2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-2235-2/20/$31.00 ©2021 IEEE | DOI: 10.1109/HPCA51647.2021.00039

DepGraph: A Dependency-Driven Accelerator for Efficient Iterative Graph Processing

Yu Zhang[†], Xiaofei Liao[†], Hai Jin[†], Ligang He[§], Bingsheng He[‡], Haikun Liu[†], Lin Gu[†]
[†]National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, China
[§]Department of Computer Science, University of Warwick, United Kingdom
[‡]National University of Singapore, Singapore
{zhyu, xfliao, hjin, hkliu, lingu}@hust.edu.cn Ligang.He@warwick.ac.uk hebs@comp.nus.edu.sg

***Abstract*—Many graph processing systems have been recently developed for many-core processors. However, for iterative graph processing, due to the dependencies between vertices' states, the propagations of new states of vertices are inherently conducted along graph paths sequentially and are also dependent on each other. Despite the years' research effort, existing solutions still severely underutilize many-core processors to quickly propagate the new states of vertices, suffering from slow convergence speed. In this paper, we propose a dependency-driven programmable accelerator, *DepGraph*, which couples with the core architecture of the many-core processor and can fundamentally alleviate the challenge of dependencies for faster state propagation. Specifically, we propose an effective dependency-driven asynchronous execution approach into novel microarchitecture designs for faster state propagations. DepGraph prefetches the vertices for the core on-the-fly along the dependency chains between their states and the active vertices' new states, aiming to effectively accelerate the propagations of the active vertices' new states and also ensure better data locality. Through transforming the dependency chains along the frequently-used paths into direct ones at runtime and maintaining these calculated direct dependencies as a set of fast shortcuts, called *hub index*, DepGraph further accelerates most state propagations. Also, many propagations do not need to wait for the completion of other propagations, which enables more propagations to be effectively conducted along the paths with higher degree of parallelism. The experimental results show that for iterative graph processing on a simulated 64-core processor, a cutting-edge software graph processing system can achieve 5.0−22.7 times speedup after integrating with our DepGraph while incurring only 0.6% area cost. In comparison with three state-of-the-art hardware solutions, i.e., HATS, Minnow, and PHI, DepGraph improves the performance by up to 3.0−14.2, 2.2−5.8, and 2.4−10.1 times, respectively.**

I. Introduction

Because of the increasing importance of graph applications, many iterative algorithms [6], [22], [27], [39], [46], [65], [67], [74] are recently designed to process large graphs iteratively until convergence for various applications, such as pinpointing influencers in social graphs, uncovering latent relationships, and launching targeted advertising. To provide timely results for these graph applications, many solutions are proposed from the perspective of both software [3], [11], [16], [23], [24], [26], [33], [51], [53] and hardware [7], [12], [38], [76]. Although several domain specific accelerators, e.g., Graphicionado [17], GraphR [49], and GraphDynS [56], are developed and can obtain high performance for graph processing, they sacrifice the programmability and low entry cost provided by general-purpose processors. Thus, some hardware accelerators, e.g., HATS [35], Minnow [59], and PHI [36], are recently designed to be integrated into the general-purpose many-core processor to enable its cores to support high-performance graph processing by ensuring faster convergence and better data locality, etc.

Despite previous research efforts from both software and hardware directions, we find that the sequential nature of vertex state propagation in iterative graph processing severely underutilizes the parallelism potential of many-core processors. In iterative graph algorithms, there are often long dependency chains between vertices' states because of the dependency between each vertex and its direct neighbor (i.e., each vertex's new state can work on its direct neighbor only when this neighbor is updated after receiving this new state). In order to affect an indirect neighbor, each vertex's new state needs to iteratively pass through the vertices on a multi-hop path in sequence. Consequently, the new state of a vertex needs many rounds to be propagated along the multi-hop graph path to reach its indirect neighbors, where a high synchronization cost is also required to propagate this new vertex state across different cores along the dependency chain. Many vertices are inactive until new states of their neighbors reach them. Besides, stale state of a vertex may be read by its neighbors (there could be many of them) to conduct unnecessary vertex state updates, which wastes much time of the cores. As a result, the many-core processor conducts the useful updates towards the convergence of iterative graph processing with a low degree of parallelism.

Challenges come with opportunities. We observe that fewer updates are needed when the vertices are handled asynchronously (i.e., the newly calculated vertex state in a round is allowed to be used by other vertices to update their states in the same round [30], [64]) along the dependency chain, and most vertex state propagations occur only on a small set of graph paths, which are the paths between the high-degree vertices. It is because many real-world graphs follow the power-law property [13], [14]. Namely, a small number of high-degree vertices are connected with most other vertices.

978-1-6654-2235-2/21/$31.00 ©2021 IEEE
DOI 10.1109/HPCA51647.2021.00039

Accum(*value1*, *value2*):
return *value1*+*value2*
//Processing the edge <*vj*, *vi*> to get the influence (which is denoted by *f*(*vj*,*vi*)(*sj*)) of the state of *vj* (i.e., *sj*) on that of *vi* (i.e., *si*)
EdgeCompute(*vj*, *vi*):
return d×*vj*.Δ*value*/*vj*.*OutDegree*

(a) incremental pagerank

Accum(*value1*, *value2*):
return *value1*+*value2*
//Processing the edge <*vj*, *vi*> to get the influence (which is denoted by *f*(*vj*,*vi*)(*sj*)) of the state of *vj* (i.e., *sj*) on that of *vi* (i.e., *si*)
EdgeCompute(*vj*, *vi*):
return *vj*.Δ*value*×*vj*.*probability*

(b) adsorption

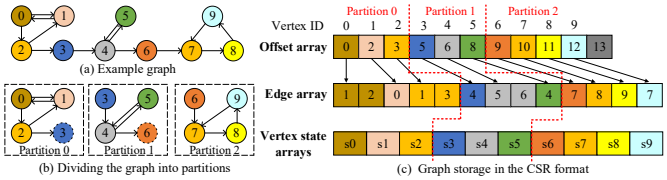
Accum(*value1*, *value2*):
return Min(*value1*, *value2*)
//Processing the edge <*vj*, *vi*> to get the influence (which is denoted by *f*(*vj*,*vi*)(*sj*)) of the state of *vj* (i.e., *sj*) on that of *vi* (i.e., *si*)
EdgeCompute(*vj*, *vi*):
return *vj*.*value*+<*vj*, *vi*>.*distance*

(c) SSSP

Accum(*value1*, *value2*):
return Max(*value1*, *value2*)
//Processing the edge <*vj*, *vi*> to get the influence (which is denoted by *f*(*vj*,*vi*)(*sj*)) of the state of *vj* (i.e., *sj*) on that of *vi* (i.e., *si*)
EdgeCompute(*vj*, *vi*):
return *vj*.*value*

(d) WCC

Fig. 1. Examples for illustrating the implementation of graph algorithms, where $EdgeCompute()$ calculates the influence of a vertex's state on its direct neighbor according to each edge, and $Accum()$ accumulates the influence of other vertices' states on a vertex



(a) Example graph (b) Dividing the graph into partitions (c) Graph storage in the CSR format

Fig. 2. Illustration of how to divide a graph and represent it using CSR

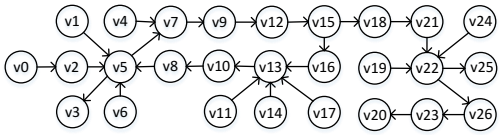


Fig. 3. Example graph to show the problems of existing solutions

Based on these observations, an effective dependency-driven programmable accelerator *DepGraph* is designed in this work, which can be integrated into the many-core processor to enable its core to speed up the iterative graph processing by residing between each core and its L2 cache.

The key idea of the DepGraph is to support an effective dependency-driven asynchronous execution approach, which embraces a novel hardware-software codesign with dependency-driven architecture on standard processor cores. Specifically, the DepGraph engine coupled with each core prefetches the vertices on-the-fly for asynchronous processing of these vertices on this core according to the dependency chain originated from the active vertex, and the vertices along different dependency chains are processed by different cores concurrently. It enables better data locality and faster state propagation along the dependency chain, while incurring smaller synchronization cost because fewer propagations occur between cores. Moreover, we identify the properties of a diverse set of graph algorithms, and develop a novel method to transform the long dependency chain between two indirect neighbouring vertices connected by a long graph path into a direct dependency (which is essentially a shortcut) between these two vertices. For each running graph algorithm, the direct dependencies are generated at runtime for high-degree vertices, and stored in a data structure called *hub index*. By such means, the propagations of the vertices' new states can reach other vertices via much fewer hops. Also, the propagations can be carried out with a higher degree of parallelism.

We have implemented DepGraph in a simulated 64-core processor and conducted extensive experiments to evaluate its effectiveness. The results show that DepGraph can reduce the total number of updates of vertices by 61.4%−82.2%, and also achieve much higher utilization of cores in conducting the propagations of useful states. As the result, DepGraph is able to achieve 5.0−22.7 times speedup with only 0.6% area overhead, compared with the cutting-edge software solutions. Compared with three state-of-the-art hardware accelerators, i.e., HATS [35], Minnow [59], and PHI [36], for many-core processors, DepGraph can improve the performance by 3.0−14.2, 2.2−5.8, and 2.4−10.1 times, respectively.

In summary, this work makes the following contributions:

- A dependency-driven asynchronous execution approach is proposed to accelerate the propagations of the active vertices' new states along the dependency chains.
- A novel method is developed to maximize the effective data parallelism for a diverse set of iterative graph algorithms through generating a set of direct dependencies (called *hub index*) for the high-degree vertices as the shortcuts between them.
- An effective programmable accelerator *DepGraph* is designed and can be integrated into the many-core processors to enable faster iterative graph processing by efficiently implementing our proposed approaches.
- Comprehensive evaluation has been carried out on a simulated 64-core processor to demonstrate the advantages of DepGraph. Experimental results show that DepGraph can achieve higher performance than existing solutions.

The rest of this paper is organized as follows: Section II explains the background and our motivations. Section III describes our approach and the first asynchronous graph processing accelerator DepGraph. Section IV evaluates the performance of the DepGraph. Section V presents the additional related work, followed by a conclusion in Section VI.

II. Background and Motivation

Graph Programming Model. Most graph processing systems use the popular *Gather-Apply-Scatter* (GAS) model [10], [14], [15], [32], [40], [42], [54], [55], [75] to express iterative graph algorithms. Figure 1 shows how to use the GAS model to implement some popular iterative graph algorithms, such as incremental pagerank [56], [64], adsorption [64], *Single Source Shortest Path* (SSSP) [56], [64], and *Weakly Connected Components* (WCC) [56], [64]. Specifically, each vertex first gathers the influence of its direct neighbors' new states via a generalized sum $Accum()$, then applies the accumulated influence to update its state, finally calculates and scatters the influence of its new state to its direct neighbor using $EdgeCompute()$. In many graph algorithms, $EdgeCompute()$ has distributive property [14], [64] over the generalized sum, and the generalized sum $Accum()$ is associative and commutative [14], [64]. The systems iteratively

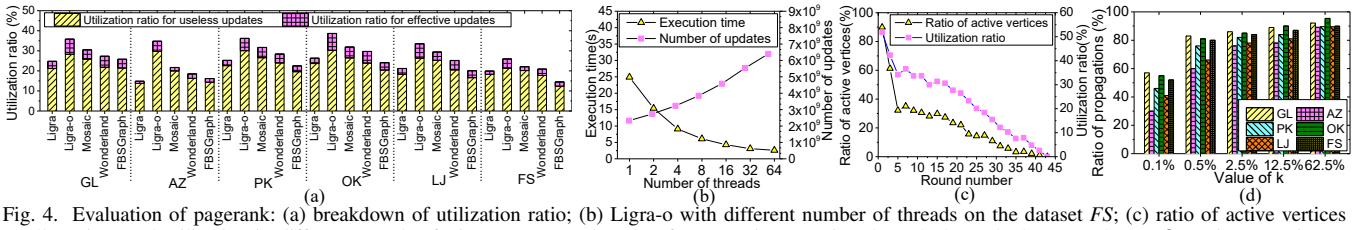


Fig. 4. Evaluation of pagerank: (a) breakdown of utilization ratio; (b) Ligra-o with different number of threads on the dataset *FS*; (c) ratio of active vertices to all vertices and utilization in different rounds of Ligra-o on *FS*; (d) ratio of propagations passing through the paths between the top k vertices on Ligra-o

process the graph round by round until all its vertices are inactive[1]. To achieve the convergence of iterative graph processing, the newly calculated state of each vertex has to be scattered to its direct neighbors in each round, and then the neighbors use the new states to make further calculations.

Graph Representation. Existing systems usually employ *Compressed Sparse Row* (CSR) [13], [32], [35], [36], [47], [56], [59], [62] to represent a graph due to its space efficiency. As shown in Figure 2, CSR has several arrays, i.e., *edge array*, *offset array*, and *vertex state arrays*. The *edge array* stores the vertices' outgoing edges, while *offset array* maintains the offset of the beginning and the end of the edges in the *edge array* for each vertex. The *edge array* also stores each edge's weight for weighted graphs. CSR uses *vertex state array*(s) to store delta state or recent state for each vertex [13], [64]. For example, for incremental pagerank, two vertex state arrays are used to hold delta state between two successive rounds and the recent state in the current round for each vertex.

Problem of Existing Approaches. The dependency chains exist between vertices' states inherently. As shown in Figure 3, there is a dependency chain from v_5 to v_{20}. Assume v_5 is the source vertex for SSSP [64] and the new state of v_5 needs to be propagated to v_{20}. The state propagation along the path from any intermediate vertex (on the dependency chain) to v_{20}, such as $v_{22}\rightarrow\cdots\rightarrow v_{20}$, becomes valid only after the new state of v_5 has reached v_{22}. Such dependency chains in the graph cause the existing systems [32], [47], [62], [64], [69] to suffer from low degree of effective parallelism in propagating the vertices' new states.

To demonstrate the phenomenon, we evaluate four cutting-edge graph processing systems, including Ligra [47], Mosaic [32], Wonderland [62], and FBSGraph [69], for processing the incremental pagerank on a server with two 28-cores 2.5 GHz Intel Xeon Platinum 8180 CPUs and 64 GB main memory. To show that the above-mentioned problem cannot be addressed by existing optimizations [36], [62], [64], Ligra-o (see Section IV), which is the optimized version of Ligra, is also evaluated. The details of the benchmarks are given in Section IV. Note that the sequential execution of incremental pagerank is also implemented as a baseline, where all vertices are handled by one thread of Ligra-o in an asynchronous manner in the depth-first search order in each round.

In the experiments, we first evaluate the number of vertex state updates (denoted by u_s) needed by the sequential execution of incremental pagerank and the number of updates (denoted by u_d) needed by each graph processing system. Then, we evaluate the resource utilization achieved by each system in processing effective updates and in processing unnecessary updates (e.g., the useless state propagation discussed above). The utilization ratio for processing effective updates can be approximated by $r_e=\frac{u_s \cdot U}{u_d}$, and the utilization ratio for useless updates is $r_u=1-r_e$, where U is the total utilization achieved by this system. Figure 4(a) shows the utilization and its breakdown for each of the above graph processing systems.

From Figure 4(a), we can observe that Ligra-o performs better than others. Taking the dataset *AZ* as an example, Ligra-o outperforms Ligra by up to 4.6× because of its optimizations. However, the thread parallelism is still underutilized for conducting effective updates in these systems, including Ligra-o. When the average length of the dependency chains is longer, r_e of these systems even becomes lower. In the datasets *GL*, *AZ*, *PK*, *OK*, *LJ*, and *FS*, the average length of the dependency chain is 4.2, 17.9, 7.7, 6.9, 8.5, and 16.1, respectively.

There are two main reasons why these systems deliver low values of r_e. First, with existing solutions, in each round, many vertices update their states based on the stale states of their neighbors, which are unnecessary updates. For example, in Figure 3, assume in a round one thread takes v_5's current state as input to calculate the new state of its direct neighbor v_7. At the same time, another thread may take v_7's current state as input to calculate the new state of its direct neighbor v_9. The update of v_9 is unnecessary because v_7 will have a new state after the first thread completes the calculation in this round. For a more distant neighbor of v_5, such as v_{22}, down the dependency chain, v_5's new state may need many rounds to be propagated to v_{22}. Before v_5's new state reaches v_{22}, the thread can only use the stale state of v_{22} (v_{26}'s direct neighbor) to update v_{26}'s state in each round, incurring many unnecessary updates. In Figure 4(a), in Ligra, Ligra-o, Mosaic, Wonderland, and FBSGraph, only 7.4%−14.5%, 14.6%−21.9%, 7.7%−16.9%, 12.1%−20.2%, and 11.3%−17.2% of the total updates are useful, respectively. Although Ligra-o has fewer unnecessary updates than other systems, there are still many updates, especially when more threads are used to handle the graph, hoping to achieve shorter execution time (Figure 4(b)).

Second, in graph processing, a vertex is inactive in a round when its state remains unchanged in previous round. Many vertices remain inactive during the execution due to the above described slow propagation of vertex's new state along dependency chains. Assume v_0 in Figure 3 is the only active vertex in a round. Then, many vertices, e.g., v_{22}, are

[1]A vertex is inactive when its state has satisfied the user-specified conditions, e.g., its state does not change in the previous round for SSSP.

inactive until v_0's new state reaches them along a chain. It causes the low degree of parallelism in existing solutions. In Figure 4(a), the total utilization delivered by Ligra-o is only 25.9%−38.6%. The utilization becomes even lower when more vertices become inactive during the execution (Figure 4(c)), where a vertex is identified as being active as its state change between two successive rounds is larger than ε=10^{-5} in the pagerank [64].

In recent years, several hardware techniques have been developed to accelerate graph processing on many-core processors. GRASP [13] uses hardware cache management to tackle cache thrashing for hot vertices. Hardware prefetchers [4], [5], [58] try to hide memory access latency. Similar to the prefetchers, HATS [35] further employs a hardware-accelerated traversal scheduler to improve data locality. For faster convergence, Minnow [59] is designed as an accelerator to privilege the processing of some high-priority vertices with low overhead. For efficient vertex state updates, several previous hardware methods [45], [60], [61] can be used, and PHI [36] further proposes a cache hierarchy for efficient commutative scatter updates in graph processing. Although these hardware solutions enable better performance of existing software systems on the many-core processors, graph processing still suffers from slow propagations of the vertices' new states due to the dependency chains.

Our Observations. We make two observations regarding vertex state propagations in our experiments. They motivate us to propose a new approach that can fully exploit the parallelism in a many-core processor for effective state propagations.

Observation one: Only the graph data along the dependency chains originated from active vertices need to be loaded for processing and fewer updates are needed when they are asynchronously handled along these chains, because vertex state is inherently propagated along dependency chain. Assume only v_{22} is active in Figure 3. Then, only the graph data along the chains v_{22}→v_{26}→v_{23}→v_{20} and v_{22}→v_{25} need to be handled. If the vertices on each chain are asynchronously updated along this chain, there is the least number of updates, which is five and the same as the number of vertices. Static software preprocessing approach [70] is proposed to generate the paths for fewer updates, however, our results show that 37.5%-77.9% of its loaded graph data and 30.2%-69.5% of its updates are unnecessary, because many vertices become inactive at run time and the vertex states also need to travel many precalculated static paths. Thus, from the perspective of effective state propagation, we should use the run-time path generation to dynamically expose the dependency chains originated from the active vertices. The vertices should also be asynchronously processed in their order on the dependency chain within each round for fewer updates.

Observation two: Most state propagations traverse a small set of paths in a graph. This observation is made through the following experiment. In the experiment, we first arrange the vertices in the descending order of their degrees, and then record the ratio of the state propagations that traverse the paths between the top k% vertices to all propagations. We investigate the ratio as the value of k changes. The results are depicted in Figure 4(d). It can be seen from this figure that more than 60% of the propagations pass through the paths between the top 0.5% vertices. This result indicates that these paths are more important from the perspective of state propagation.

III. Overview of Our Approach

Based on the observations, an accelerator *DepGraph* is designed based on our *dependency-driven asynchronous execution approach* to accelerate the propagation speed of active vertices' new states. In each round of graph processing, the DepGraph engines coupled with different cores fetch active vertices concurrently, which are then used by DepGraph as the starting vertices (called *roots*) to prefetch other vertices on-the-fly according to the dependencies between vertices. In this way, DepGraph enables the core to handle these vertices efficiently and asynchronously along the dependency chains. Specifically, DepGraph enables faster propagations of these vertices' new states along the dependency chains according to the first observation (see Section II). Because only active vertices are taken as the roots to start the computation, only the edges and vertices involving the propagations of active vertices' new states are loaded for processing, and better data locality can be achieved.

To maximize effective data parallelism, DepGraph transforms the dependency chains between high-degree vertices into direct ones at runtime based on the corresponding graph application. The set of the newly generated direct dependencies can be stored to serve as the shortcuts for most state propagations according to the second observation. With these shortcuts, the new states of most vertices can reach other vertices in fewer rounds. More importantly, by adding these shortcuts, the asynchronous state propagations along the dependency chain can be parallelized.

A. The Dependency-driven Execution Approach

This subsection first introduces two basic concepts, and then proposes our effective dependency-driven asynchronous execution approach.

1) Basic Concepts: **Definition 1** (*Hub-vertex* and *hub-path*). Let G=<V, E> represents a graph. V is the set of vertices and E is the set of edges in the graph. A path p_l=<$v_l^0, v_l^1, \cdots, v_l^{|p_l|}$> is a sequence of connected vertices. l is the path identifier, and the superscript of a vertex represents the order of the vertex on the path. $|p_l|$ is the length of p_l.

As discussed in Section II, a vertex v_h plays a more important role in state propagation when its degree $D(v_h)$ is higher. v_h is called a *hub-vertex* when $D(v_h)$ is higher than the degree threshold T. In practice, it can be hard to set T directly. Thus, the users can set a parameter λ ($0 \leq \lambda \leq 1$), which is used to specify the ratio of hub-vertices to all vertices in the graph. An appropriate value of λ is usually small due to the power-law property of real-world graphs [14]. λ can then be used to determine the value of T. We sort all vertices in the graph in the descending order of degree, and then the first λ proportion of vertices are hub-vertices, among which the degree of the

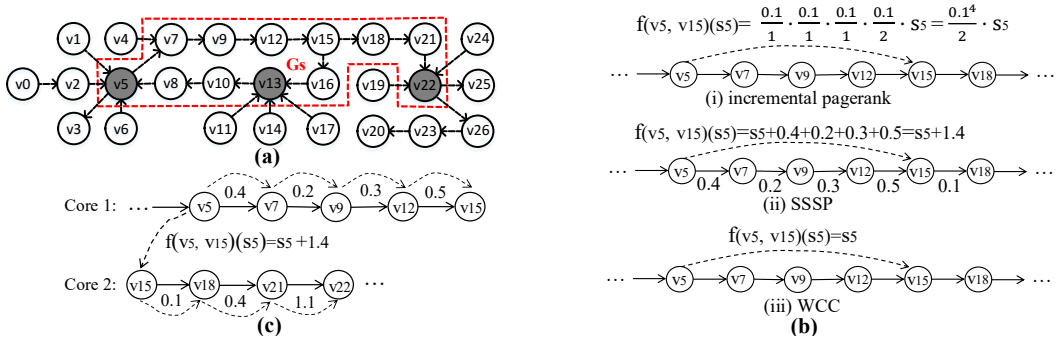


Fig. 5. An example for illustrating our dependency-driven asynchronous execution approach: (a) G_s is the core-subgraph, where v_5, v_{13}, and v_{22} are assumed to be hub-vertices; (b) the transformation of the dependency chain between the states of v_5 and v_{15} along the core-path $v_5 \rightarrow \cdots \rightarrow v_{15}$, where the damping factor of the pagerank is set to 0.1; (c) parallel state propagations in SSSP after transforming the dependency chain between s_5 and s_{15} into a direct one

last hub-vertex is the value of T. To avoid the costly sorting on all the vertices, a sampling method can be used to quickly determine the value of T. We can randomly select a proportion (denoted by β) of vertices, and then the degree of the vertex in the $(\lambda \cdot \beta \cdot n)$ position is the value of T, where n is the number of vertices in the graph. H denotes the set of hub-vertices in the graph. A path h_l=<v_l^0, v_l^1, $\cdots$, $v_l^{|h_l|}$> is called a *hub-path* if $v_l^0 \in H$ and $v_l^{|h_l|} \in H$ (i.e., both its head vertex v_l^0 and its tail vertex $v_l^{|h_l|}$ are in H).

Definition 2 (*Core-subgraph*). The core-subgraph, denoted by G_s, is an union of hub-paths between any two hub-vertices.

Each hub-vertex's new state can immediately affect another one, if we calculate a direct dependency for each hub-path in G_s. However, the hub-paths in G_s share many edges, and many direct dependencies need to be generated to get this goal, undermining its efficiency. To minimize the overhead to generate and to store the direct dependencies for better performance, the core-subgraph is further represented as a set of disjoint core-paths, i.e., any two core-paths do not have common vertices except the head vertex and/or the tail vertex on the paths. The core-subgraph can be formally represented as $G_s = \cup\ m_l$, where m_l=<v_j, $\cdots$, v_i> is a core-path, and $\forall\ m_{l'}$=<v'_j, $\cdots$, v'_i> $\in G_s \wedge \forall\ m_{l''}$=<$v''_j$, $\cdots$, v''_i>$\in G_s$ s.t. $m_{l'} \cap m_{l''} \subseteq \{v'_j, v'_i\} \cap \{v''_j, v''_i\}$. The vertex at which two core-paths intersect is called *core-vertex*. Take the graph in Figure 5(a) as an example. The core-subgraph, e.g., G_s, can be represented as four core-paths: $v_5 \rightarrow \cdots \rightarrow v_{15}$, $v_{15} \rightarrow \cdots \rightarrow v_{22}$, $v_{15} \rightarrow \cdots \rightarrow v_{13}$, and $v_{13} \rightarrow \cdots \rightarrow v_5$, where v_5, v_{13}, and v_{15} are core-vertices. Through this core-graph representation, the state of any hub-vertex, e.g., v_5, can reach any other hub-vertex, e.g., v_{22}, only through some certain core-paths, e.g., $v_5 \rightarrow \cdots \rightarrow v_{15}$ and $v_{15} \rightarrow \cdots \rightarrow v_{22}$. Once the direct dependency between the head and tail vertices of each core-path is generated, any hub-vertex can quickly get the influence of any other hub-vertex's new state only using these calculated direct dependencies (see Section III-A2).

2) Dependency-driven Asynchronous Execution: In this execution model, in each round of graph processing, each core uses the active vertices in this current round as the starting vertices to retrieve and asynchronously process other vertices in sequence down the disjoint paths, while the vertices on different paths are handled concurrently over cores whenever possible (to be introduced later). Then, the vertices' new states can be propagated to other vertices along the path in the same round while a good degree of parallelism is achieved by processing the vertices on different paths in parallel. Take the path $v_{22} \rightarrow v_{26} \rightarrow v_{23} \rightarrow v_{20}$ in Figure 5(a) as an example. With this method, the states of all vertices, e.g., v_{20}, can be updated based on the new states of their precursors, e.g., v_{23}, in sequence along the path between v_{22} and v_{20} in one round. Moreover, since the vertices on the same dependency chain are handled by the same core, all state propagations on this chain are conducted on a single core, which reduces the memory access cost and the synchronization cost.

There may be the dependencies between different disjoint paths. For example, in Figure 5(a), in order to propagate the new state of v_5 to v_{22} along the path between v_{15} and v_{22}, this propagation is valid only after the new state of v_5 has been propagated to v_{15} along the path between v_5 and v_{15}. In order to maximize the parallelization of state propagations along different paths, it creates a direct dependency between the head vertex and the tail vertex on each core-path at runtime. The influence of the direct dependency is calculated based on the specific iterative graph application implementation (such as SSSP and PageRank in Figure 1 using different formulae to quantify the influence). Such a set of direct dependencies are stored in an array called *hub index*, in which each entry represents a generated direct dependency. This hub index serves as the shortcuts to enable faster propagations (i.e., in fewer rounds) of the new states of hub-vertices.

For example, in Figure 5(a), a direct dependency will be created between v_5 and v_{15} since the path between v_5 and v_{15} is a core-path. In Figure 5(b), the influence of v_5 on v_{15}, denoted by $f_{(v_5,v_{15})}(s_5)$, can be qualified based on the specific graph algorithms. $f_{(v_5,v_{15})}(s_5)$=$\frac{0.1^4}{2} \cdot s_5$ if the incremental pagerank [56], [64] is used. If the SSSP algorithm [56], [64] is used, $f_{(v_5,v_{15})}(s_5)$=s_5+1.4. $f_{(v_5,v_{15})}(s_5)$=s_5 if WCC [56], [64] is used. With this dependency transformation, v_{15} can obtain the influence of v_5's new state immediately via the generated direct dependency. It also indicates that the new states of many vertices, e.g., v_{10}, can reach v_{15} more quickly by first reaching v_5 and then v_{15} through the direct dependency between v_5 and v_{15}. Then, many vertices, e.g., v_{15}, can more quickly accumulate the influences from different paths to converge.

TABLE I
EXAMPLES OF ALGORITHMS WITH THE PROPERTIES

Generalized sum	Iterative graph algorithms
sum	pagerank, adsorption, katz metric, SimRank, HITS, belief propagation
min or max	SSSP, WCC, k-core, SSWP

Another important benefit of introducing the direct dependency is that the parallelization of state propagations along different paths can be maximized. For example, in Figure 5(c), the propagation of v_5's new state from v_{15} to other vertices down the dependency chain (e.g., v_{18}) does not have to wait until all vertices on the path from v_5 to v_{15} have been updated. v_5's new state can reach v_{15} via the direct dependency. Assume we are running SSSP. Then, the influence of v_5's new state on v_{15} can be calculated via $f_{(v_5,v_{15})}(s_5)$=s_5+1.4. Assume that the vertices on the path between v_5 and v_{15} and the vertices on the path between v_{15} and v_{22} are scheduled to run on two cores (see Figure 5(c)). Then, updating the successors of v_{15} can be conducted on core 2 in parallel with updating the predecessors of v_{15} (i.e., v_7, v_9 and v_{12}) on core 1. Thus, a higher degree of parallelism is achieved to accelerate the asynchronous propagation of new states of vertices, e.g., v_5.

3) Applicability of Dependency Transformation: For correctness, the graph algorithms should have two properties.

- *Property 1:* The graph algorithm can be expressed using the GAS model as shown in Figure 1.
- *Property 2:* Edge processing function $EdgeCompute()$ is a linear expression, which is usually shown to be a multiplication or an addition [9], [49], [56], [64]. More fundamentally, many graph algorithms can be represented as a variant of sparse matrix and vector computation [9], [49], and these algorithms satisfy this linearity relation.

Previous works [14], [32], [49], [50], [56], [64] have shown that most iterative graph algorithms have such two properties. Table I lists some examples, which include the popular pagerank [47], [49], [56], [64], adsorption [64], SSSP [49], [56], [64], WCC [56], [64], as well as other examples such as katz metric [64], SimRank [64], HITS [64], belief propagation [59], k-core [70], and *single-source widest path* (SSWP) [56]. For other graph algorithms without such two properties, such as triangle counting [43] and clique detection [43], the users need to disable the dependency transformation functionality in DepGraph. Note that the second property indicates that the dependency between the states of two connected vertices is a linear relationship. It means that the direct dependency $f_{(v_j,v_i)}(s_j)$ created between two vertices v_j and v_i is also linear relationship, i.e., $f_{(v_j,v_i)}(s_j)$=μ·s_j+ξ, where μ and ξ are the constants. For example, in Figure 5(b), for the direct dependency between v_5 and v_{15}, μ=$\frac{0.1^4}{2}$, ξ=0 for pagerank, μ=1, ξ=1.4 for SSSP, and μ=1, ξ=0 for WCC. This property allows efficient calculation and storage of the direct dependency between the vertices' states (presented in Section III-B2).

4) Correctness of Transformation: This subsection demonstrates that the dependency transformation is correct for a diverse set of iterative graph algorithms.

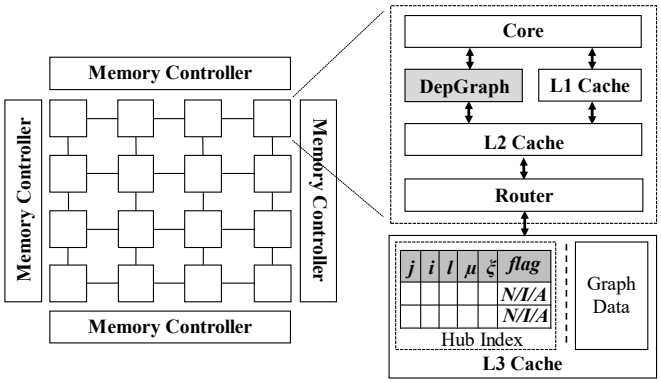


Fig. 6. System architecture

Theorem 1: When the iterative graph algorithms with the two properties described in Section III-A3 process the graph with the dependency transformation, they yield the same results as the original ones without dependency transformation.

Proof 1: Given a graph G=(V, E), and each vertex $v_i \in V$ maintains a state s_i, which is iteratively updated based on the states of its direct neighbors. Let s_i^R denote the state of v_i after handling the graph asynchronously for R ($R \geq 0$) rounds. Let $x \oplus y$ denote the operation of $Accum(x, y)$. Because each vertex's update is a generalized sum $\oplus$ over the states of its direct neighbors, we have:

$$s_i^R = s_i^0 \oplus \big(\sum_{v_t \in N(i)} \oplus f_{(v_t,v_i)}(s_t^{k_t}) \big) \tag{1}$$

where

$$\sum_{v_t \in N(i)} \oplus f_{(v_t,v_i)}(s_t^{k_t}) = f_{(v_{i_1},v_i)}(s_{i_1}^{k_{i_1}}) \oplus \cdots \oplus f_{(v_{i_m},v_i)}(s_{i_m}^{k_{i_m}}) \tag{2}$$

$N(i)$ is the neighbor set of v_i, $0 \leq k_t < R$, $v_{i_1}, \cdots, v_{i_m} \in N(i)$. Because $f_{(v_j,v_i)}(x)$ has distributive property over $\oplus$, and the generalized sum $\oplus$ has associative and commutative properties, after applying Equation (1) iteratively for the related vertices, we can have:

$$s_i^R = s_i^0 \oplus \sum_{<v_j,v_{j_1},\cdots,v_{j_m},v_i> \in P(j,i,R)} \oplus\, c_{<v_j,v_{j_1},\cdots,v_{j_m},v_i>}(s_j^0) \tag{3}$$

where

$$c_{<v_j,v_{j_1},\cdots,v_{j_m},v_i>}(s_j^0) = f_{(v_{j_m},v_i)} \circ \cdots \circ f_{(v_j,v_{j_1})}(s_j^0) \tag{4}$$

$v_j \in N(i)^R$ and $N(i)^R$ is the set of vertices which can propagate their states to v_i within R rounds. $P(j, i, R)$ is the set of paths which can be used to propagate the state from v_j to v_i within the R rounds. $f_{(v_k,v_i)} \circ f_{(v_j,v_k)}(s_j)$ denotes $f_{(v_k,v_i)}\big(f_{(v_j,v_k)}(s_j)\big)$. From Equation (4), we can find that the dependency chain between the states of v_j and v_i along each path $v_j \to v_{j_1} \to \cdots \to v_{j_m} \to v_i$ can be transformed into a direct dependency in advance. Then, v_j's new state can affect v_i immediately, while the influence of original state propagation along the dependency chain remains the same. Thus, the dependency transformation does not violate the correctness.

B. DepGraph Architecture

The above approach suffers from high runtime cost because it needs to fetch the edges on-the-fly along dependency chains and maintain the hub index (see Section IV-A). To efficiently implement the approach, an accelerator *DepGraph* is designed.

As shown in Figure 6, each DepGraph engine is coupled with a core of the many-core processor and accesses the memory subsystem via the L2 cache. Namely, the DepGraph engine issues the instructions to access the data from the L2 cache. If the data does not exist in the L2 cache, the data is retrieved from the L3 cache or the main memory.

DepGraph is a hardware-software co-design, which relies on the existing software graph processing system [47]. The existing software graph processing system running on the core preprocesses the graph (e.g., dividing the graph into partitions and assigning them to the cores for parallel processing), calls the APIs provided by DepGraph to initialize the DepGraph engine, then processes graph edges/vertices, balances workload (e.g., through the work stealing scheme [8]), etc. When dividing the graph in the preprocessing stage, the software system also finds the hub-vertices and core-vertices in the graph partitions (based on the user-specified parameters λ and β) as well as the initial active vertices, which can be achieved by traversing the graph only once, and passes the information of these vertices to the DepGraph engine.

After obtaining the active vertices from the graph processing system running above, the DepGraph engine of each core uses the active vertices in the partition assigned to this core as the starting vertices to prefetch other vertices (and edges) along the dependency chain. DepGraph then passes the fetched vertices and edges to the graph processing system, which in turns handles the vertices and edges using the user-defined functions $Accum()$ and $EdgeCompute()$ in an asynchronous way.

With the obtained hub-vertices and core-vertices, the DepGraph generates and maintains a hub index at runtime to accelerate the state propagations to pass through core-paths and maximize effective data parallelism of graph processing. Specifically, the DepGraph engine in each core generates the entries in the hub index for the received hub-vertices and core-vertices, and writes this partial hub index to the L2 cache, which is transferred to the main memory. The whole hub index is maintained in the memory by all DepGraph engines across different cores and reused by them for efficient graph processing. Because the hub index will not change for static graphs once generated, there is no cache inconsistency issue when the entries in the hub index are replicated in the cache. The evicted entries of the hub-index are just directly dropped.

1) Microarchitecture of DepGraph: Figure 7 shows the microarchitecture of DepGraph. Note that DepGraph performs the processing of each edge as an atomic operation, because the basic processing unit in the existing software systems [32], [47] is an edge. It means that the edge prefetched for the paired core can be handled by any thread on the core. The acceleration kernel does not need to be pinned and also allows the memory to be paged out. When the graph processing system running on a core needs to handle a partition, it first configures the DepGraph engine as shown in step ① of Figure 7, which includes passing the information of the hub-vertices and core-vertices found in the partition to DepGraph.

The graph processing system running on each core contiguously places and maintains the active vertices of its local partition in a local circular queue in the memory. The *Hardware Dependency-aware Traveler Logic* (HDTL) is designed in the DepGraph engine. HDTL takes an active vertex from the local circular queue, and starts from this vertex to prefetch the edges (as well as the related vertices) in the graph partition along the graph path (see step ② in Figure 7). The prefetched edges are stored in an *FIFO Edge Buffer*. DepGraph provides an ISA instruction *DEP_FETCH_EDGE* to obtain these prefetched edges from the *FIFO Edge Buffer*. This ISA instruction is exposed to the graph processing systems through a low-level API named $DEP\_fetch\_edge()$. This process repeats until the local circular queue is empty, which means that there are no more active vertices in the partition. Then, the next partition allocated to this core is handled. The iterative graph algorithm converges when there are not active vertices in all partitions.

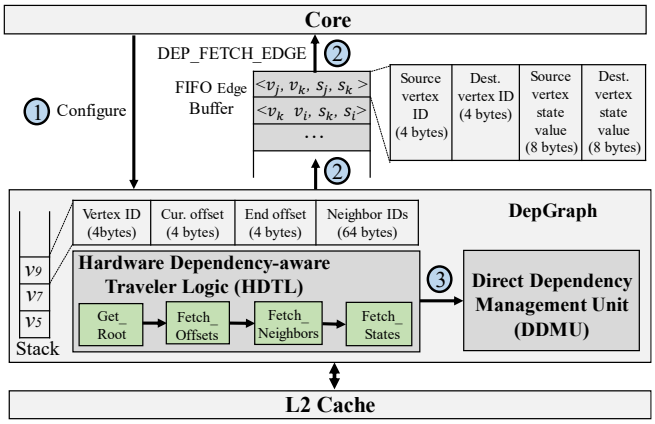


Fig. 7. Microarchitecture of DepGraph

Moreover, a lightweight hardware unit called *Direct Dependency Management Unit* (DDMU) is designed in DepGraph to generate and maintain the hub index efficiently at runtime. When an active vertex is retrieved from the local circular queue, as shown in step ③, DDMU checks if the direct dependency related to this vertex exists in the hub index. If so, the direct dependency is used as the shortcut for state propagation. Otherwise, DDMU calculates the direct dependency related to this vertex and stores it in the hub index for future reuse.

2) Implementation Details: **Initialization.** Before processing a partition on a core, the thread running on the core calls a low-level API $DEP\_configure()$ provided by DepGraph to configure DepGraph, which conveys the information to the memory-mapped registers accessible to the DepGraph engine (as the way configuring a DMA engine). Because most graph processing systems represent the graph using the popular CSR format, as described in Figure 8, the default information regarding the graph representation conveyed through the $DEP\_configure()$ includes: (a) the sizes and base addresses of *offset array*, *edge array*, and *vertex state arrays*; (b) the IDs of the beginning vertex and the end vertex of the partition G^m assigned to this core. Note that the reconfigurable logic implementation of DepGraph allows the implementation of other graph formats without extra overhead. The invocation of $DEP\_configure()$ also passes the size and the base address of an in-memory bitmap storing the set $H^{m''} = H^m \cup H^{m'}$

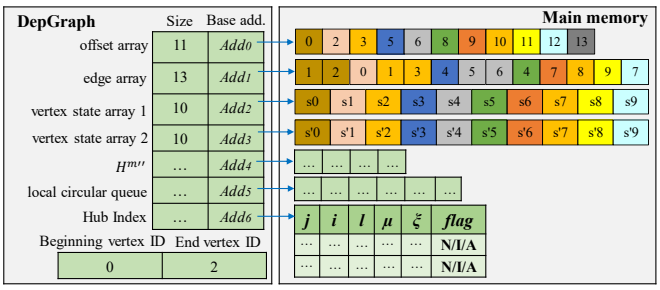


Fig. 8. The arrays and the corresponding functional units of DepGraph

(which is pre-calculated by software system as presented before) to DepGraph. H^m is the set of hub-vertices and core-vertices in the partition G^m, and $H^{m'}$ is the set of G^m's boundary vertices connected to hub-vertices or core-vertices of the remaining graph. Then, the size and the base address of the local circular queue of this core is also passed to DepGraph. The active vertices in G^m are inserted into the local circular queue of this core, starting from which the DepGraph engine fetches other vertices (and edges) along the dependency chain.

Dependency-aware Edge Prefetching. In each round of graph processing, for each core, the HDTL of its DepGraph engine prefetches the edges of its partition G^m on-the-fly via the depth-first search, starting from the active vertex (called the root vertex of the search) taken from the local circular queue. The prefetching continues until all related edges are visited. As shown in Figure 7, a fixed-depth stack is used to record the information for prefetching. Each entry in the stack contains the following information: a) the ID of visited vertex in a traversal, b) the current offset/end offset of the unvisited edges of this vertex, and c) a cache line of unvisited neighbors' IDs. The edges are fetched along the path by recursively adding the visited edges of each traversal into the *FIFO Edge Buffer*.

Each traversal repeats the following four stages (shown in Figure 7). If the stack is empty, an active vertex is taken from the local circular queue and pushed into the stack, which is the Get_Root stage. Then, the $Fetch\_Offsets$ stage fetches the beginning/end offsets of the edges of the topmost vertex (e.g., v_9) in the stack from the *offset array*. In the $Fetch\_Neighbors$ stage, the IDs of this vertex's unvisited neighbors are retrieved from the *edge array* based on its unvisited edges, and one of these neighbors (e.g., v_{12}) is pushed into the stack as a new topmost vertex. The $Fetch\_States$ stage fetches the states of the relevant vertices from *vertex state arrays*. HDTL runs the above four stages as a pipeline, which outputs a fetched edge (e.g., $<v_9, v_{12}>$) as well as the states of the pair of vertices of this edge into the *FIFO Edge Buffer* each time. At the end of the pipeline, whenever a vertex belonging to $H^{m''}$ has been fetched or no other unvisited vertices of G^m can be fetched from the current root vertex, the topmost vertex is popped out of the stack and a new traversal begins so as to prefetch the edges along a new path based on the stack's current state (i.e., the execution continues with the next neighbor of the new topmost vertex in the stack). The last prefetched vertex along each path is also inserted into the local circular queue to be used as a new root vertex. Note that a core-path is found when a path starts from a vertex in $H^{m''}$ and then reaches another vertex which is also in $H^{m''}$.

Generating the Hub Index. Whenever a core-path m_l between vertices v_j and v_i is identified in the above stage, the dependency chain between the states (i.e., s_j and s_i) of its head vertex v_j and its tail vertex v_i along the path is transformed into a direct one according to the graph algorithm run by the graph processing system, if such a direct dependency does not exist in the hub index. Because the dependency between any two vertices is linear relationship, i.e., $f_{(v_j,v_i)}(s_j) = \mu \cdot s_j + \xi$, for the graph algorithms supported by DepGraph [49], [56], [64] (see Section III-A3), the direct dependency can be calculated based on the previous values of s_j and s_i at two successive rounds of graph processing. In detail, after processing the core-path m_l, *DDMU* detects the flag of the corresponding entry in *hub index*. If the flag is **N**, the current values of s_j and s_i are stored as the values of μ and ξ of the linear function, respectively, in this entry, and the flag of this entry is then set to **I**. If the flag is **I**, it means this entry has stored the previous values of s_j and s_i, denoted by s'_j and s'_i. Under such a circumstance, the values of μ and ξ can be determined by solving two linear equations, i.e., $s'_i = \mu \cdot s'_j + \xi$ and $s_i = \mu \cdot s_j + \xi$, whose solution is $\mu = \frac{s'_i - s_i}{s'_j - s_j}$ and $\xi = s'_i - \mu \cdot s'_j$. The calculated values of μ and ξ are stored in the corresponding entry of *hub index*, and the flag is set to **A** to indicate this entry is available and can be used as a shortcut.

Maintaining the Hub Index. Each dependency is represented by only two parameters, i.e., μ and ξ of the linear function. Thus, the hub index is implemented as an in-memory key-value table. Each entry of this table stores $<j, i, l, \mu, \xi>$, which represents the direct dependency between the states of the head vertex v_j and the tail vertex v_i of a core-path m_l. The direct dependencies between the same pair of vertices but along different core-paths are successively stored with the ID (e.g., l of Figure 6) of the core-path as the identification. Because we have ensured that any two core-paths do not share common edges, the ID of its second vertex (e.g., v_7 as shown in Figure 5(a)) is used as the ID of the core-path between v_5 and v_{15}. Note that the replacement algorithm, e.g., *Dynamic Re-Reference Interval Prediction* (DRRIP [18]) or *Least Recently Used* (LRU), is used for the L3 cache to manage its data by default. The hub index will be frequently accessed because the core-paths need to be repeatedly traversed by most state propagations. Thus, the hub index is very likely to be maintained in the L3 cache for fast reuse.

Faster Propagation Based on Hub Index. When DepGraph fetches the edges starting from a root vertex, the following measures are used for effective parallel propagations. If the root vertex (e.g., v_5 in Figure 5(a)) is in $H^{m''}$, DepGraph tries to retrieve μ and ξ for all core-paths originated from this vertex from the *hub index*. *DDMU* in DepGraph tries to access the information first from the L2/L3 cache and then from the main memory. Note that an in-memory hash table is used to help this access. Each entry of this hash table is *<vertex ID, beginning_offset, end_offset>*. This hash table

has $|H|/\omega$ number of entries, where $|H|$ is the number of hub-/core- vertices and ω is set to 0.75 by default [41]. It first uses the ID of root vertex, e.g., v_5, to obtain the offsets (i.e., *beginning_offset* and *end_offset*) of the beginning and end entries in the hub index for v_5 from the hash table, and then uses the offsets to read the entries from the hub index for v_5.

Next, for each core-path originating from the root vertex (e.g., the core-path between v_5 and v_{15}), DepGraph directly calculates the influence of v_5 on v_{15} (i.e., $f_{(v_5,v_{15})}(s_5)$) using the retrieved μ and ξ, aiming to immediately affect v_{15}. The tail vertex (i.e., v_{15} in this case) then can be inserted into the local circular queues of all cores that own a partition with v_{15} being marked as the active vertex. For this goal, like HATS [35], it only needs to simply check the partition boundaries by comparing the ID of v_{15} with the IDs of the beginning and the end vertex of the partition assigned to each core. Using this way, the local circular queue of a core without v_{15}'s graph data may be inserted with v_{15}. It does not matter because no edges originated from v_{15} will be prefetched from its partition. By doing so, the influence propagations of v_5's new state can be conducted concurrently on these cores to quickly affect many other vertices along different graph paths to fully exploit the parallelism.

As shown in Figure 5(c), for each core-path (e.g., the one between v_5 and v_{15}), its tail vertex, i.e., v_{15} in the example, will also eventually receive the influence, i.e., $f_{(v_5,v_{15})}(s_5)$, of the new state of the head vertex, i.e., v_5, along the path. Thus, v_{15} will receive $f_{(v_5,v_{15})}(s_5)$ twice. To ensure the correctness, it needs to ensure that only one copy of $f_{(v_5,v_{15})}(s_5)$ finally affects v_{15} as well as its followers, e.g., v_{18}. Specifically, if the generalized sum of the graph algorithm (e.g., SSSP or WCC) is min or max as listed in Table I, the result of the generalized sum of two copies of $f_{(v_5,v_{15})}(s_5)$ is still $f_{(v_5,v_{15})}(s_5)$. Thus, it does not induce incorrect results although two copies of $f_{(v_5,v_{15})}(s_5)$ are received. However, when the generalized sum of the graph algorithm (e.g., pagerank or adsorption) is sum, the result of the generalized sum of two copies of $f_{(v_5,v_{15})}(s_5)$ will not be $f_{(v_5,v_{15})}(s_5)$, and the thread then needs to reset s_{15} by $s_{15} \leftarrow s_{15} - f_{(v_5,v_{15})}(s_5)$, i.e., the state of v_{15} takes away the influence of v_5. To reset the vertex state transparently via the above method, DepGraph inserts a fictitious edge $<-1, v_{15}, NULL, f_{(v_5,v_{15})}(s_5)>$ (this edge has a fake vertex ID, i.e., -1) into the *FIFO Edge Buffer* at the end of the prefetched core-path. The API $DEP\_fetch\_edge()$ provided by DepGraph resets the state of the tail vertex of the core-path when such a fictitious edge is detected. This edge is deleted after state reset and is not returned for further execution.

To transparently identify whether the generalized sum is sum or min/max, DepGraph automatically detects the results of the function $Accum(x, y)$ by inputting x=1 and y=1 at the initialization stage. If the result is 2, it is sum. If the returned result is still 1, the generalized sum is min/max. Otherwise, it returns $error$ to notify that this graph algorithm is not supported by dependency transformation. Under such circumstance, users need to disable the dependency transformation functionality to ensure correct results for this algorithm.

TABLE II
CONFIGURATION OF THE SIMULATED SYSTEM

Cores	64 cores, x86-64 ISA, 2.5 GHz, Intel Skylake-like OOO [44]
L1 Instruction Cache	32 KB per-core, 4-way set-associative, 3-cycle latency
L1 Data Cache	32 KB per-core, 8-way set-associative, 4-cycle latency
L2 Cache	256 KB per-core, core-private, 8-way set-associative, 7-cycle latency
L3 Cache	128 MB, shared, 32 banks, 16-way hashed set-associative, 27-cycle bank latency, DRRIP replacement [18]
Global NoC	8×8 mesh, 512-bits/cycle/link, X-Y routing, 3 cycles/hop
Coherence	MESI, 64 B lines, in-cache directory, no silent drops
Main Memory	12-channel DDR4-2400 CL17

TABLE III
CHARACTERISTIC STATISTICS OF DATA SETS
($\bar{D}$ denotes the average vertex degree and d denotes the graph diameter)

Data sets	$\#Vertices$	$\#Edges$	$\bar{D}$	d
ego-Gplus (GL)	107,614	13,673,453	127	6
com-Amazon (AZ)	334,863	925,872	6	44
soc-Pokec (PK)	1,632,803	30,622,564	19	11
com-Orkut (OK)	3,072,441	117,185,083	76	9
com-LiveJournal (LJ)	3,997,962	34,681,189	17	17
com-Friendster (FS)	65,608,366	950,652,916	29	32

IV. EXPERIMENTAL EVALUATION

We use ZSim [44] to simulate the 64-core processor, whose parameters are listed in Table II as in previous work [59]. The simulated processor employs out-of-order cores, where each core owns private L1 and L2 cache and is modeled after and validated against Intel Skylake core. Note that the core is also extended to support AVX512 as MacSim [1]. The L3 cache is shared by all cores. The cores communicate with each other via mesh network with similar parameters as Intel Knights Landing [59]. We then integrate our DepGraph accelerator into the simulated 64-core processor and evaluate its performance through four popular graph algorithms [64], i.e., pagerank, adsorption, SSSP, and WCC. Six real-world graphs [2] in Table III are used in our evaluation. The program is compiled by GCC 9.2 with the -O3 flag and have vectorization.

To demonstrate the advantages of DepGraph, we first optimize Ligra [47] (which is one best-performing system for in-memory graph processing) by incorporating several optimizations such as asynchronous execution [64], abstraction-based optimization [62], and the ones described in the work [36] to optimize Ligra. The optimized version of Ligra is named Ligra-o, which is used as the performance baseline and outperforms the original Ligra by up to 4.6 times as shown in Section II. For *GL*, *AZ*, *PK*, *OK*, *LJ*, and *FS*, the preprocessing time of Ligra-o is 7.6, 0.4, 17.5, 67.3, 19.6, and 546.0 ms, respectively, while the preprocessing of DepGraph (accessing the graph twice) costs 8.0, 0.43, 18.9, 72.4, 21.4, and 595.1 ms, respectively. Our approach only increases the preprocessing time of Ligra-o by up to 9.2% because it needs extra time to find hub- and core-vertices. In general, the one-time preprocessing time and the execution time of graph processing solutions are reported separately [77]. The following results only report the execution time.

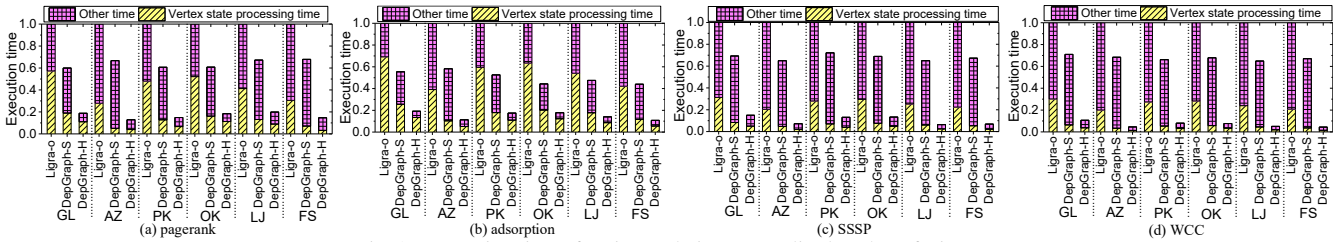


Fig. 9. Execution time of various solutions normalized to that of Ligra-o

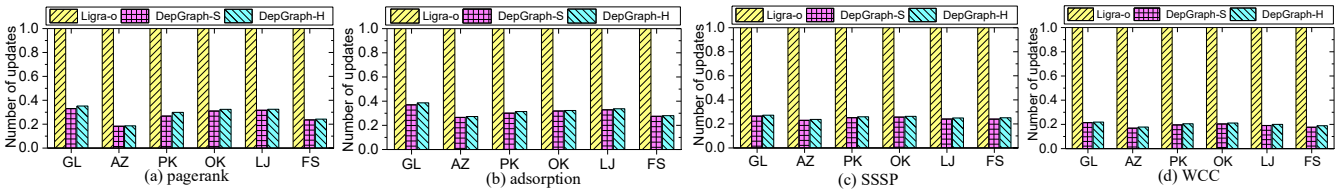


Fig. 10. Number of updates of different solutions normalized to that of Ligra-o

We then evaluate the DepGraph-augmented Ligra-o. DepGraph-S and DepGraph-H are the versions of Ligra-o integrated with software-only DepGraph and hardware implemented DepGraph, respectively, where λ=0.5% and β=0.001 by default. The stack's depth of DepGraph is set to 10. The software-only DepGraph is a fully software system implementing our approach, which fetches edges along dependency chains and maintains the hub index in memory for reuse during the execution. Note that both Ligra-o and DepGraph-S have been optimized to efficiently use SIMD, which outperform the corresponding non-SIMD versions by up to 2.04 and 2.17 times, respectively. Our hardware implemented DepGraph is compared with HATS [35], Minnow [59], and PHI [36], which are cutting-edge accelerators for graph processing on many-core processors. We calculate energy consumption of chip components using McPAT [25] and main memory using Micron DDR3L datasheets [34].

A. Execution Time Breakdown of DepGraph

To understand our approach, we first break down the execution time of the graph algorithm into the vertex state processing time and the other time (which includes the memory access time, etc.). Figure 9 shows the execution time of each graph algorithm run with different solutions on the simulated 64-core processor. We can observe that Ligra-o needs more vertex state processing time than other solutions. It is because Ligra-o needs more updates (due to slow propagations) and also there exist many dependencies between these propagations. DepGraph-S can address these issues. In Figure 9, the vertex state processing time of DepGraph-S is only 16.9%−37.0% of that of Ligra-o in different cases.

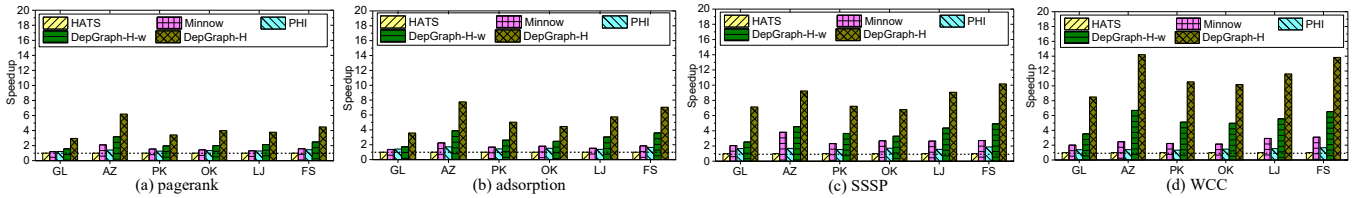
However, DepGraph-S has these benefits only at the cost of high runtime cost. The other time of DepGraph-S occupies 57.9%−95.0% (the cost for fetching the edges on-the-fly along the dependency chains occupies 36.5%−60.6%, and the cost to maintain and to use the hub index occupies 20.2%−32.7%) of the total execution time. It is apparently the key bottleneck of DepGraph-S. This runtime cost can be reduced in DepGraph-H with the hardware support, which enables our approach to perform much better than the existing solutions. In Figure 9, the other time of DepGraph-H is only 4.5%−22.9% of that of DepGraph-S, and only occupies 30.2%−78.2% of the total execution time in different cases. Note that the memory consumption needed by DepGraph-H for storing the hub index occupies 0.9%−2.8% of the total storage in our tested cases.

Figure 9 also shows that DepGraph-H achieves the best performance in all cases, which improves by up to 5.0−22.7 times over Ligra-o. It is because DepGraph-H not only benefits from the advantages of DepGraph-S in terms of fewer unnecessary updates, higher degree of effective parallelism, and no loading of the graph data related to the inactive vertices, but also achieves smaller runtime cost than DepGraph-S. Note that the hub-index based optimization contributes 56.9%−71.5% of the improvements achieved by DepGraph-H in various cases as shown in Figure 11. DepGraph-H still outperforms the existing solutions when its hub-index is disabled (i.e., DepGraph-H-w), due to effective state propagations along the dependency chains. It means that the mesh-like graphs can also benefit from DepGraph-H.

Figure 10 shows that the number of updates of DepGraph-H is little more than DepGraph-S because DepGraph-H may propagate a few more stale states than DepGraph-S when inter-chain state propagations are needed. DepGraph-H reduces the number of updates of Ligra-o by 61.4%−82.2% in different cases. It is mainly because new states of vertices can be propagated along the dependency chains more efficiently in DepGraph-H than in Ligra-o. The fewer number of updates not only lead to smaller processing cost, but also indicate smaller cost for accessing the graph data in DepGraph-H. Besides, according to Figure 4(a), although existing graph processing systems, e.g., Ligra-o, have high utilization of the cores, a major part of the core utilization is employed to handle the unnecessary updates. As described in Figure 12, DepGraph-H enables much higher average utilization of cores for processing useful updates.

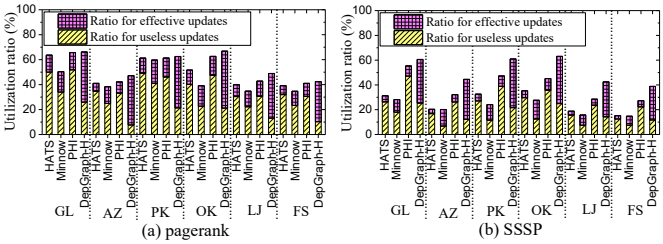
B. Comparison with Other Accelerators

We also evaluate the speedup of DepGraph-H in comparison with Ligra-o integrated with other accelerators. As depicted in Figure 11, Minnow performs better than both HATS and PHI for most cases because of the faster state propagation through the priority-based execution. However, as shown in



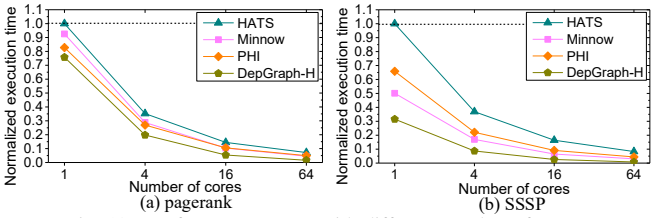
(a) pagerank (b) adsorption (c) SSSP (d) WCC

Fig. 11. Speedups of Ligra-o integrated with different accelerators



(a) pagerank (b) SSSP

Fig. 12. The breakdown of core utilization over different solutions



(a) pagerank (b) SSSP

Fig. 13. Performance on *FS* with different number of cores

TABLE IV
AREA AND POWER COST OF VARIOUS ACCELERATORS

Accelerators	Area overhead		Power overhead	
	(mm^2)	%core	(mW)	%TDP
HATS	0.007	0.38%	425	0.22%
Minnow	0.017	0.92%	849	0.43%
PHI	0.008	0.43%	493	0.25%
DepGraph	0.011	0.61%	562	0.29%

Figure 12, HATS, Minnow, and PHI suffer from low utilization of cores in propagating the useful vertex states. There are two main reasons. First, many unnecessary updates are generated and many vertices are inactive due to the slow propagations of many new states along the dependency chains. Second, Minnow and PHI also suffer from the poor data locality to propagate the states along dependency chains. Compared with them, DepGraph-H enables higher utilization of the cores in faster useful state propagations than these solutions. As described in Figure 11, in different cases, DepGraph-H outperforms HATS, Minnow, and PHI by up to 3.0−14.2, 2.2−5.8, and 2.4−10.1 times, respectively.

C. Scalability

Figure 13 shows that DepGraph-H performs better than others when more cores are used, due to higher utilization of cores to propagate useful vertex states. It is because more unnecessary updates are generated when more cores are used in HATS, Minnow, and PHI, although they get better performance under such circumstances. Besides, many vertices become inactive at running time and make the parallelism of threads still low for HATS, Minnow, and PHI, although more cores are used in them. However, with DepGraph-H, more effective data parallelism is ensured for graph algorithms

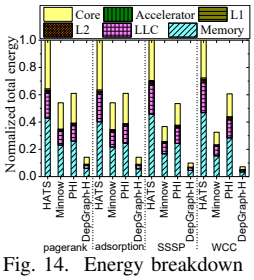
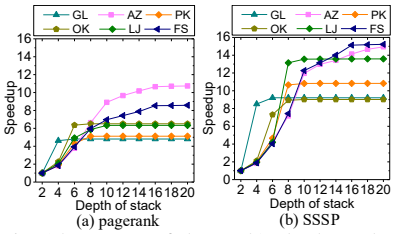
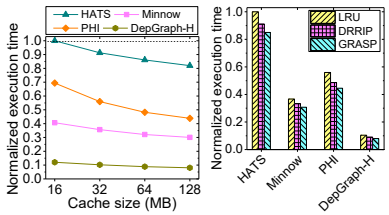


Fig. 14. Energy breakdown



(a) pagerank (b) SSSP

Fig. 15. Impact of the stack's depth on the performance of DepGraph-H



(a) Impact of the size of LLC (b) Impact of LLC policies

Fig. 16. Impact of LLC on SSSP over *FS*

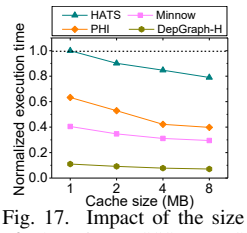


Fig. 17. Impact of the size of L2 cache on SSSP on FS

through our dependency transformation approach. Thus, when more cores are used, DepGraph-H still ensures fewer unnecessary updates and also more parallelism of the threads to propagate useful vertex states along dependency chains than existing solutions. It also indicates that DepGraph-H has better scalability than others.

D. Area Overhead and Energy Evaluation

Table IV lists the area cost and the power consumption of different accelerators under typical operating conditions, where they are written by Verilog RTL and are synthesized through a commercial 14 nm process with the same target frequency. We can find that the hardware support of DepGraph only incurs area cost of up to 0.6% of an out-of-order CPU core of the simulated chip, because DepGraph uses the existing memory subsystem to store the hub index. The area cost of DepGraph is only its hardware logic (i.e., HDTL and DDMU) and a few buffers (i.e., 6.1 Kbits of storage for the stack and 4.8 Kbits of storage for the *FIFO Edge Buffer*). Figure 14 evaluates their energy consumption normalized to that of HATS over *FS*. We can observe that DepGraph-H consumes less energy than the Ligra-o integrated with other accelerators for the convergence of graph algorithm due to more effective utilization of cores and faster convergence speed. Note that both the ratio of different accelerators' area/power to that of the simulated x86 core and the energy breakdown are evaluated approximately (with some errors) by McPAT [25] at 22 nm.

E. Sensitivity Studies

Figure 15 shows that the performance of DepGraph-H is almost flat after a stack depth of 10. It means that DepGraph-H

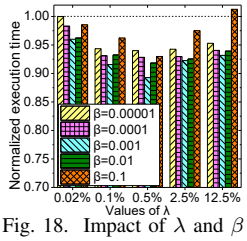


Fig. 18. Impact of λ and β on DepGraph-H over *FS*

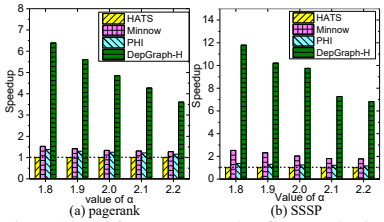


Fig. 19. Performance on the five synthetic graphs with different skewness

TABLE V
DESCRIPTION OF THE FIVE SYNTHETIC GRAPHS

α	1.8	1.9	2.0	2.1	2.2
$\#Edges$	667 M	246 M	104 M	56 M	37 M

is mostly insensitive of the stack depth and we can simply use a fixed depth. Figure 16(a) shows DepGraph-H consistently outperforms others when the size of *Last-Level Cache* (LLC) increases. Figure 16(b) shows the impact of three typical cache management policies, i.e., LRU, DRRIP [18], GRASP [13]. Note that GRASP is implemented based on DRRIP, and is used to manage LLC in other experiments. DRRIP performs better than LRU due to better temporal locality, and GRASP obtains the best performance by further reducing cache thrashing. The results mean that we can further optimize the management policies of LLC for DepGraph-H to get better performance, because a high-performance policy can reduce the cost to access its hub index. Similarly, Figure 17 shows that DepGraph-H performs better than others when L2 cache is larger.

Figure 18 shows the performance of SSSP on *FS* using DepGraph-H with different λ and β. It is a tradeoff for choosing their values. When too many hub-vertices are used, it may undermine its efficiency due to higher cost for accessing a larger hub index. Too few hub-vertices may miss some core-paths which are helpful in accelerating the propagations. Nevertheless, DepGraph-H outperforms existing solutions in default settings. Figure 19 shows the performance on randomly constructed power-law graphs (see Table V) with fixed ten-million vertices yet with different Zipfian factor α, which are generated in the same way as in the previous work [14]. A synthetic graph with lower α indicates that it has a higher vertex degree skewness [14]. We can find that DepGraph performs better when α is lower, because more propagations can be accelerated by the hub-index approach.

V. ADDITIONAL RELATED WORK

Software Graph Processing Systems. Lots of graph processing systems [31], [48], [66], [71]–[73] are recently developed. Ligra [47] is designed for shared-memory multicore machines, and NGraph [29] is the first graph processing system based on the hybrid memory [19], [28]. Mosaic [32] represents the graph by Hilbert-ordered tiles and can efficiently utilize many-core processors to handle the graph. For faster convergence, many systems, such as GraphLab [30] and CoRAL [52], asynchronously execute graph algorithms, and FBSGraph [69] further divides the graph into paths to handle it along these paths. HotGraph [68] and Wonderland [62] use a graph abstraction method to guide faster state propagation. However, they still suffer from many unnecessary updates and also low effective utilization of cores. Our DepGraph enables existing systems to get much higher effective parallelism towards faster convergence. Similar to us, Hub^2-Labeling approach [20] constructs a distance-preserving subgraph for high-degree vertices and also generates an index, however, it is an optimization specific to the query of k-degree shortest path by reducing the search space and cannot be used for general graph processing.

Other Hardware-based Solutions. Graphicionado [17] tries to reduce random memory access cost. GraphDynS [56] alleviates irregularities of graph processing using a software/hardware cooperative approach. GraFBoost [21] uses a sort-reduce accelerator to sequentialize the fine-grained random accesses to flash storage. Some hardware accelerators [9], [37], [49], [63] also propose to employ the processing-in-memory technology to improve its execution efficiency. GraphABCD [57] tries to get faster convergence on heterogeneous platforms using graph algorithms' block coordinate descent view. Although these hardware-based solutions have demonstrated promising results, the sequential nature of vertex state propagation in graph processing makes their provided parallelism with severe underutilization on the processor's cores. Different from them, DepGraph proposes a practical dependency-driven approach to fully exploit the high parallelism of many-core processors to accelerate these useful vertex state propagations.

VI. CONCLUSION

To the best of our knowledge, DepGraph is the first asynchronous graph processing accelerator on general processor cores, which enables each core in the many-core processor to efficiently support graph processing by effectively matching the sequential nature of vertex state propagation. Through dependency-aware graph data prefetching, it enables the core to propagate the vertices' new states efficiently along the dependency chains. DepGraph also maintains a set of fast shortcuts, i.e., *hub index*, for the core-paths, thus further accelerates most propagations and also maximizes the degree of effective parallelism. Our results show that DepGraph can greatly reduce unnecessary updates and improve utilization of cores in propagating useful states, resulting in much less energy consumption and a speedup of 22.7 times over an optimized software baseline with only 0.6% area cost.

ACKNOWLEDGEMENTS

We would like to thank all anonymous reviewers for their constructive comments and suggestions. This paper is supported by National Key Research and Development Program of China under grant No. 2018YFB1003500, National Natural Science Foundation of China under grant No. 61832006, 61825202, 62072193, and 61929103. This work is also supported by Science and Technology on Parallel and Distributed Processing Laboratory (PDL). Xiaofei Liao is the corresponding author of this paper.

REFERENCES

[1] "Macsim," https://github.com/gthparch/macsim, 2020.

[2] "Stanford large network dataset collection." http://snap.stanford.edu/data/index.html, 2020.

[3] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng, "Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk I/O," in *Proceedings of the 2017 USENIX Annual Technical Conference*, 2017, pp. 125–137.

[4] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 39:1–39:11.

[5] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 578–592.

[6] S. Baluja, R. Seth, D. Sivakumar, Y. Jing, J. Yagnik, S. Kumar, D. Ravichandran, and M. Aly, "Video suggestion and discovery for YouTube: Taking random walks through the view graph," in *Proceedings of the 17th International Conference on World Wide Web*, 2008, pp. 895–904.

[7] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture*, 2019, pp. 373–386.

[8] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[9] N. R. Challapalle, S. Rampalli, L. Song, N. Chandramoorthy, K. Swaminathan, J. Sampson, Y. Chen, and V. Narayanan, "GaaS-X: Graph analytics accelerator supporting sparse data representation using crossbar architectures," in *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture*, 2020, pp. 433–445.

[10] J. Cheng, Q. Liu, and Z. Li, "VENUS: Vertex-centric streamlined graph computation on a single PC," in *Proceedings of the 2015 IEEE International Conference on Data Engineering*, 2015, pp. 124–134.

[11] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, "NXgraph: An efficient graph processing system on a single machine," in *Proceedings of the 2016 IEEE International Conference on Data Engineering*, 2016, pp. 409–420.

[12] S. Eyerman, W. Heirman, K. Du Bois, J. B. Fryman, and I. Hur, "Many-core graph workload analysis," in *Proceedings of the 2018 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 282–292.

[13] P. Faldu, J. Diamond, and B. Grot, "Domain-specialized cache management for graph analytics," in *Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture*, 2020, pp. 234–248.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, 2012, pp. 17–30.

[15] 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 *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014, pp. 599–613.

[16] S. Grossman, H. Litz, and C. Kozyrakis, "Making pull-based graph processing performant," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018, pp. 246–260.

[17] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016, pp. 56:1–56:13.

[18] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 60–71.

[19] H. Jin, Z. Li, H. Liu, X. Liao, and Y. Zhang, "Hotspot-aware hybrid memory management for in-memory key-value stores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 4, pp. 779–792, 2020.

[20] R. Jin, N. Ruan, B. You, and H. Wang, "Hub-Accelerator: Fast and exact shortest path computation in large social networks," in *arxiv, abs/1305.0507*, 2013, pp. 1–12.

[21] S.-W. Jun, A. Wright, S. Zhang, S. Xu, and Arvind, "GraFBoost: Using accelerated flash storage for external graph analytics," in *Proceedings of the 45th ACM/IEEE Annual International Symposium on Computer Architecture*, 2018, pp. 411–424.

[22] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single PC," *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 13–23, 2015.

[23] A. Kusum, K. Vora, R. Gupta, and I. Neamtiu, "Efficient processing of large graphs via input reduction," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 245–257.

[24] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, 2012, pp. 31–46.

[25] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 469–480.

[26] X. Li, M. Zhang, K. Chen, and Y. Wu, "ReGraph: A graph processing framework that alternately shrinks and repartitions the graph," in *Proceedings of the 2018 International Conference on Supercomputing*, 2018, pp. 172–183.

[27] D. Liben-Nowell and J. Kleinberg, "The link-prediction problem for social networks," *Journal of the American Society for Information Science and Technology*, vol. 58, no. 7, pp. 1019–1031, 2007.

[28] H. Liu, R. Liu, X. Liao, H. Jin, B. He, and Y. Zhang, "Object-level memory allocation and migration in hybrid memory systems," *IEEE Transactions on Computers*, vol. 69, no. 9, pp. 1401–1413, 2020.

[29] W. Liu, H. Liu, X. Liao, H. Jin, and Y. Zhang, "Ngraph: Parallel graph processing in hybrid memory systems," *IEEE Access*, vol. 7, pp. 103 517–103 529, 2019.

[30] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[31] X. Lv, W. Xiao, Y. Zhang, X. Liao, H. Jin, and Q. Hua, "An effective framework for asynchronous incremental graph processing," *Frontiers of Computer Science*, vol. 13, no. 3, pp. 539–551, 2019.

[32] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *Proceedings of the 12th European Conference on Computer Systems*, 2017, pp. 527–543.

[33] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010, pp. 135–146.

[34] Micron, "1.35V DDR3L power calculator (4Gb x16 chips)," 2013.

[35] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018, pp. 1–14.

[36] A. Mukkara, N. Beckmann, and D. Sanchez, "PHI: Architectural support for synchronization and bandwidth efficient commutative scatter updates," in *Proceedings of the 52nd AnnualIEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 1009–1022.

[37] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling instruction-level pim offloading in graph computing frameworks," in *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture*, 2017, pp. 457–468.

[38] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture*, 2016, pp. 166–177.

[39] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford Digital Library Technologies Project, Tech. Rep., 1998.

[40] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in

Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2011, pp. 12–25.

[41] K. A. Ross, "Efficient hash probes on modern processors," in *Proceedings of the 23rd IEEE International Conference on Data Engineering*, 2007, pp. 1297–1301.

[42] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013, pp. 472–488.

[43] A. H. N. Sabet, J. Qiu, and Z. Zhao, "Tigr: Transforming irregular graphs for GPU-friendly graph processing," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 622–636.

[44] D. Sanchez and C. Kozyrakis, "ZSim: fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 475–486.

[45] S. L. Scott, "Synchronization and communication in the t3e multiprocessor," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 26–36.

[46] Z. Shao, L. Hou, Y. Ai, Y. Zhang, and H. Jin, "Is your graph algorithm eligible for nondeterministic execution?" in *Proceedings of the 44th International Conference on Parallel Processing*, 2015, pp. 430–439.

[47] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013, pp. 135–146.

[48] B. Si, Y. Liang, J. Zhao, Y. Zhang, X. Liao, H. Jin, H. Liu, and L. Gu, "Ggraph: An efficient structure-aware approach for iterative graph processing," *IEEE Transactions on Big Data*, vol. DOI: 10.1109/TBDATA.2020.3019641, pp. 1–13, 2020.

[49] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating graph processing using reram," in *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture*, 2018, pp. 531–543.

[50] S. Song, X. Liu, Q. Wu, A. Gerstlauer, T. Li, and L. K. John, "Start late or finish early: A distributed graph processing system with redundancy reduction," *Proceedings of the VLDB Endowment*, vol. 12, no. 2, pp. 154–168, 2018.

[51] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "GraphGrind: addressing load imbalance of graph partitioning," in *Proceedings of the 2017 International Conference on Supercomputing*, 2017, pp. 16:1–16:10.

[52] K. Vora, C. Tian, R. Gupta, and Z. Hu, "CoRAL: Confined recovery in distributed asynchronous graph processing," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 223–236.

[53] K. Vora, G. Xu, and R. Gupta, "Load the edges you need: A generic I/O optimization for disk-based graph processing," in *Proceedings of the 2016 USENIX Annual Technical Conference*, 2016, pp. 507–522.

[54] L. Wang, L. Zhuang, J. Chen, H. Cui, F. Lv, Y. Liu, and X. Feng, "Lazygraph: lazy data coherency for replicas in distributed graph-parallel computation," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018, pp. 276–289.

[55] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "SYNC or ASYNC: time to fuse for distributed graph-parallel computation," in *Proceedings of the 2015 ACM Sigplan Symposium on Principles and Practice of Parallel Programming*, 2015, pp. 194–204.

[56] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 615–628.

[57] Y. Yang, Z. Li, Y. Deng, Z. Liu, S. Yin, S. Wei, and L. Liu, "GraphABCD: Scaling out graph analytics with asynchronous block coordinate descent," in *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture*, 2020, pp. 419–432.

[58] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 178–190.

[59] D. Zhang, X. Ma, M. Thomson, and D. Chiou, "Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 593–607.

[60] G. Zhang, V. Chiu, and D. Sanchez, "Exploiting semantic commutativity in hardware speculation," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[61] G. Zhang, W. Horn, and D. Sanchez, "Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems," in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015, pp. 13–25.

[62] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen, "Wonderland: A novel abstraction-based out-of-core graph processing system," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 608–621.

[63] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "GraphP: Reducing communication for pim-based graph processing with efficient data partition," in *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture*, 2018, pp. 544–557.

[64] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 8, pp. 2091–2100, 2014.

[65] Y. Zhang, L. Gu, X. Liao, H. Jin, D. Zeng, and B. B. Zhou, "FRANK: A fast node ranking approach in large-scale networks," *IEEE Network*, vol. 31, no. 1, pp. 36–43, 2017.

[66] Y. Zhang, X. Liao, L. Gu, H. Jin, K. Hu, H. Liu, and B. He, "AsynGraph: Maximizing data parallelism for efficient iterative graph processing on gpus," *ACM Transactions on Architecture and Code Optimization*, vol. 17, no. 4, pp. 29:1–29:21, 2020.

[67] Y. Zhang, X. Liao, H. Jin, L. Gu, L. He, B. He, and H. Liu, "CGraph: A correlations-aware approach for efficient concurrent iterative graph processing," in *Proceedings of the 2018 USENIX Annual Technical Conference*, 2018, pp. 441–452.

[68] Y. Zhang, X. Liao, H. Jin, L. Gu, G. Tan, and B. B. Zhou, "HotGraph: Efficient asynchronous processing for real-world graphs," *IEEE Transactions on Computers*, vol. 66, no. 5, pp. 799–809, 2017.

[69] Y. Zhang, X. Liao, H. Jin, L. Gu, and B. B. Zhou, "FBSGraph: Accelerating asynchronous graph processing via forward and backward sweeping," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 5, pp. 895–907, 2018.

[70] Y. Zhang, X. Liao, H. Jin, B. He, H. Liu, and L. Gu, "DiGraph: An efficient path-based iterative directed graph processing system on multiple GPUs," in *Proceedings of the 2019 Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 601–614.

[71] Y. Zhang, X. Liao, H. Jin, and G. Tan, "SAE: Toward efficient cloud data analysis service for large-scale social networks," *IEEE Transactions on Cloud Computing*, vol. 5, no. 3, pp. 563–575, 2017.

[72] Y. Zhang, X. Liao, X. Shi, H. Jin, and B. He, "Efficient disk-based directed graph processing: A strongly connected component approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 4, pp. 830–842, 2018.

[73] Y. Zhang, J. Zhao, X. Liao, H. Jin, L. Gu, H. Liu, B. He, and L. He, "CGraph: A distributed storage and processing system for concurrent iterative graph analysis jobs," *ACM Transactions on Storage*, vol. 15, no. 2, pp. 10:1–10:26, 2019.

[74] J. Zhao, Y. Zhang, X. Liao, L. He, B. He, H. Jin, H. Liu, and Y. Chen, "GraphM: an efficient storage system for high throughput of concurrent graph processing," in *Proceedings of the 2019 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 3:1–3:14.

[75] A. C. Zhou, S. Ibrahim, and B. He, "On achieving efficient data transfer for graph processing in geo-distributed datacenters," in *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems*, 2017, pp. 1397–1407.

[76] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu, "Hitgraph: High-throughput graph processing framework on fpga," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2249–2264, 2019.

[77] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "GraphQ: Scalable pim-based graph processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 712–725.