Design and Experimental Evaluation of Distributed Heterogeneous Graph-Processing Systems

Yong Guo*, Ana Lucia Varbanescu§, Dick Epema*, and Alexandru Iosup*
*TU Delft, the Netherlands, Email: {Yong.Guo, D.H.J.Epema, A.Iosup}@tudelft.nl
§University of Amsterdam, the Netherlands, Email: A.L.Varbanescu@uva.nl

***Abstract*—Graph processing is increasingly used in a variety of domains, from engineering to logistics and from scientific computing to online gaming. To process graphs efficiently, GPU-enabled graph-processing systems such as TOTEM and Medusa exploit the GPU or the combined CPU+GPU capabilities of a single machine. Unlike scalable distributed CPU-based systems such as Pregel and GraphX, existing GPU-enabled systems are restricted to the resources of a single machine, including the limited amount of GPU memory, and thus cannot analyze the increasingly large-scale graphs we see in practice. To address this problem, we design and implement three families of distributed heterogeneous graph-processing systems that can use both the CPUs and GPUs of multiple machines. We further focus on graph partitioning, for which we compare existing graph-partitioning policies and a new policy specifically targeted at heterogeneity. We implement all our distributed heterogeneous systems based on the programming model of the single-machine TOTEM, to which we add (1) a new communication layer for CPUs and GPUs across multiple machines to support distributed graphs, and (2) a workload partitioning method that uses offline profiling to distribute the work on the CPUs and the GPUs.**

We conduct a comprehensive real-world performance evaluation for all three families. To ensure representative results, we select 3 typical algorithms and 5 datasets with different characteristics. Our results include algorithm run time, performance breakdown, scalability, graph partitioning time, and comparison with other graph-processing systems. They demonstrate the feasibility of distributed heterogeneous graph processing and show evidence of the high performance that can be achieved by combining CPUs and GPUs in a distributed environment.

I. Introduction

Increasingly large graphs are being generated every day, not only by big companies such as Facebook [1] and LinkedIn [2], but also by Small and Medium Enterprises (SMEs) [3] such as Wikimedia for online encyclopedia [4], Friendster for social networks [5] and XFire for online gaming [6]. To process these graphs, many graph-processing systems, using a variety of hardware platforms (e.g., multiple CPUs, GPUs, or combinations thereof) have been designed and implemented. With CPUs and GPUs becoming increasingly more powerful and affordable, SMEs who could previously invest only in CPU-based commodity clusters can now afford to buy a heterogeneous environment. However, current graph-processing systems cannot operate on both distributed and heterogeneous settings. This raises the important research question of *How to design a distributed and heterogeneous graph processing system?* In this work, we explore systematically this question, through the design and experimental evaluation of three families of distributed heterogeneous graph-processing systems.

Typical *distributed CPU-based* graph-processing systems such as Pregel [7], GraphX [8], and PGX.D [9] can handle large graphs by using multiple machines, but choose to ignore the additional computational power of accelerators because of the increased complexity of the programming environment. GPU-enabled systems, on the other hand, can accelerate graph processing considerably [10], but choose to ignore the distributed environment because of the added complexity of (multi-layered) partitioning. For example, Medusa [11] and Gunrock [12] can utilize multiple GPUs on a *single* machine and TOTEM [13] is a *single-machine* heterogeneous graph-processing system that can use one CPU and multiple GPUs. MapGraph [14], [15] can use GPUs from multiple machines.

In this work, we combine the scalability of distributed CPU-based graph-processing systems with the computational power and energy efficiency of GPU-enabled graph-processing systems. Specifically, we design and implement distributed heterogeneous graph-processing systems that can use *both* the CPUs and the GPUs of multiple machines. Our study bridges the gap between existing distributed CPU-based systems and GPU-enabled systems for large-scale graph processing.

We explore the design space of these distributed heterogeneous systems with a focus on partitioning. Graph partitioning is mandatory for systems with multiple processing units [16], [17]. Well balanced partitions can improve the performance of graph-processing systems, but the way to build and balance them depends heavily on the system characteristics. In the presence of heterogeneity in the platform, balance is difficult to determine and achieve [18], [19].

In this work, we propose three different graph-partitioning architectures: Distributed-Parallel (DP), Parallel-Distributed (PD), and Combined (C). For the DP and PD architectures, we select and combine existing graph-partitioning policies that have promising characteristics for the respective phases; for C systems, we design a new policy to construct balanced partitions for multiple CPUs and GPUs.

To understand the performance differences between these systems and policies in the context of real-life graph processing applications, we implement our distributed CPU+GPU systems on top of the popular system TOTEM, from which we adopt the programming model, the data structures, and several optimization techniques. To enable the inter-machine communication necessary for a distributed system, we enhance

978-1-5090-2453-7/16 $31.00 © 2016 IEEE
DOI 10.1109/CCGrid.2016.53

the communication layer of TOTEM (a single-machine system in which data is transferred only between the CPU and the local GPUs) to use MPI [20] and GPUDirect [21]. We further address other technical issues, such as separately building partitions on working machines and aggregating results. Finally, because TOTEM doesn't provide a method to compute the workload partitioning between the CPU and the GPU, we propose a new method that leads to balanced partitions among processing units. Our method is based on the approach proposed by Shen et al. [18], which uses an offline profiling method to compute the relative capability of the CPU and the GPU. We extend this method to determine a balanced partitioning of the input datasets on the CPU and the GPU.

We comprehensively evaluate the performance of three families of distributed heterogeneous graph-processing systems with different graph-partitioning policies. In our experiments, we show how our systems can process large-scale graphs faster than single-machine GPU-enabled systems and distributed CPU-based systems. Moreover, we show that the systems we design can analyze large-scale graphs that cannot be handled by single-machines systems.

Our contribution is four-fold:

1. We explore the design space of distributed heterogeneous graph-processing systems (Section III). Specifically, we explore three families of such systems using different graph-partitioning architectures.
2. We design and select graph-partitioning policies for the three families of systems (Section III).
3. For each of the three families of systems, we implement the first working system (Section III).
4. We conduct comprehensive experiments to evaluate the performance of our distributed heterogeneous graph-processing systems (Section IV).

II. Extended BSP-based programming model for graph-processing systems

Through comprehensive experiments, we have already evaluated the performance of existing GPU-enabled graph-processing systems [22]. Our past results indicate that TOTEM is, among the systems we have tested, the most reliable one. Furthermore, TOTEM has clear and comprehensive data structures for representing graphs and partitions, and several optimization techniques for improving performance. In this section, we introduce the programming model and the most important features of TOTEM (as shown in Figure 1).

TOTEM is a vertex-centric system following the Bulk Synchronous Parallel (BSP) programming model [23]. In the BSP model, iterative graph algorithms are executed in multiple, consecutive supersteps. Each superstep coordinates processing data in parallel, across physical processing units (P). Graph vertices are partitioned across processing units. To avoid processing all vertices during each superstep, vertices can be activated and only the active vertices are processed. Each superstep includes three phases: a *computation phase* during which all active vertices in each partition execute the same operations of the graph algorithm; a *communication phase* during which vertices send messages via *cut edges*, that is, edges whose vertices belong to partitions managed by different processing units, to other partitions; and a *synchronization phase*, which guarantees that all messages are transferred.

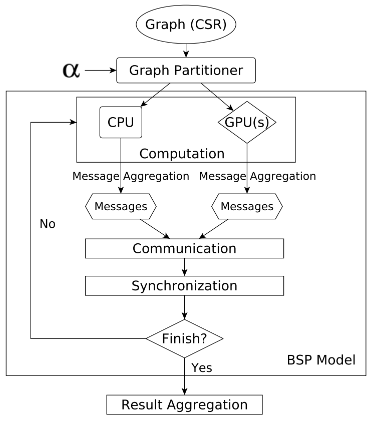


Fig. 1. The BSP-based programming model and features of a single-machine heterogeneous graph-processing system.

To use memory efficiently, TOTEM uses the Compressed Sparse Rows (CSR) [24] data format to represent graphs and their partitions. The CSR format includes two arrays, the vertex array V and the edge array E. Each element in V stores, for a vertex, the head of its list of neighbours, as an index in E, and E stores for each vertex index a list of its neighbours. Input graphs are split into multiple partitions, which are further assigned to the CPU and the GPU(s).

A message aggregation technique is implemented in TOTEM to reduce the number of messages sent via cut edges between partitions. The messages sent from vertices in a partition to the same remote vertex are temporarily buffered. All messages for the same remote vertex are combined into one message before sending. To allow for this aggregation, each partition maintains two sets of buffers: the outbox buffers and the inbox buffers. Each outbox buffer stores messages to a remote partition, and each inbox buffer collects messages from a remote partition. Thus, for each partition, the system has $|P| - 1$ outbox and $|P| - 1$ inbox buffers ($|P|$ is the number of partitions). This message aggregation technique can significantly reduce communication [13].

For heterogeneous CPU+GPU systems, a crucial operation is partitioning the workload between CPUs and GPUs. For many graph processing algorithms, the computation workload can be heavily related to the number of edges of the input graph [9], [13], [25]. Thus, partitioning the workload is equivalent to determining the fraction of the edges to be put on the CPU(s), which both in TOTEM and in our system is denote by α, with the remainder to be put on the GPU(s). Existing studies have not proposed a method to select this value. In Section III-E, we describe our method to calculate the value of α. This method is based on existing work on heterogeneous CPU+GPU systems [18], [19], which we extended and adapted to graph-processing workloads.

III. The Design of Distributed Heterogeneous Graph-processsing Systems

In this section we discuss the design of our three families of distributed heterogeneous systems for graph processing. We further present the partitioning policies we have selected and/or designed for these systems, and we discuss the most challenging aspects of the implementation.

A. Three families of distributed heterogeneous systems

To extend single-machine graph-processing systems to a distributed architecture, graph partitioning is a key aspect for both functionality and performance. In this section, we focus on the architecture of graph partitioning in distributed heterogeneous systems.

Balanced partitions often lead to good system performance. To achieve balanced partitions in distributed heterogeneous systems, the graph partitioner (see Figure 1) must consider three main aspects: inter-machine workload distribution, intra-machine workload distribution (i.e., between the CPU and the GPU), and communication minimization. To explore these different aspects, we design three families of distributed heterogeneous graph-processing systems, with the architectures depicted in Figure 2. We describe these architectures further.

Distributed-Parallel (DP) systems: the partitioner takes two phases to partition the input graph on the processing units. First, in the *distributed phase*, the partitioner assigns vertices to different computing machines, similar to the graph partitioning approach used by many distributed graph-processing systems such as Pregel [7], Giraph [26], or GPS [27]. Next, in the *parallel phase*, each machine further splits the subgraph it received across its local CPUs and GPUs, similar to the actions taken in TOTEM.

Parallel-Distributed (PD) systems: in contrast to DP systems, PD systems reverse the sequence of the distributed phase and the parallel phase: the graph is first divided into two subgraphs, one to be processed by the CPUs and the other to be processed by the GPUs, and then each subgraph is further distributed across CPUs and GPUs.

Combined (C) systems: unlike DP and PD systems, the combined systems use a single-phase partitioning, the *combined phase*. The partitioner directly assigns vertices to processing units, considering both the CPUs and the GPUs of the entire system. To still achieve balanced partitions, the heterogeneity of processing units is the main challenge that must be tackled. We describe our approach to this challenge in Section III-D. We note that our work is the first to consider a combined partitioning approach for heterogeneous systems.

For all three families of distributed heterogeneous systems, the graph partitioner operates on a single master machine. The partitioned data (vertices, edges, etc.) are then sent to the working machines. Besides being out of the scope of this work, a distributed partitioner is also non-trivial to implement, so we leave its design and implementation for future work.

TABLE I
Four classes of graph-partitioning policies.

Class	Examples
Computation-focused	IO [9], HIGH [13], LOW [13]
Communication-focused	METIS [28], LDG [29]
Computation-communication	MW [17], MI [17]
Unfocused	hash [7], random [29], chunking [30]

B. Classification of partitioning policies

Graph-partitioning has been studied for many years and many policies, with different goals, have been proposed [16]. For example, the main target of the state-of-the-art graph partitioner *METIS* [28] is to reduce the number of edge cuts between partitions, which leads to less communication. The *total-degree balanced* policy (IO) used in the PGX.D graph-processing system [9] aims to balance the total degrees of all partitions, allowing each computing unit to have the same workload. Based on their goals and focuses, we identify four classes of graph-partitioning policies, and summarize them in Table I. We discuss each of the four classes below.

The **computation-focused** policies focus on achieving balanced computation workloads across the processing units, with *no* consideration for the edge cuts between partitions. Policies in this class are based on the intuition that the computation workload of graph-processing algorithms can occur incrementally, and mainly along the edges of graphs [9], [13], [25]. Many computation-focused policies, such as the IO policy used by the PGX.D system, are designed to balance the in-degree and/or out-degree of partitions. Considering the utilization of the cores of processing units, in particular for GPUs, the vertex-degree centric policies have been used in heterogeneous CPU+GPU systems. For example, the HIGH and LOW policies used by TOTEM fall into this class. For both of these policies, the vertices of a graph are first sorted by their out-degrees, and then they are split up into two parts. For the HIGH (LOW) policy, the part with higher (lower) out-degrees is assigned to the CPU and the remainder to the GPU.

The **communication-focused** policies are proposed mainly to minimize the communication between partitions. Many *traditional heuristics* adopt theoretical methods to achieve minimum communication, such as METIS and its family of partitioning policies [31]. Emerging *streaming partitioning policies*, which treat vertices as a stream and assign them one-by-one instead of in bulk, also include many policies to reduce the communication. For example, the LDG policy [29] places a vertex to a partition that already has most of the new vertex's neighbours. Some of the communication-focused policies also make an effort, albeit minimal, to balance computation workload, for example the LDG policy through a penalty function to avoid that too many vertices accumulate to one partition and thus lead to highly imbalanced computation.

The **computation-communication** policies consider *both* the computation and the communication workloads. For example, the *Min-Workload* (MW) policy [17] combines the two workloads, by greedily assigning vertices to partitions that incur minimum combined workload. The *Min-Increased* (MI) policy [17] places vertices with the least increase of workload.

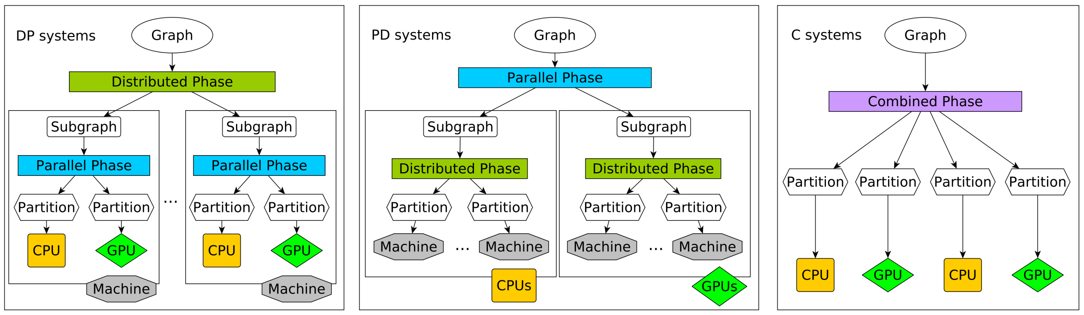


Fig. 2. Three architectural families of distributed heterogeneous graph-processing systems: DP (left), PD (middle), and C (right) systems.

The **unfocused** class includes policies designed for simplicity—either to reduce the implementation effort, or because partitioning is not believed to deliver a balanced workload. For example, the *hash* policy is commonly used by Pregel-like systems [7], [26]. The *random (R)* policy is another lightweight unfocused policy that randomly places vertices to different partitions. Non-random policies, such as chunking [30], take subsets of equal length from the input graph file and place them round-robin in different partitions.

C. Selection of partitioning policies

As discussed in Section III-A, our three families of distributed heterogeneous graph-processing systems use one or two phases for partitioning graphs: a distributed and a parallel phase, or a combined phase. For each phase, any graph-partitioning policy could in principle be selected and used. However, because of policy complexity and the architecture of the graph-processing systems, some policies may perform poorly in specific partitioning phases. In this section we discuss the selection of suitable policies for each family of systems; we will evaluate these choices in Section IV.

a) The distributed phase: For the distributed phase, we need to address the load-balancing in the distributed system. Because of message aggregation, our systems already reduce inter-machine communication. Moreover, the partitioning policies in both communication-focused and computation-communication-focused classes are very expensive—for example, METIS (communication-focused) takes more than 8.5 hours to partition a large graph (1.4 billion edges) with a typical machine that could be used by an SME (3.6 GHz CPU and 16 GB memory) [32]. Therefore, we select the partitioning policies for the distributed phase from the computation-focused class, as they will provide fast partitioning and good balancing of the computation workload, leading to reasonable load balancing.

To understand how to balance the workload, we observe that the computation can be divided into two parts: one for processing incoming messages, and another one for applying updates and creating outgoing messages. The partitioning policy should aim to balance the substantial part of the workload, requiring a decision to be made between balancing the in-degree or the out-degree of the partitions. Because of local message aggregation, the number of incoming messages for a partition is at most $(|P|-1)\times|V_p|$, with V_p the set of vertices of partition p). Because $|V_p| << |E_p|$, with E_p the set of edges of partition p, this in-message processing part of the workload is significantly smaller than the updates and out-going message preparation part, which requires computation for each edge. With this knowledge, our policy for the distributed phase must focus on balancing the out-degree. We select the *out-degree balanced* policy (O), which is one of the partitioning policies proved successful by PGX.D [9]. By the O policy, vertices are assigned to the first $|P|-1$ partitions until the sum of vertex out-degrees in each partition reaches $|E|/|P|$. The last partition takes all remaining vertices.

b) The parallel phase: For the parallel phase, we need to address the heterogeneity of the CPU(s) and the GPU(s). We cannot use here a degree balanced policy (such as O or IO from PGX.D), because the GPU can process much faster than the CPU when they have similar computation workload. TOTEM has proposed for the parallel phase two policies—HIGH and LOW—to handle heterogeneity. Given the results from both [13] and [33], which indicate that HIGH and LOW can both be successful for different algorithms and datasets, we select both the HIGH and LOW policies for independent use in the parallel phase.

c) The combined phase: There is no policy that addresses the heterogeneity of the CPU and the GPU in distributed environments. Thus, we design a new policy in Section III-D.

d) The random policy: Last, we also select the random policy from the unfocused class for the distributed and the combined phases. The random policy is lightweight and easy to implement. We aim to use it as a control policy for the O policy in the distributed phase, and for the newly designed policy in the combined phase. We do not use the random policy in the parallel phase, because previous studies on the performance of TOTEM [13], [33] have shown that in most cases, the random policy is the worst performing one.

D. The design of a profiling-based greedy policy

In this section, we design a *profiling-based greedy* (PG) policy for the combined phase of combined systems. The PG policy belongs to the computation-focused class.

Requirements. Combined systems need to directly place vertices on CPUs and GPUs in a single combined phase (see Section III-A). To balance the workload, the partitioning policy of combined systems needs to address the heterogeneity of CPUs and GPUs.

None of the existing partitioning policies is able to deal with this type of heterogeneity. For existing computation-focused policies, many of them are proposed to distribute graphs for CPU-based systems on homogeneous clusters, but these policies do not focus on GPUs. Other computation-focus policies, such as the HIGH and LOW policies used in TOTEM, are designed without considering distributed environments.

General idea of the new PG policy. To balance the assignment of vertices across both CPUs and GPUs, the policy needs to assess the relative computation abilities of the GPU and the CPU. We use the ratio of the GPU and CPU capabilities, r, to estimate a balanced workload—i.e., a workload ratio of $r : 1$ for the GPU versus the CPU constitutes a balanced workload. These capabilities can be obtained from an offline profiling process as introduced in Section II and discussed in more detail in Section III-E. Our policy is inspired by streaming partitioning policies: we treat the vertex list as a stream, and we place the vertices from this stream, one-by-one, in the partition currently having the smallest computation workload. The PG policy simplifies the process of graph partitioning by considering only one vertex at a time, and achieves balanced partitions for CPUs and GPUs.

Technical details. We define the computation workload on the CPU as the sum of the vertex out-degrees of all vertices in the partition placed on the CPU. The computation workload on the GPU is the similar sum computed for vertices placed on the GPU, but divided by r to account for the computation ratio of the GPUs to CPUs. In the PG policy, we maintain an array, indexed by partition, of the computation workload of all partitions. For each next vertex, we search for the partition with the smallest workload, and place it there. We update the computation workload of this partition by adding to it the out-degree of the added vertex. If the partition is for a GPU and the required memory is too close to the GPU memory capacity, the partition is removed from the computation workload array and will not be further considered for the remainder of the partitioning process. When all GPU partitions are full, all remaining vertices go to the CPUs.

Limitations. The computation ratio of the GPUs to CPUs must be known for the PG partitioning policy to work. The profiling process to calculate this ratio is time consuming, requiring many experiments (see Section III-E). The more experiments, the more accurate the computed ratio is. Because the partition quality of this policy strongly relies on the accuracy of this ratio, when hardware infrastructure changes, the profiling process should be re-executed for better accuracy.

Comparison with other policies. The complexity of the PG policy is low, because it only needs to maintain the computation workload array for all partitions and to search for the partition with least workload for each assignment. DP and PD systems use two-phase partitioning, combining two policies, which means they access each vertex at least twice, to decide its final partition. Moreover, although O and Random have low complexity, HIGH and LOW need time-consuming sorting of vertices. We expect PG to be faster than PD and DP. We further explore the partitioning time in Section IV-F.

E. Implementation details

In this section, we describe the non-trivial elements of implementing the three families of distributed heterogeneous systems, and of their partitioning policies. We begin from the open-source code of TOTEM, which already implements the general single-machine model introduced in Section II. To this code, we add a profiling component to determine r, the ability to operate as a *distributed* system with any of the three architectures, and the partitioning policies.

Profiling the relative computation ability of the CPU and of the GPU. Previous studies have shown that heterogeneous systems can outperform CPU-only or GPU-only systems in many application domains, including graph processing, but the performance gain is very sensitive to a good workload partitioning [13], [18]. Determining a good workload partitioning is equivalent, in the case of our systems, to computing the right α, i.e., the right workload fraction to be placed on the CPU (Section II). This fraction depends on the relative ability of the CPU and GPU to process a given workload, i.e., how much slower is the CPU compared to the GPU. Previous studies have already shown that using hardware performance models is unfeasible [34], and using the theoretical bounds of the hardware platforms does not give accurate results [18], [19]. Therefore, we propose a profiling method to understand the computation heterogeneity between these processing units. In our offline profiling method, we let the CPU and the GPU compute the same workload, and calculate the ratio of the CPU run time to the GPU run time. Due to the irregular, data-dependent nature of graph processing, we repeat the experiment for multiple runs and compute an average (see Section IV-B for details) to obtain an accurate execution profile describing r.

Ideally, the most accurate values of r are computed using the graphs and algorithms that are to be used at runtime. However, such a profiling method would be too expensive to use in practice, and would cancel out the partitioning speed of our selected partitioning policies. Therefore, we choose to trade-off accuracy for applicability, and implement our profiling method using a 4-step micro-benchmarking strategy.

Step1. We select a representative graph processing algorithm. Specifically, we use PageRank because it is stable in its performance (e.g., by contrast, BFS shows high performance variability depending on the root of the search).

Step2. We use five synthetic datasets: Scale-20 to Scale-24, created by the Graph500 generator [35].

Step3. We randomly partition each graph and set 10% to 50% (with a step of 10%) total edges on the CPU and the remaining (90% to 50%) on the GPU. Then, we reverse the workload of the CPU and the GPU for each partitioning. Thus, we can obtain 10 pairs of CPU and GPU run times for processing the same workloads. We calculate the computation ratio r as the CPU run time over the GPU run time. We repeat

TABLE II
GRAPH-PARTITIONING POLICIES FOR PARTITIONING PHASES.

Phase	Policies
Distributed phase	Out-degree balanced (O), Random (R)
Parallel phase	HIGH (H), LOW (L)
Combined phase	Profilling-based Greedy (PG), Random (R)

the process with 5 random seeds for partitioning, and derive a mean value of r for each workload.

Step4. We observe the correlation between r and the different graph sizes (because there is no single value of r for all graph sizes), and we determine how to select r for different graphs (Section IV-B). We then calculate the fraction α as $\alpha = \max\{\alpha_l, 1/(r+1)\}$, where α_l is derived from the limitation to ensure that the GPU is not out of memory. As we know the data structures for representing partitions on GPUs, α_l can be estimated using the vertex and edge counts of the partition.

Communication in the distributed system. To connect the CPUs and GPUs on multiple machines, we extend the communication part of the TOTEM system. We use MPI [20] and, where available, the Nvidia GPUDirect [21] technology to communicate between processing units in our distributed heterogeneous systems. GPUDirect eliminates the copy process between the CPU and the GPU(s), which means messages can be directly transferred between each pair of processing units with low overhead. The usage of GPUDirect improves the performance of delivering messages and simplifies the coding effort. We use MPI barriers to ensure that all messages are synchronously delivered.

Implemented Policies. Based on the selection and design of partitioning policies, we summarize in Table II the policies we consider and implement for the partitioning phases of the three families of systems.

Other distributed systems aspects. We deploy the graph partitioner on a master machine. For each family of distributed heterogeneous systems, we implement all the partitioning policies or policy combinations we have selected or created in Sections III-C and III-D, respectively. After partitioning the input graph, the master sends all data to the working machines, partition by partition. Working machines simultaneously reconstruct (build) their partitions on each processing unit. We use the master to control the process of executing the graph algorithm. For each iteration in the execution of the graph algorithm, the master collects information from working machines, checks if all partitions have finished their execution, and determines if execution should be stopped. The master is also responsible for aggregating updates from all partitions to the original graph.

IV. EXPERIMENTAL RESULTS

In this section, we present the experiments conducted to evaluate the performance of our three families of distributed heterogeneous systems. We introduce our experimental setup in Section IV-A, and summarize it in Table III. Our experiments include an evaluation of the profiling method (Section IV-B) and a thorough evaluation of our three families of distributed heterogeneous systems, using algorithm run time (Section IV-C), a breakdown of algorithm run time (Section IV-D), and scalability (Section IV-E). We further analyze the partitioning time for different policies (Section IV-F), and provide a performance comparison between our systems and other graph-processing systems (Section IV-G).

TABLE III
EXPERIMENTAL SETUP OF THE EXPERIMENTS IN SECTION IV.

Section	Algorithms	Datasets	Metric	Machines
IV-B	All	Scale-20 to Scale-25	Algorithm run time	1, 4
IV-C	All	G1 to G5	Algorithm run time	4
IV-D	PageRank	G4	Breakdown	4
IV-E	All	G4, G5	Scalability	1-10
IV-F	-	Scale-20 to Scale-25	Partitioning time	1-10
IV-G	All	G1 to G5	Algorithm run time	1, 4

TABLE IV
SUMMARY OF DATASETS USED IN THE EXPERIMENTS.

	Graph	$\|V\|$	$\|E\|$	**d**	$\bar{\mathbf{D}}$
G1	WikiTalk (D)	2,388,953	5,018,445	0.1	2
G2	DotaLeague (U)	61,171	101,740,632	2,719.0	1,663
G3	Datagen_p10m (D)	9,749,927	687,174,631	0.7	70
G4	Scale-25 (U)	17,062,472	1,047,207,019	0.4	61
G5	Friendster (U)	65,608,366	3,612,134,270	0.1	55

$|V|$ and $|E|$ are the vertex count and edge count of the graphs, **d** is the link density ($\times 10^{-5}$), and $\bar{\mathbf{D}}$ is the average vertex out-degree. (D) and (U) stand for the original directivity of the graph. For each original undirected graph, we transform it into a directed graph (see Section IV-A).

A. Experimental setup

Hardware: We conduct our experiments on the DAS4 cluster [36]. All machines we used in our experiments are equipped with an Nvidia GeForce GTX 480 GPU (1.5 GB onboard memory) and an Intel Xeon E5620 2.4 GHz CPU (24 GB memory). The machines are connected by 24 Gbit/s InfiniBand. For the scalability test, we vary the number of working machines from 1 to 10. Our systems need one extra machine as the master.

Algorithms: Based on our literature survey on graph processing [37], we select 3 popular graph-processing algorithms. These are Breadth First Search (BFS), PageRank, and Weakly Connected Component (WCC). We use the same implementation as in our previous study [22]. For BFS on each graph, we use the same source vertex. For PageRank, we set the maximum number of iterations to 10 as the only termination condition. WCC does not have any specific configuration.

Datasets: We select 5 graphs with various characteristics, as seen in Table IV. We include two real-world graph from SNAP [38] (i.e., WikiTalk and Friendster), and one from the Game Trace Archive [39] (i.e., DotaLeague). We also use two synthetic graphs, Scale-25 and Datagen_p10m, from Graph500 [35] and the LDBC generators [40], respectively. For undirected graphs (G2, G4, and G5), we use two directed edges to represent an undirected edge, as required by the CSR format. The WCC algorithm decides two vertices are connected if there is an edge between them. Thus, for the WCC algorithm on directed graphs (G1 and G3), for each pair of vertices that are connected with only a single directed edge, we create a reverse. The new graphs are *G1WCC* and *G3WCC*, with edge counts 9,313,364 and 1,374,349,262, respectively.

Notation for system-policy configuration: We selected different policies for different phases of three families of systems

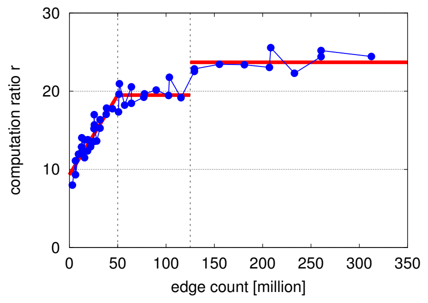


Fig. 3. The relationship between the computation ratio and the edge count.

(Table II). We use the notation of “System-Policy/Policies” to denote the system-policy configuration. For example, C-PG stands for the combined system using the PG policy, and DP-OH indicates the DP system using the O and H policies. The DP and PD systems need α as an input parameter, and C-PG needs the computation ratio r. C-R does not need any input parameter.

Further configuration and settings: We use CUDA 5.5 as the GPU compiler, Intel TBB 4.1 [41] for sorting vertices by their out-degrees, and Open Mpi 1.8.2 for sending messages. We repeat each experiment 10 times and report the mean value. We do not show error bars because the results from different runs are stable, with the largest variance under 5%.

B. Calculating r and α

In this section we discuss the computation ratio r for our machines, and we derive the computation workload fraction α for CPUs for our graphs.

Key findings:

- The computation ratio r varies with the number of processed edges.
- The values of α obtained from PageRank can help BFS and WCC achieve good performance.

Following our micro-benchmarking strategy (Section III-E), we obtain different values for r, all ranging between 8.0 to 25.0. Figure 3 shows how r relates to the number E_m of millions of processed edges. Using regression for the smallest graphs and approximating r as being constant for the other ranges, we find the following trends for r:

$$r = \begin{cases} 9.3 + 0.2 \times E_m, & 0 < E_m \leq 50 \\ 19.5, & 50 < E_m \leq 125 \\ 23.7, & 125 < E_m \leq 350 \end{cases}$$

When using multiple machines, each machine will have a value of r that corresponds to the number of edges it has to process. In our experiments, the edges are evenly distributed, because we use identical machines and a load-balancing driven policy. Thus, we can use the same value of r for all machines. Because of the GPU memory limitation, the maximum value of E_m per working machine is about 350. All these values are likely to change for different machine configurations (i.e., different GPUs and CPUs).

We calculate α for each experiment in the following sections with different machine counts and graphs. For example, α for graphs G1 to G5 on 4 working machines is 0.09, 0.06, 0.04, 0.04, and 0.85, respectively.

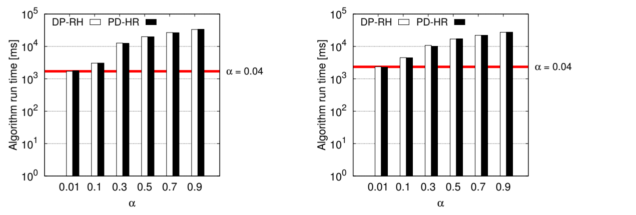


Fig. 4. The algorithm run time for BFS (left) and WCC (right) for different values of α (vertical axes have a logarithmic scale).

Our micro-benchmarking strategy uses PageRank for determining α (Section III-E). We preserve the same values for α for all 3 algorithms. To determine how suitable α is for the other algorithms, we run BFS and WCC on the G4 graph on 4 working machines using the DP-RH and PD-HR configurations with different values of α. We compare the algorithm run time of using these values with the one obtained using the calculated value of 0.04. Figure 4 shows these results (the horizontal line represents the algorithm run time for $\alpha = 0.04$ of DP-RH, which is very similar to PD-HR). For both BFS and WCC, the calculated α leads to the best performance, with the only exception when using the value of 0.01 of PD-HR.

C. Overview of the performance of three families of systems

In this section, we analyze algorithm run time, defined as the time for actually executing the graph algorithm; algorithm run time does not include the time spent on operations like initialization and result aggregation, and includes no system overhead [22].

Key findings:

- There is no overall winner, but C-R is in general the worst performing architecture.
- Our new PG policy for combined systems shows good performance.

Figure 5 shows the algorithm for all combinations of algorithms, systems, and datasets. The results are similar for all algorithms: no system-policy configuration outperforms the others in all cases. C-PG is typically in Top 3. C-R performs the worst because it has no consideration for the heterogeneity of the CPU and the GPU. When we fix the policy used for the distributed phase, and change the policy for the parallel phase, we can compare the influence of the HIGH and LOW policies. In almost all cases, the performance of the HIGH policy is better. We also notice that for G5, the performance of different system-policy configurations is very similar. This happens because $\alpha = 0.85$, and therefore the CPU dominates the overall algorithm run time. For G4, we need to set α to 0.35 for DP-OH and PD-HO to ensure that all partitions assigned to GPUs do not break the GPU memory limitation. We further analyze this setting of α for DP-OH in Section IV-D.

D. Breakdown of algorithm run time

We further break down the algorithm run time into CPU- and GPU-computation time, and communication time, and

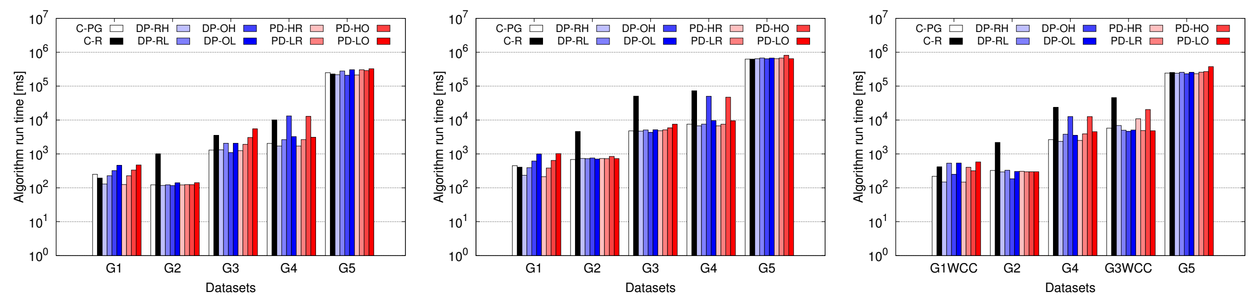


Fig. 5. The algorithm run time of BFS (left), PageRank (middle), and WCC (right) on 5 datasets for all system-policy configurations (vertical axes have a logarithmic scale). Datasets are sorted in increasing order of their edge counts.

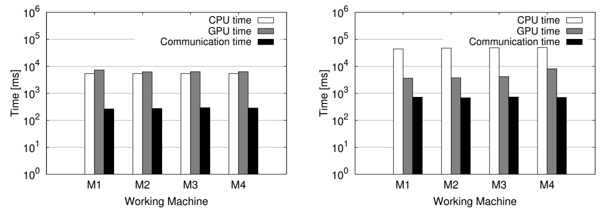


Fig. 6. The breakdown of the algorithm run time of C-PG (left) and DP-OH (right, $\alpha = 0.35$) when running PageRank on graph G4 on each of the four working machines (vertical axes have a logarithmic scale).

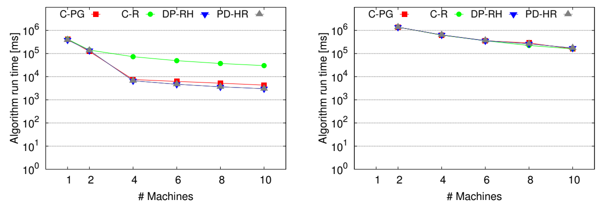


Fig. 7. The scalability of running PageRank for G4 (left) and G5 (right) (vertical axes have a logarithmic scale). We failed to run G5 on 1 machine due to resources limitation.

discuss their impact on performance.

Key findings:

- The computation time is the dominant part of the algorithm run time.
- C-PG can achieve a balanced intra- and inter-machine computation workload.
- The O policy may lead to poor performance and needs to be tuned.

The breakdown of the algorithm run time per machine of PageRank on G4 is presented in Figure 6 for the C-PG and DP-OH configurations. The communication time is significantly shorter than the computation time (i.e., the maximum of the CPU time and the GPU time), which is empirical evidence for our discussion on message aggregation in Section III-C. For C-PG, the computation times of the working machines are close to each other, and within each machine, the difference between the CPU time and the GPU time is also small. However, for DP-OH, the computation workload is not balanced across the CPU and the GPU, because DP-OH needs to set α to 0.35 in order to hold all partitions on GPUs. Although the edge counts on the GPUs are balanced by DP-OH, the fourth GPU partition has 200 times more vertices than the first GPU partition. This imbalance of the vertex counts is caused by the behavior of the O policy and the input graph G4. In G4, high-degree vertices have small vertex IDs and are assigned to the first machine (and then to the first GPU) by the O policy. The O policy needs tuning to avoid such imbalance.

E. Scalability

In this section, we discuss the scalability of our systems using a number of working machines from 1 to 10.

Key finding:

- Our three families of systems show good scalability.

From Section IV-C, we select the best performing system-policy configurations. These are C-PG for the Combined systems, DP-RH for the Distributed-Parallel systems, and PD-HR for the Parallel-Distributed systems. We also select C-R for comparison. Figure 7 depicts the algorithm run time of PageRank for the G4 and G5 graphs (the results we obtained for BFS and WCC are similar, and therefore not included). For G4, using up to 4 machines leads to excellent scalability. The values of α for 1, 2, and 4 machines are 0.8, 0.5, and 0.04, respectively, meaning that increasingly more workload is placed on the GPUs, and is therefore accelerated. However, when using more than 4 machines, the performance gain is not significant, simply because G4 is not large enough to stress larger clusters. For the graph G5, such a scalability limitation is not visible when using up to 10 machines. Even for 10 machines, the algorithm run time is still heavily dominated by the execution on the CPUs.

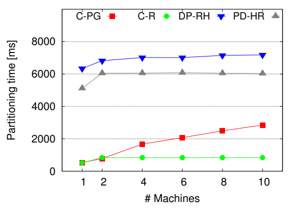
F. Partitioning time

In this section, we analyze the time spent on partitioning graphs for different system-policy configurations.

Key findings:

- C-PG has shorter partitioning time than DP-RH and PD-HR. Its partitioning time increases with partition count.
- The size of graphs can significantly influence the partitioning time, especially for DP and PD systems.

We run two sets of experiments, one for partitioning Scale-25 with increasing partition count (i.e., using an increasing number machines), see Figure 8 (left), and one for partitioning 6 Graph500 graphs (Scale-20 to Scale-25) into 8 partitions on 4 machines (4 CPU and 4 GPU partitions), see Figure 8 (right). Since the results of different configurations of DP and



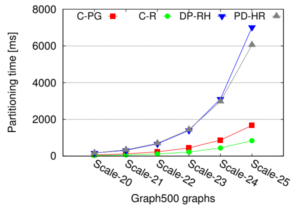


Fig. 8. The time spent on partitioning the Scale-25 graph into different numbers of partitions (left) and on partitioning Graph500 graphs into 8 partitions on 4 machines (right).

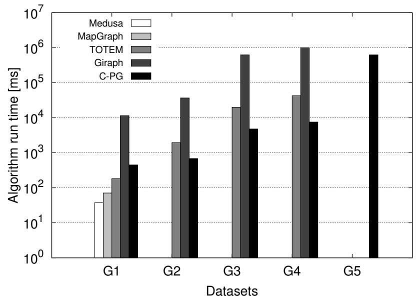


Fig. 9. The algorithm run time of PageRank on 5 datasets with different graph processing systems (the vertical axis has a logarithmic scale, missing bars are explained in the text).

PD systems are similar, we only present the results of DP-RH and PD-HR. The time for partitioning Scale-25 of C-PG increases linearly with the partition count, as for each vertex the operation of searching for the partition with the smallest workload has to be performed. As the time required for this operation increases with the partition count, more time is consumed by C-PG. However, compared with the time-consuming sorting operation in the HIGH and LOW policies used in DP and PD systems, the partitioning time of C-PG is much shorter. Moreover, the gap will increase as the graphs grow larger, see Figure 8 (right).

G. Comparison with other graph-processing systems

In this section, we compare the performance of our systems with other graph-processing systems, including GPU-enabled systems (Medusa [11], MapGraph [14], and TOTEM [13]), and a distributed CPU-based system (Giraph [26]).

Key finding:

- Our system can process all 5 datasets and achieves good performance compared with the other systems.

We select C-PG, and we deploy both our system and Giraph on 4 working machines. The other three systems work on one machine (altough MapGraph claims to be usable on GPUs of multiple machines, the latest publicly available version tested in this section can only work on a single machine). For TOTEM, we set α for each graph according to the rules introduced in [13]. We use the HIGH policy in TOTEM, as it outperforms the LOW policy for our selected datasets.

We run the PageRank algorithm on each system for graphs G1 to G5 and show the algorithm run time in Figure 9. Although Medusa and MapGraph can process the smallest graph G1 much faster, they both fail to run already on the medium-sized graph G2 due to the limited GPU memory. TOTEM fails to run on the graph G5 during the graph partitioning step. In contrast, our system can process all graphs. On the graph G4, our system using 4 machines is about 6 times as fast as the single-machine TOTEM. This super-linear speed-up is due to getting much more acceleration from the GPUs. Finally, our system outperforms Giraph significantly (by a factor of more than 50). The reasons include the acceleration of using GPUs and the reduced communication workload in our system. Giraph fails to run for G5 with 4 machines, but successfully executes with 20 machines, with an algorithm run time closes to that of C-PG with 4 machines. Similar results are observed when running BFS and WCC.

In our previous work [22], we found that TOTEM takes around 7,000 ms to finish PageRank on G4 with 8 GPUs (and no CPUs) in a single machine. In contrast, as shown in Figure 7, C-PG takes only about 5,300 ms for the same job on a distributed system with 8 machines with one CPU and one GPU each. We attribute this performance gain to two reasons: (1) the CPUs play an important role in the performance of our distributed systems, and (2) the inter-machine communication is not really a bottleneck.

V. Related Work

There are three directions of research that contribute to the success of our work: designing graph-processing systems, graph partitioning, and workload partitioning for heterogeneous systems. In this section we place our work in the context of each of these research directions.

Graph-processing systems. There are tens of graph processing systems developed in the past 10 years, each one designed with specific requirements in mind. Among these requirements, support for large-scale graph and efficient use of existing hardware infrastructure are often the most important ones. For example, Pregel [7], Giraph [26], or PGX.D [9] are distributed CPU-based systems that offer a simple, high-level programming model and focus on processing very large graphs with reasonable performance and very good scalability. Other systems, like TOTEM [13], Medusa [11], Gunrock [12], focus on offering users efficient ways to accelerate their graph processing using GPUs on a single machine. Despite their high performance, these systems cannot handle large-scale graphs efficiently. In this work, we combine the advantages of both worlds: we are the first to design and evaluate three families of distributed heterogeneous graph-processing systems.

Graph partitioning. Many graph-partitioning policies and methods have been proposed in various research areas. In our previous work [16], we have summarized the characteristics of existing partitioning policies and classified them into different classes from different perspectives: edge-cut [7] and vertex-cut [42] , static [9] and dynamic [27], and traditional heuristics [31] and streaming policies [29]. In this work, we combine existing policies for parallel and distributed systems to address the 2-layer systems we have designed (DP and PD systems). We further propose a novel partitioning policy, inspired by the streaming policies, to tackle both the heterogeneity and the scale of GPU-enabled distributed systems.

Heterogeneous systems. A lot of work has been recently dedicated to the efficient use of heterogeneous, CPU+GPU systems [43]–[45]. Most of this work focuses on workload partitioning - static or dynamic - between the different processing units in the system. In this work, we draw inspiration from static workload partitioning, which uses an estimation of the relative compute capabilities of the processing units - CPUs and GPUs - to compute an efficient partitioning before runtime. We adapt and extend the state-of-the-art profiling-based approach from [18] to a method that determines the right fraction of edges per processing unit. This fraction is an important parameter for our graph partitioner.

VI. Conclusion

In this work, we bridge the gap between large-scale systems and accelerated systems for graph processing by designing three families of distributed heterogeneous systems. Each family focuses on a different partitioning architecture—Distributed-Parallel, Parallel-Distributed, or Combined. We combine promising policies for the DP and PD systems, and propose a new policy for the C sytems. To tackle heterogeneity while partitioning, we adapt and extend a profiling-based method to compute the workload fractions for the CPU(s) and the GPU(s).

For the implementation of systems, we address several technical challenges, such as implementing communication of CPUs and GPUs on multiple machines and building partitions independently on each processing unit.

To evaluate performance, we conduct experiments for all three families of systems, using different partitioning policies. Our results demonstrate the feasibility of distributed heterogeneous systems for graph processing. Performance-wise, the systems are competitive with the state-of-the-art.

Acknowledgment

This work is supported by the Veni project @large (#11881) and Graphitti (#12480), by the Dutch KIEM project KIESA, by Commit and the Commit project Commissioner, by the NSFC No.61379146, No.61272483, No.61272056, and by the Fund No.JC13-06-03.

References

[1] "Facebook," https://www.facebook.com/.
[2] "LinkedIn," https://gb.linkedin.com/.
[3] "SMEs," http://ec.europa.eu/growth/smes/.
[4] "Wikimedia," https://wikimediafoundation.org/wiki/FAQ/en/.
[5] "Friendster," http://www.friendster.com/.
[6] "XFire," http://xfire.com/.
[7] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-scale Graph Processing," in *SIGMOD*, 2010.
[8] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph Processing in a Distributed Dataflow Framework," in *OSDI*, 2014.
[9] S. Hong, S. Depner, T. Manhardt, J. V. D. Lugt, M. Verstraaten, and H. Chafi, "PGX.D: A Fast Distributed Graph Processing Engine And Lessons From It," in *SuperComputing*, 2015.
[10] P. Harish and P. Narayanan, "Accelerating Large Graph Algorithms on the GPU Using CUDA," in *HiPC*, 2007.
[11] J. Zhong and B. He, "Medusa: Simplified Graph Processing on GPUs," *TPDS*, 2013.
[12] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A High-Performance Graph Processing Library on the GPU," in *PPoPP*, 2015.
[13] A. Gharaibeh, E. Santos-Neto, L. B. Costa, and M. Ripeanu, "Efficient Large-Scale Graph Processing on Hybrid CPU and GPU Systems," *TOPC*, 2013.
[14] Z. Fu, M. Personick, and B. Thompson, "MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs," in *GRADES*, 2014.
[15] Z. Fu, H. Dasari, B. Bebee, M. Berzins, and B. Thompson, "Parallel Breadth First Search on GPU Clusters," in *IEEE Big Data*, 2014.
[16] Y. Guo, S. Hong, H. Chafi, A. Iosup, and D. Epema, "Modeling, Analysis, and Experimental Comparison of Streaming Graph-Partitioning Policies: A Technical Report," Delft University of Technology, Tech. Rep. PDS-2015-002, 2015.
[17] N. Xu, B. Cui, L. Chen, Z. Huang, and Y. Shao, "Heterogeneous environment aware streaming graph partitioning," *TKDE*, 2015.
[18] J. Shen, A. L. Varbanescu, and H. Sips, "Look Before You Leap: Using the Right Hardware Resources to Accelerate Applications," in *HPCC*, 2014.
[19] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping," in *Micro*, 2009.
[20] "Open MPI," http://www.open-mpi.org/.
[21] "GPUDirect," https://developer.nvidia.com/gpudirect/.
[22] Y. Guo, A. L. Varbanescu, A. Iosup, and D. Epema, "An Empirical Performance Evaluation of GPU-Enabled Graph-Processing Systems," in *CCGrid*, 2015.
[23] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, 1990.
[24] R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994.
[25] T. Zhang, J. Zhang, W. Shu, M.-Y. Wu, and X. Liang, "Efficient Graph Computation on Hybrid CPU and GPU Systems," *J. Supercomput.*, 2015.
[26] "Giraph," http://giraph.apache.org/.
[27] S. Salihoglu and J. Widom, "GPS: A Graph Processing System," Tech. Rep., 2012.
[28] G. Karypis and V. Kumar, "Multilevel Graph Partitioning Schemes," in *ICPP*, 1995, pp. 113–122.
[29] I. Stanton and G. Kliot, "Streaming Graph Partitioning for Large Distributed Graphs," in *ACM SIGKDD*, 2012.
[30] T. White, *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.
[31] "Family of Graph and Hypergraph Partitioning Software," http://glaros.dtc.umn.edu/gkhome/views/metis.
[32] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "FENNEL: Streaming graph partitioning for massive scale graphs," in *WSDM*, 2014.
[33] S. Sallinen, D. Borges, A. Gharaibeh, and M. Ripeanu, "Exploring Hybrid Hardware and Data Placement Strategies for the Graph 500 Challenge," in *SuperComputing*, 2014.
[34] S. Madougou, A. L. Varbanescu, C. de Laat, and R. van Nieuwpoort, "An Empirical Evaluation of GPGPU Performance Models," in *Euro-Par*, 2014.
[35] "Graph500," http://www.graph500.org/.
[36] "DAS4," http://www.cs.vu.nl/das4/.
[37] Y. Guo, et al., "How Well do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis: Extended Report," Delft University of Technology, Tech. Rep. PDS-2013-004, 2013.
[38] "SNAP," http://snap.stanford.edu/index.html/.
[39] Y. Guo and A. Iosup, "The Game Trace Archive," in *NetGames*, 2012.
[40] "LDBC," http://ldbcouncil.org/.
[41] "Intel TBB," https://www.threadingbuildingblocks.org/.
[42] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," in *OSDI*, 2012.
[43] J. Shen, A. L. Varbanescu, H. Sips, M. Arntzen, and D. Simons, "Glinda: A Framework for Accelerating Imbalanced Applications on Heterogeneous Platforms," in *CF*, 2013.
[44] F. Song, S. Tomov, and J. Dongarra, "Enabling and Scaling Matrix Computations on Heterogeneous Multi-core and multi-GPU Systems," in *ICS*, 2012.
[45] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, "Multi-GPU and multi-CPU Parallelization for Interactive Physics Simulations," in *Euro-Par*, 2010.