph is limited since they employ the asynchronous model and enforce race condition detection for concurrent data updating. Second, gGraph utilizes the partition-block layout of data storage and the load and store unit for efficient data prefetching such that the computation and I/O can be overlapped as much as possible. Third, the GPUs used in gGraph can provide additional computational power in addition to CPUs.

7.4 Performance of different algorithms

Figure 9 compares the performance of all platforms running three algorithms. The result is the average performance on all graphs. Connected component and page rank are graph analysis algorithms while single source shortest path is a graph traversal algorithm. The three algorithms have different criteria to terminate. Page rank will run for a user-specified number of iterations (six iterations in our experiments) then finish. Connected component will stop executing when all connected components are found. The termination criteria for single source shortest path is that the shortest paths from the source vertex to all reachable vertices are found.

In general, platforms achieve better performance in connected component and page rank than in single source shortest path. GraphChi achieves a poor performance in single source shortest path algorithm, since its asynchronous computation model makes it very inefficient in implementing graph traversal algorithms such as BFS and single

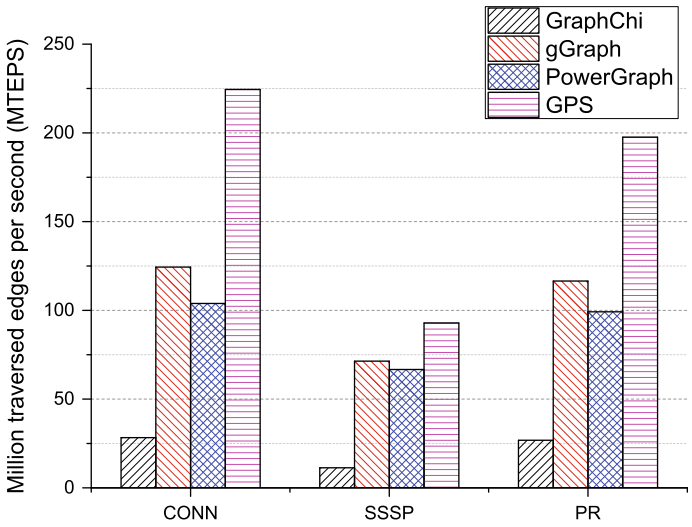

Fig. 9 Performance comparison of all algorithms

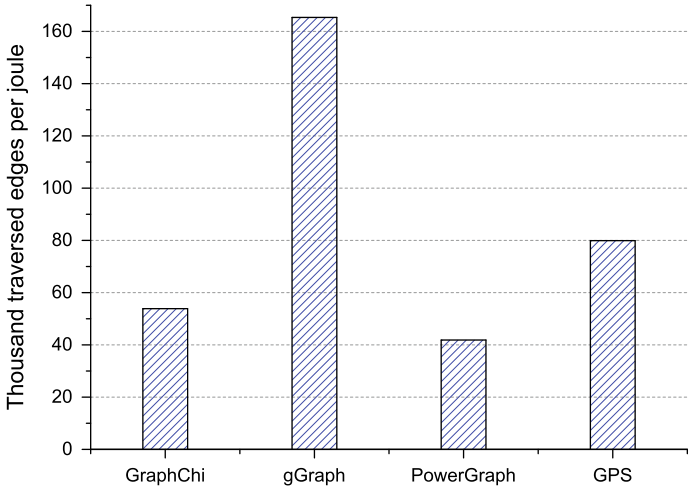

Fig. 10 Energy efficiency of all platforms

source shortest path. GPS and gGraph achieve the best and the second best performance among all platforms, respectively. The good performance of GPS comes from its optimizations over Pregel, including a more efficient global synchronization mechanism, better graph partitioning and repartitioning schemes and so on.

7.5 Energy efficiency

Figure 10 compares the energy efficiency of four platforms expressed as thousand traversed edges per joule. As defined in Sect. 6.5, the energy efficiency is calculated as the performance (in traversed edges per second) divided by the average power. We try

to use the energy efficiency as a normalized/fair comparison since different platforms use different number of CPUs and GPUs and consume different power. The average measured power of a workstation is 358 W. And the average power of the server is 411 W without GPUs and 671 W with two GPUs running, respectively. The results in the figure are the average of three algorithms on all graphs. The energy efficiency of GraphChi, gGraph, PowerGraph and GPS is 53.87, 165.34, 41.86 and 79.91, respectively. As a non-distributed, hybrid CPU and GPU system, gGraph shows the best energy efficiency because it achieves good performance with limited hardware. The ability to utilize energy-efficient GPUs also improves its energy efficiency. Another non-distributed platform, GraphChi, also achieves comparable or even better performance than distributed platforms. This manifests the advantage of non-distributed system on energy efficiency.

8 Conclusion

This paper introduces a general, disk-based graph processing platform called gGraph which can process billion-scale graphs very efficiently by using both CPUs and GPUs on a single PC. GGraph exploits full parallelism and full overlap of CPU and GPU processing and I/O processing as much as possible—data are pre-fetched prior to the computation. Our experiments show that gGraph outperforms GraphChi and PowerGraph in terms of performance. In addition, gGraph achieves the best energy efficiency among all evaluated platforms. The evaluation results manifest that, while distributed platforms can scale up and provide highest performance, we believe that non-distributed platforms (e.g., SSD-based, hybrid CPU and GPU graph processing platforms) are valuable complements, instead of replacements, to distributed graph processing platforms. These non-distributed platforms are eligible candidates for some energy-critical or economy-critical cases in supercomputing.

Although gGraph is designed for non-distributed computing, some of the techniques in this paper also apply to distributed, hybrid CPU and GPU systems. In the future, we plan to extend gGraph and the techniques to process large-scale graphs in the cloud environment.

Acknowledgments The authors would like to thank anonymous reviewers for their fruitful feedback and comments that have helped them improve the quality of this work. This work is partly supported by the National Natural Science Foundation of China (Grant No. 61202026, No. 61332001 and No. 61373155) and Program of China National 1000 Young Talent Plan.

References

1. Addario-Berry L, Kennedy WS, King AD, Li Z, Reed B (2010) Finding a maximum-weight induced k-partite subgraph of an i-triangulated graph. Discret Appl Math 158(7):765–770
2. Arabnia HR (1990) A parallel algorithm for the arbitrary rotation of digitized images using process-and-data-decomposition approach. J Parallel Distrib Comput 10(2):188–192
3. Arabnia HR, Oliver MA (1989) A transputer network for fast operations on digitised images. In: Arnold D, Ruiter Bd (eds.) Computer graphics forum, vol. 8. Wiley Online Library, pp 3–11
4. Avery C (2011) Giraph: large-scale graph processing infrastruction on Hadoop. In: Proceedings of Hadoop Summit

5. Boldi P, Santini M, Vigna S (2008) A large time-aware web graph. In: ACM SIGIR forum, vol 42. ACM, pp 33–38
6. Chakrabarti D, Zhan Y, Faloutsos C (2004) R-MAT: a recursive model for graph mining. In: SIAM international conference on data mining (SDM), vol 4. SIAM, pp 442–446
7. Chan A, Dehne F, Taylor R (2005) Cgmgraph/cgmlib: implementing and testing cgm graph algorithms on PC clusters and shared memory machines. Int J High Perform Comput Appl 19(1):81–97
8. Che S, Beckmann B, Reinhardt S, Skadron K (2013) Pannotia: understanding irregular GPGPU graph applications. In: IEEE international symposium on workload characterization (IISWC), pp 185–195
9. Fackbook (2015) Facebook statistics. http://www.statisticbrain.com/facebook-statistics/
10. Fang J, Varbanescu AL, Sips H (2011) A comprehensive performance comparison of CUDA and OpenCL. In: 2011 international conference on parallel processing (ICPP). IEEE, pp 216–225
11. Ferrer R, Planas J, Bellens P, Duran A, Gonzalez M, Martorell X, Badia RM, Ayguade E, Labarta J (2011) Optimizing the exploitation of multicore processors and GPUs with OpenMP and OpenCL. In: Cooper K, Mellor-Crummey J, Sarkar V (eds.) Languages and compilers for parallel computing. Springer, Berlin, pp 215–229
12. Gharaibeh A, Costa LB, Santos-Neto E, Ripeanu M (2013) On graphs, GPUs, and blind dating: a workload to processor matchmaking quest. In: 2013 IEEE 27th international symposium on parallel and distributed processing (IPDPS). IEEE, pp 851–862
13. Gharaibeh A, Santos-Neto E, Costa LB, Ripeanu M (2013) Efficient large-scale graph processing on hybrid CPU and GPU systems. arXiv:1312.3018
14. Gonzalez JE, Low Y, Gu H, Bickson D, Guestrin C (2012) Powergraph: distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX symposium on operating systems design and implementation (OSDI), pp 17–30
15. Graph 500 Steering Committee and others: Graph 500 benchmark. http://www.graph500.org/
16. Gregor D, Lumsdaine A (2005) The parallel bgl: a generic library for distributed graph computations. In: Parallel Object-Oriented Scientific Computing (POOSC)
17. Harish P, Vineet V, Narayanan P (2009) Large graph algorithms for massively multithreaded architectures. Centre for Visual Information Technology, I. Institute of Information Technology, Hyderabad, India, Tech. Rep. IIIT/TR/2009/74
18. ILK Research Group, Tilburg University (2015) The size of the world wide web (the internet). http://www.worldwidewebsize.com
19. Isard M, Budiu M, Yu Y, Birrell A, Fetterly D (2007) Dryad: distributed data-parallel programs from sequential building blocks. ACM SIGOPS Oper Syst Rev 41(3):59–72
20. Kang U, Tong H, Sun J, Lin CY, Faloutsos C (2012) Gbase: an efficient analysis platform for large graphs. VLDB J 21(5):637–650
21. Kang U, Tsourakakis CE, Faloutsos C (2009) Pegasus: a peta-scale graph mining system implementation and observations. In: Ninth IEEE international conference on data mining (ICDM). IEEE, pp 229–238
22. Karypis G, Kumar V (1999) Parallel multilevel series k-way partitioning scheme for irregular graphs. SIAM Rev 41(2):278–300
23. Kwak H, Lee C, Park H, Moon S (2010) What is Twitter, a social network or a news media? In: Proceedings of the 19th international conference on World wide web. ACM, pp 591–600
24. Kyrola A, Blelloch G, Guestrin C (2012) Graphchi: large-scale graph computation on just a PC. In: Proceedings of the 10th USENIX symposium on operating systems design and implementation (OSDI), vol 8, pp 31–46
25. Lee J, Han WS, Kasperovics R, Lee JH (2012) An in-depth comparison of subgraph isomorphism algorithms in graph databases. In: Proceedings of the 39th international conference on very large data bases. VLDB Endowment, pp 133–144
26. Low Y, Bickson D, Gonzalez J, Guestrin C, Kyrola A, Hellerstein JM (2012) Distributed graphlab: a framework for machine learning and data mining in the cloud. In: Proceedings of the very large data bases (VLDB) endowment, vol 5, issue 8, pp 716–727
27. Malewicz G, Austern MH, Bik AJ, Dehnert JC, Horn I, Leiser N, Czajkowski G (2010) Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM international conference on management of data (SIGMOD). ACM, pp 135–146
28. Maserrat H, Pei J (2010) Neighbor query friendly compression of social networks. In: Proceedings of the 16th ACM international conference on Knowledge discovery and data mining (SIGKDD). ACM, pp 533–542

29. Merrill D, Garland M, Grimshaw A (2012) Scalable GPU graph traversal. In: ACM SIGPLAN notices, vol 47. ACM, pp 117–128
30. Neo Technology, inc. (2015) Learning neo4j. http://neo4j.com/book-learning-neo4j/
31. Page L, Brin S, Motwani R, Winograd T (1999) The pagerank citation ranking: bringing order to the web
32. Pearce R, Gokhale M, Amato NM (2010) Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In: Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis. IEEE Computer Society, pp 1–11
33. Salihoglu S, Widom J (2013) Gps: a graph processing system. In: Proceedings of the 25th international conference on scientific and statistical database management. ACM, p 22
34. Shao B, Wang H, Li Y (2013) Trinity: a distributed graph engine on a memory cloud. In: Proceedings of the 2013 international conference on Management of data. ACM, pp 505–516
35. Shen J, Fang J, Sips H, Varbanescu AL (2012) Performance gaps between OpenMP and OpenCL for multi-core CPUs. In: 2012 41st international conference on parallel processing workshops (ICPPW). IEEE, pp 116–125
36. Shiloach Y, Vishkin U (1982) An O(log(n)) parallel connectivity algorithm. J Algorithms 3(1):57–67
37. Shun J, Blelloch GE (2013) Ligra: a lightweight graph processing framework for shared memory. In: Proceedings of the 18th ACM symposium on principles and practice of parallel programming (SIGPLAN). ACM, pp 135–146
38. Valiant LG (1990) A bridging model for parallel computation. Commun ACM 33(8):103–111
39. Vineet V, Harish P, Patidar S, Narayanan P (2009) Fast minimum spanning tree for large graphs on the GPU. In: Proceedings of the conference on high performance graphics 2009. ACM, pp 167–171
40. Yang J, Leskovec J (2012) Defining and evaluating network communities based on ground-truth. In: Proceedings of the ACM SIGKDD workshop on mining data semantics. ACM, p 3