ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture

Guohao Dai[1], Tianhao Huang[1], Yuze Chi[2], Ningyi Xu[3], Yu Wang[1], Huazhong Yang[1]
[1]Department of Electronic Engineering, TNLIST, Tsinghua University, Beijing, China
[2]Computer Science Department, University of California, Los Angeles, USA
[3]Hardware Computing Group, Microsoft Research Asia, Beijing, China
[1]dgh14@mails.tsinghua.edu.cn, [1]yu-wang@tsinghua.edu.cn, [3]xu.ningyi@microsoft.com

ABSTRACT

The performance of large-scale graph processing suffers from challenges including poor locality, lack of scalability, random access pattern, and heavy data conflicts. Some characteristics of FPGA make it a promising solution to accelerate various applications. For example, on-chip block RAMs can provide high throughput for random data access. However, large-scale processing on a single FPGA chip is constrained by limited on-chip memory resources and off-chip bandwidth. Using a multi-FPGA architecture may alleviate these problems to some extent, while the data partitioning and communication schemes should be considered to ensure the locality and reduce data conflicts.

In this paper, we propose ForeGraph, a large-scale graph processing framework based on the multi-FPGA architecture. In ForeGraph, each FPGA board only stores a partition of the entire graph in off-chip memory. Communication over partitions is reduced. Vertices and edges are sequentially loaded onto the FPGA chip and processed. Under our scheduling scheme, each FPGA chip performs graph processing in parallel without conflicts. We also analyze the impact of system parameters on the performance of ForeGraph. Our experimental results on Xilinx Virtex UltraScale XCVU190 chip show ForeGraph outperforms state-of-the-art FPGA-based large-scale graph processing systems by 4.54x when executing PageRank on the Twitter graph (1.4 billion edges). The average throughput is over 900 MTEPS in our design and 2.03x larger than previous work.

Keywords

large-scale graph processing; multi-FPGA architecture

1. INTRODUCTION

With demand for data analysis continuing to grow, the large-scale graph processing which discovers relationships among data is gaining increasing attention in many domains [1]. Previous work has provided large-scale graph processing systems, including CPU-based [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13], GPU-based [14, 15], FPGA-based [16, 17, 18, 19, 20, 21, 22, 23], and emerging systems [24].

As emphasized in this work, the key problem in large-scale graph processing is to provide a high bandwidth of data access [25, 26]. However, some characteristics of large-scale graphs bring challenges to fully utilizing bandwidth. These challenges include: (1) **Poor locality.** Graphs represent unstructured relationships between entities, and thus a small partition can have access to the whole graph. Poor locality leads to frequent global data access, while only local data access will have a large bandwidth in state-of-the-art computing platforms. (2) **Lack of scalability.** Communication over partitions causes heavy traffic in large-scale graph processing. Thus, it is difficult to design a system which scales to larger graphs. (3) **Random data access pattern.** The data access pattern of two neighboring vertices can be quite unlike. Such unstructured characteristic of graphs randomizes graph data access pattern. (4) **Heavy data conflicts.** Vertices from different partitions may read/write the same vertex simultaneously, leading to heavy conflicts. Moreover, unpredictable data access pattern brings great challenges to avoid conflicts. These four challenges need to be carefully considered so as to provide a high bandwidth and design a high-performance large-scale graph processing system.

To tackle these challenges, many solutions have been designed in previous work and most of them mainly focus on fully utilizing the bandwidth. GraphChi [4] divides a large graph into several *intervals* and *shards* as partitions of vertices and edges. Based on the partitioning scheme, the locality is ensured by accessing each partition in turns. Some previous work [10, 18, 4, 23] also sorts data to eliminate the randomness and conflicts of graph data access. However, the overhead of pre-processing on sorting data before execution needs to be reckoned especially when the graph may dynamically change during run-time. Compared with CPUs and GPUs, the random access feature of on-chip BRAMs is provided to implement random data access with high throughput on FPGA. However, the size of on-chip BRAMs of one FPGA chip is much smaller than the typical size of a large graph. Consequently, using the multi-FPGA architecture is a promising way to provide larger on-chip BRAM resources. However, most of FPGA-based systems are designed for one FPGA board [23] or require a global-accessible memory [18], with poor scalability for larger graphs.

To provide a high-performance large-scale graph processing system based on the multi-FPGA architecture, we design ForeGraph. We divide graphs into small partitions and as-

FPGA '17, February 22-24, 2017, Monterey, CA, USA
© 2017 ACM. ISBN 978-1-4503-4354-1/17/02. . . $15.00
DOI: http://dx.doi.org/10.1145/3020078.3021739

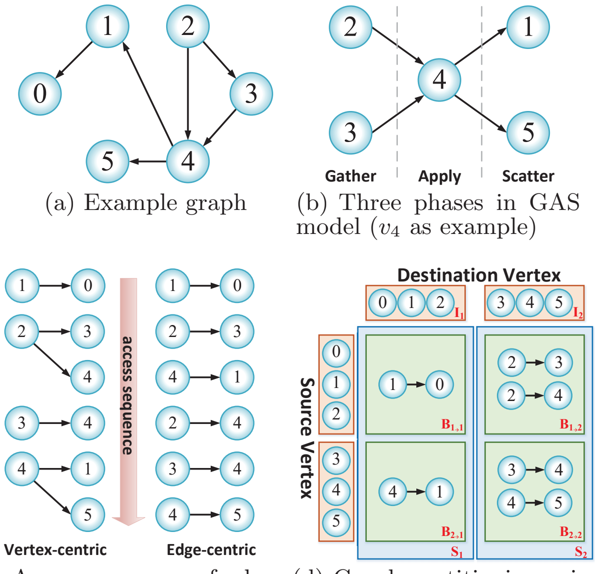
Table 1: Notations of a graph

Notation	Meaning
G	a graph $G = (V,\ E)$
V	vertices in G, $\lvert V \rvert = n$
E	edges in G, $\lvert E \rvert = m$
v_i	vertex i
$e_{i.j}$	edge from v_i to v_j
e_{src}	source vertex of edge e
e_{dst}	destination vertex of edge e
I_x	interval x
S_y	shard y
$B_{x \to y}$	block $x.y$ linked from I_i to I_j
$SI_{x,i}$	the i-th sub-interval of I_x
$SB_{x \to y,i,j}$	the $(i,\ j)$ sub-block of $B_{x \to y}$
P	number of intervals
Q	number of sub-intervals in an interval
K	number of processing elements on a chip

sign each partition to an FPGA board to ensure the locality of data access. Communication overhead among different FPGA boards is minimized to make ForeGraph scalable to large graphs. Vertices and edges are sequentially loaded onto FPGA chips to avoid random data access. Each partition is further divided into smaller ones and assigned to different processing elements (PEs) on FPGA chips, so conflicts are eliminated. Such multi-FPGA architecture can provide sufficient on-chip BRAM resources and off-chip bandwidth, which are essential to improve the performance of FPGA-based large-scale graph processing systems. Specifically, this paper makes the following contributions.

- **Scalable multi-FPGA graph processing.** The multi-FPGA architecture provides large on-chip BRAM resources with random access feature and sufficient off-chip bandwidth of graph data access. Data are allocated to each FPGA board rather than stored in a global-accessible memory (e.g. *Shared-Vertex Memory* in [18]) (Section 3.2 and Section 3.3). Moreover, communication overhead among boards is minimized (Section 3.4). These two technologies make our system scale to larger graphs.
- **Pre-processing with low overhead.** Vertices and edges are divided into partitions according to their indexes (Section 3.5). Data are not required to be sorted within each partition thus the overhead of pre-processing is reduced (from $O(m \log m)$ to $O(m)$, m denotes the number of edges). Locality is ensured, and conflicts are removed under our partitioning scheme.
- **Fully utilizing off-chip bandwidth.** We minimize data transmission on one board to fully utilize off-chip bandwidth. We adopt several optimization techniques in Section 4. For example, we compress the vertex index and use only 4 Bytes to represent an edge (2 Bytes for source and destination vertex respectively), even though there are millions of vertices in the graph.
- **Extensive experiments.** We have conducted comprehensive experiments to evaluate the performance in Section 6. Experimental results on five graphs show that ForeGraph can execute graph algorithms on graphs with billions of edges. ForeGraph outperforms state-of-the-art FPGA-based systems by 5.89x, and the average throughput is 2.03x larger than previous work.

The remaining of this paper is organized as follows. Section 2 introduces the background information of large-scale graph processing and correlative systems. The whole architecture of ForeGraph is shown in Section 3. Some optimization methods to fully utilize off-chip bandwidth are shown in Section 4. The performance of ForeGraph is analyzed and presented in Section 5 and Section 6 from both theoretical and experimental perspectives. We finally conclude this paper in Section 7.



(a) Example graph (b) Three phases in GAS model (v_4 as example) (c) Access sequence of edges in VC and EC model (d) Graph partitioning using intervals and shards

Figure 1: Example graph and corresponding models.

2. BACKGROUND AND RELATED WORK

In this section, the background information of graph processing models is presented. Then, we will introduce previous FPGA-based large-scale graph processing systems. Notations used in this paper are shown in Table 1.

2.1 Graph Processing Models

Let V and E denote the vertex and edge sets in a graph G, the computation task over $G = (V,\ E)$ is to calculate the updated value of V and E. We assume each edge is directed, and an undirected graph can be realized by adding an opposing edge to each directed edge.

Gather-Apply-Scatter. When updating the value of V, updates are propagated from the source vertex to the destination vertex. Such model is known as the *Gather-Apply-Scatter* (GAS) model [2] which divides the update into three phases. In the *Gather* phase, a vertex receives value from source vertices of in-edges. Then, the updated value is calculated in the *Apply* phase. After that, the updated value is propagated to the destination vertices of out-edges. The GAS model can be executed in the form of iterations. In each iteration, each edge is accessed once to propagate updates from the source vertex to the destination vertex. Figure 1(b) illustrates the three phases of the GAS model.

Vertex-centric and edge-centric. As mentioned in the GAS model, updates are propagated from vertices to vertices through edges. Thus, the access sequence of edges differentiates different models, including *vertex-centric* (VC) model [6] and *edge-centric* (EC) model [12]. In VC model, a vertex scatters value to destination vertices of all out-edges (or

Algorithm 1 Pseudo-code of Breadth-First Search

```
Input: $G = (V, E)$, root vertex $r$
Output: depth of each $v \in V$, $d(v)$
 1: $d(r) = 0$
 2: for each $v \in V$ & $v \neq r$ do
 3:     $d(v) = \infty$
 4: end for
 5: $finished =$ false
 6: while ($finished =$ false) do
 7:     $finished =$ true
 8:     for each edge $e$ do
 9:         if $d(e_{src}) + 1 < d(e_{dst})$ then
10:             $finished =$ false
11:             $d(e_{dst}) = d(e_{src}) + 1$
12:         end if
13:     end for
14: end while
15: return $d(v)$, $v \in V$
```

gathers value from source vertices of all in-edges). In contrast, in EC model, all edges are sequentially accessed while the access sequence of source/destination vertices is disordered. Both VC and EC model have been implemented in previous systems and achieved excellent performance. Figure 1(c) shows an example of VC and EC model.

Interval-shard based partitioning. Graph partitioning is a widely used method which ensures the locality of graph data access. GraphChi [4] uses an interval-shard based partitioning model. Vertices and edges in a graph are divided into P *intervals* (vertex sets) and *shards* (edge sets). Later systems [10, 11] further divide edges into P^2 blocks according to the corresponding intervals of the source and destination vertices. For example, $B_{x \rightarrow y}$ contains all edges linked from I_x to I_y. Figure 1(d) shows an example of interval-shard based partitioning model.

Based on these models, the computation task is performed in the form of iterations. In each iteration, all blocks are accessed at most once. Updates are propagated from source vertices to destination vertices. Algorithm 1 shows the pseudo-code of Breadth-First Search (BFS) using these models. In the beginning, the depth of the root vertex is set to zero, and others are infinite. In each iteration, edges are sequentially accessed (Note that the accessing order of edges is not specified in Algorithm 1, we will explain the detailed order in ForeGraph in our implementation.). The depth of the corresponding destination vertex will be modified according to the depth of source vertex. Such algorithm can be easily transformed into other graph algorithms by modifying the code of propagation.

2.2 FPGA-based Graph Processing Systems

FPGA has been proved as a promising solution to many applications and previous work has provided large-scale graph processing systems based on FPGA [16, 27, 17, 18, 19, 20, 21, 22, 23]. Most of these systems are designed for a single FPGA chip. Some of these systems are only dedicated to specific algorithms, like Breadth-First Search (BFS) or PageRank (PR). There are also many general purposed systems which can apply to different graph algorithms, including GraphStep [20], GraphGen [21] and GraphOps [22]. GraphStep and GraphGen are two systems that applied VC model to FPGA. GraphOps provides a modular hardware library for constructing accelerators for graph analytics algorithms. Shijie *et al.* [23] proposed a system to minimize row-conflicts using EC model. However, the size of graphs on all these systems is limited by memory resources on an FPGA board.

There are also some systems based on the multi-FPGA architecture. Betkaoui *et al.* [17] proposed a BFS solution on Convey HC-1 machine consisted of 4 FPGA boards. However, this system can hardly be applied to other graph algorithms. FPGP [18] provided a large-scale graph processing framework and it can be expanded to multi-FPGA architecture using a *Shared-Vertex Memory* (SVM). However, since all FPGA boards need to be connected to SVM, the system performance, as well as the scalability, is limited by the bandwidth of the SVM.

3. SYSTEM ARCHITECTURE

In this section, we will discuss the system architecture of ForeGraph. The data allocation and processing flow in ForeGraph will be explained in detail, followed by interconnection scheme and partitioning scheme.

3.1 Overall Architecture

The overall architecture of ForeGraph is shown on the left of Figure 2. ForeGraph consists of several FPGA boards. On each board, there is an FPGA chip to perform processing logic and off-chip memory to store graph data. All boards are connected by the interconnection. Such interconnection can be realized using the bus (e.g. PCI-e), directed optical fiber connections or other available structures. The detailed processing logic is shown on the right of Figure 2. The logic includes an interconnection controller, an off-chip memory controller, a data controller, a dispatcher and several processing elements (PEs).

- **Interconnection controller.** Data transmission among FPGA boards is controlled by interconnection controller.
- **Off-chip memory controller.** The off-chip memory controller arranges the data read/write of the off-chip memory. The controller can be realized by using existing IP core generators (e.g. Memory Interface Generator in Xilinx Vivado). When performing graph algorithms, data loaded to processing elements are all from the off-chip memory through this controller.
- **Data controller.** The data controller connects the off-chip memory controller and the interconnection controller. It packs and calculates the memory address and target board ID when transmitting data among boards.
- **Processing elements (PEs).** PEs are the kernel logic for executing graph algorithms on the FPGA board. As mentioned in Section 2.1, updates are propagated from the source vertex to the destination vertex using the corresponding edge. Thus, each PE contains a source buffer and a destination buffer storing source vertices and destination vertices respectively. Both source buffer and destination buffer are implemented using general purposed dual-port BRAMs. There is another edge buffer storing edges loaded from the off-chip memory. Edges are sequentially loaded from off-chip memory when updating. Both vertices and edges are sent to the processing logic, and the results will be calculated and written to the destination buffer. Different graph algorithms only differ in processing logic. When the updating for all vertices in the destination buffer finished, the results will be written to the off-chip memory, and new vertices and edges will

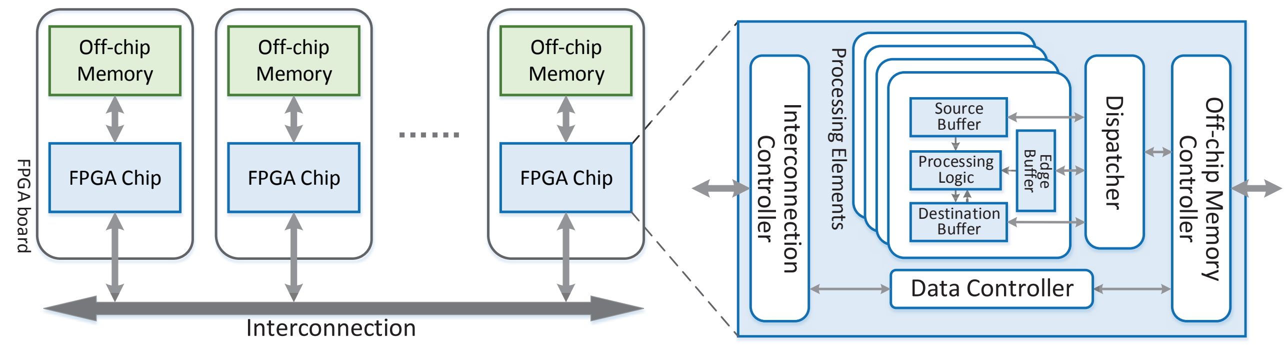


Figure 2: Overall architecture of ForeGraph (left) and on-chip processing logic (right).

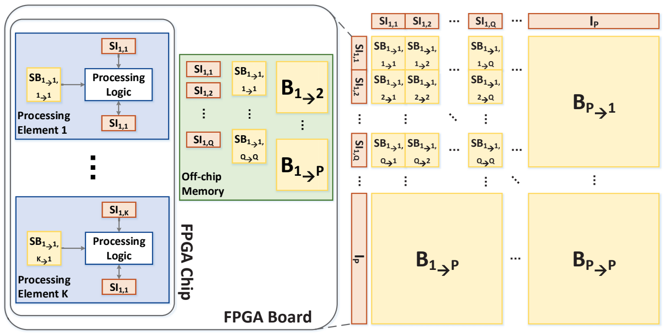


Figure 3: On-board data allocation (left) and two-level partitioning in ForeGraph (right).

be loaded to these buffers. Assuming the bandwidth of off-chip memory is around 10 GB/s per board, and the processing logic runs at the frequency around 200 MHz. We use 8 Bytes to represent an edge (4 Bytes for source vertex and destination vertex respectively). Based on the fact that the throughput of a single PE (200 MHz $\times$ 8 Bytes = 1.6 GB/s) is much smaller than the bandwidth of off-chip memory, using several PEs can fully utilize the off-chip memory bandwidth.

- **Dispatcher.** The dispatcher connects the off-chip memory controller and data buffers in PEs. When vertices and edges are loaded from the off-chip memory, the dispatcher sends data to corresponding PEs. The data allocation in different PEs is explained in detail in Section 3.2.

3.2 Data Allocation in Off-chip Memory

Graphs are divided into intervals (I) and shards (S) in ForeGraph. Each interval, as well as its corresponding shard, is assigned to the off-chip memory on an FPGA board. For example, in a ForeGraph system consisting of P FPGA boards, I_1 and S_1 are stored on the first FPGA board. S_1 consists of P blocks namely $B_{1\to1} \sim B_{P\to1}$. Each block is responsible for updating I_1 using different source intervals. These source intervals are stored in other FPGA boards and loaded to the first board in turn during run-time.

Considering there are several PEs in a chip and each PE contains two exclusive vertex buffers using BRAMs, the on-chip memory resources are not enough to store an interval as the graph size continues to grow. In ForeGraph, we adopt a two-level graph partitioning scheme shown on the right of Figure 3. Take the first interval I_1 as an example, in this two-level graph partitioning scheme, I_1 is further divided into Q sub-intervals, $SI_{1,1} \sim SI_{1,Q}$. Correspondingly, block $B_{1\to1}$ is further divided into Q^2 sub-blocks, $SB_{1\to1,1\to1} \sim SB_{1\to1,Q\to Q}$. Each sub-block is responsible for updating a destination sub-interval using a source sub-interval.

When executing graph algorithms, different source sub-intervals are loaded to different PEs, while the destination sub-intervals are same in these PEs. These PEs update the destination sub-interval using corresponding sub-blocks. An example of data allocation when executing graph algorithms in ForeGraph is shown on the left of Figure 3. $SI_{1,1} \sim SI_{K,1}$ are loaded to PE 1 $\sim$ PE K and the destination sub-interval is $SI_{1,1}$ in all PEs. Edges in corresponding sub-blocks are loaded to edge buffers of each PE. When all PEs finished updating for $SI_{1,1}$, ForeGraph substitutes unused sub-intervals in the off-chip memory since those in PEs can continue to execute graph algorithms.

Intervals on other boards are loaded to local off-chip memory in turns. For example, the processing flow of the first board: updating I_1 using I_1 and $B_{1\to1}$ $\to$ loading I_2 from the second board $\to$ updating I_1 using I_2 and $B_{2\to1}$ $\to$ discarding I_2 on the first board, and so on.

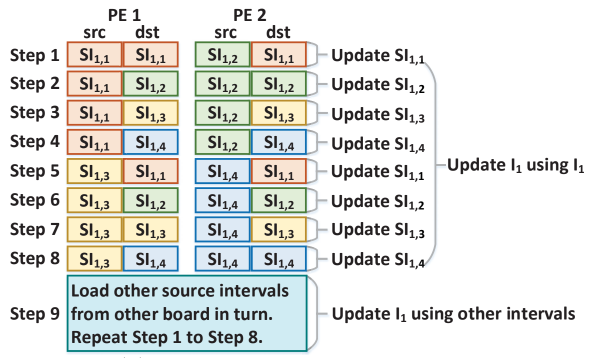
3.3 On-chip Data Replacement Flow

When using an interval to update another interval, all Q^2 sub-blocks will be accessed. However, only K sub-intervals are processed at one time. Thus, ForeGraph schedules how to substitute sub-intervals in off-chip memory for those on the chip. An example of two different replacement strategies is shown in Figure 4. An interval is divided into four sub-intervals, and two PEs are implemented on the chip.

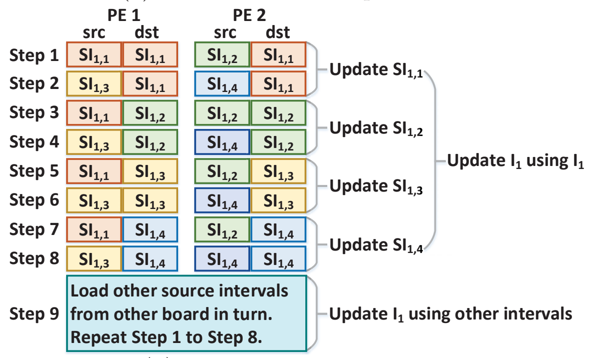
In the destination-first replacement (DFR) strategy, when two PEs finish updating the same destination sub-interval, ForeGraph writes it to the off-chip memory and replaces it with another sub-interval (Step 1 to Step 8 in Figure 4(a)). After all sub-intervals being updated using source sub-intervals in two PEs, ForeGraph replaces them with other new sub-intervals (Step 4 to Step 5 in Figure 4(a)). When all edges in $B_{1\to1}$ have been accessed, other intervals will be loaded, and ForeGraph will repeat previous steps using these intervals as source intervals (Step 9 in Figure 4(b)).

In the source-first replacement (SFR) strategy, the source sub-intervals rather than the destination sub-intervals will be replaced (Step 1 to Step 2, Step 3 to Step 4, Step 5 to Step 6, Step 7 to Step 8 in Figure 4(b)). When a sub-interval has been updated by all sub-intervals, ForeGraph replaces it with a new sub-interval (Step 2 to Step 3, Step 4 to Step 5, Step 6 to Step 7 in Figure 4(b)). Similarly, other intervals will be loaded, and previous steps will be repeated after all edges in $B_{1\to1}$ have been accessed (Step 9 in Figure 4(b)).

DFR and SFR differ in the data amount they read from/write



(a) Destination-first replacement.



(b) Source-first replacement.

Figure 4: Two different replacement strategies.

to off-chip memory. In both DFR and SFR, all Q^2 sub-blocks (or sub-interval pairs) need to be processed. For there are K PEs on a chip, we need Q^2/K steps to finish the updating of a sub-block. For example, in Figure 4 with $K = 2$ PEs on a chip and $Q = 4$ sub-intervals in an interval, there are 8 ($=4^2/2$) steps in total. In DFR, the destination sub-interval (same in all PEs) needs to be written to the off-chip memory, and a new sub-interval is loaded after each step. Thus, the read and write time for destination sub-intervals in DFR are both Q^2/K. Moreover, all Q source sub-intervals need to be loaded once. Consequently, the number of sub-intervals read/write are $(Q+Q^2/K)$ and Q^2/K respectively. In SFR, source sub-intervals in all K PEs need to be replaced after each step. Thus, the read time for source sub-intervals is $(Q^2 = Q^2/K \times K)$. Moreover, all destination sub-intervals need to be read/written once in SFR, which results in Q more read/write times of sub-intervals. According to the analysis, the number of sub-intervals read from/written to the off-chip memory is $(Q+Q^2)$ and Q respectively in SFR.

Table 2: Number of sub-intervals read from/written to the off-chip memory when processing a block

	read	write
destination-first replacement	$Q+Q^2/K$	Q^2/K
source-first replacement	$Q+Q^2$	Q

As we can see from Table 2, the advantage of DFR lies in the read time of sub-intervals while SFR costs less write time (we assume that $1 < K < Q$). Let T_r and T_w denote the average read/write time of a sub-interval, Formula (1) shows the situation where DFR outperforms SFR.

$$(Q+\frac{Q^2}{K}) \times T_r + \frac{Q^2}{K} \times T_w < (Q+Q^2) \times T_r + Q \times T_w \quad (1)$$

In ForeGraph, data are stored in DRAMs (same read/write bandwidth, different from other emerging devices like the non-volatile memory). Therefore, it is fair to assume that $T_r = T_w$. Formula (1) can be simplified into Formula (2).

$$(Q+1)(K-2) > -2 \quad (2)$$

Generally speaking, there are more than two PEs on a chip, which leads to $(Q+1)(K-2) \geq 0 > -2$. Thus, we adopt DFR in ForeGraph to minimize the data transmitted between the chip and the off-chip memory.

3.4 Interconnection

Much previous work proposed interconnection schemes among FPGAs like Catapult [28]. In Catapult, up to 48 FPGAs are connected using SerialLite III link in a torus network. It provides a peak theoretical bandwidth at 2×766 MB/s of each connection. The latency of each connection is around 400 ns. ForeGraph adopts the interconnection scheme in Catapult. We simulate the network consumption and compare it with other network structure (e.g. mesh, bus, and *etc.*) in Section 6.

Compared with distributed systems like Pregel [6] which transmits *messages* (update value from source vertices) to other computing nodes, we combine *messages* and update vertices locally in ForeGraph. Only updated the value of vertices are transmitted, and we minimize the data transmission amount so ForeGraph scales to larger graphs.

3.5 Index-based Partitioning

In ForeGraph, vertices are divided into P intervals and further into $P \cdot Q$ sub-intervals. Edges are also classified by their source and destination vertices. Different from previous systems like Graphchi [4] which needs to sort all edges in (sub-) blocks, ForeGraph only needs to assign vertices and edges to the corresponding (sub-) intervals and (sub-) blocks. Such implementation can significantly reduce the time consumption of pre-processing.

Before partitioning, we determine P and Q. Then all vertices are assigned to corresponding sub-intervals using *hash* function. For example, v_1, $v_{1+\frac{n}{PQ}}$, $v_{1+2\cdot\frac{n}{PQ}}$, and *etc.* are assigned to $SI_{1,1}$ (such partitioning method can balance the size of each block, shown in Table 8). Edges are also classified in this way without sorting. We reduce the overhead of pre-processing from $O(m \log m)$ to $O(m)$ by using index-based partitioning scheme (m denotes the number of edges).

Another advantage of this index-based partitioning is the fact that it can easily apply to dynamic graph algorithms. Previous systems need to sort all edges before updating, thus when the structure of the graph changes (e.g. inserting/deleting edges), the entire pre-processing needs to be redone. In ForeGraph, such overhead can be avoided because the order of edges in a (sub-) block is not required.

4. SYSTEM OPTIMIZATION

Based on the design for multi-FPGA in Section 3, we introduce three optimization methods to reduce data transmission amount and fully utilize PEs on one FPGA board.

4.1 Vertex Index Compression

Both vertices and edges are stored in off-chip memory thus compressing these data can significantly improve the performance of ForeGraph. In ForeGraph, we need to store the

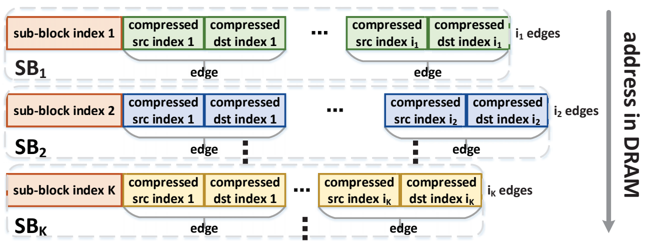


Figure 5: Vertex index compression using the sub-block index (SB_x contains i_x edges).

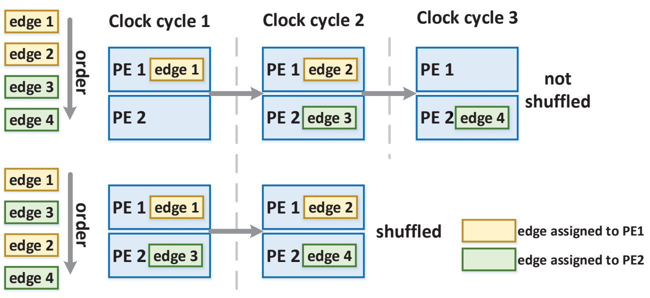


Figure 6: Shuffling edges to fully utilize PEs (FPGA chip can load two edges per clock cycle).

value of each vertex and the source/destination vertex indexes of each edge. The storage space for the value of each vertex is related to the dedicated algorithms (e.g. 8bits for the depth of each vertex in BFS). The compression of these data is not in the scope of this paper's discussion. The storage space for each edge is twice the size of the vertex index. For example, in the Twitter [29] graph which consists of 42 million vertices, we need $\log_2(42 \times 10^6) = 25$ bits to represent a vertex in this graph. The length of the vertex index can be even over 32 bits when the graph contains more than 4 billions ($= 2^{32}$) of vertices.

Source/Destination vertices of edges in a sub-block are all in the same sub-interval. We can use a sub-block index as a prefix to the vertex index. For instance, assuming there are 100 sub-intervals in the graph, we divide vertices with constant stride (100) into a sub-intervals. In this way, SI_1 includes v_1, v_{101}, v_{201}... In this sub-interval, we use 1 as a prefix and all vertices in SI_1 can be indexed according to the position in sub-interval (e.g. v_{101} is the second vertices in SI_1). Thus, the indexes of vertices do not exceed $\frac{n}{100}$ (number of vertices in a sub-interval).

Figure 5 shows the data placement in DRAM using our vertex index compression method. Each sub-block begins with its sub-block index, followed by compressed edge index. In ForeGraph implementation, we use 2 Bytes (16 bits) to represent the vertex index. Thus, there are less than $2^{16} = 65536$ vertices in a sub-interval.

4.2 Shuffling Edges

As mentioned in Section 3.3, K PEs on a chip update one sub-interval using K consecutive sub-blocks. Utilization of PEs is inefficient in the way shown in Figure 5. Consecutive edges will be sent to only one PE because the source vertices are in the same sub-interval. However, a PE can only update one edge per clock cycle. Meanwhile, other PEs are idle in this situation. To settle such problem, we shuffle edges in these K sub-blocks.

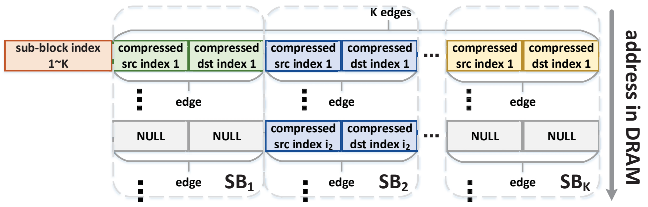


Figure 7: Shuffling edges in K sub-blocks (assuming SB_2 is larger than SB_1 and SB_K, $i_2 > i_1, i_2 > i_k$).

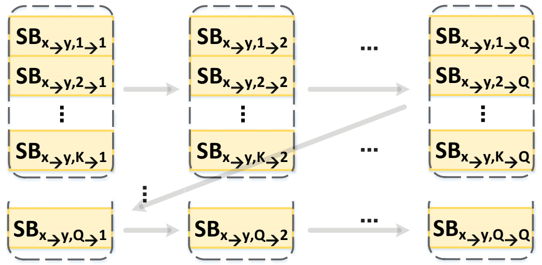


Figure 8: The order of accessing sub-blocks (sub-blocks in a dashed box are shuffled).

Figure 6 shows an example of why shuffling edges can fully utilize all PEs. In Figure 6, there are two PEs and four edges are assigned to both of them. The bandwidth of the off-chip memory provides the throughput of loading two edges to the FPGA chip per cycle. If edges in a sub-block are in consecutive order, it takes three clock cycles to finish updating because only one edge is processed in the first and third clock cycle. However, if edges are shuffled, it only takes two clock cycles, and two edges are processed during each cycle.

Based on this shuffling method, we proposed the edge shuffling method which is shown in Figure 7. Edges in K consecutive sub-blocks are shuffled. K consecutive edges in DRAM are in different sub-blocks thus they are sent to different PEs. If the sizes of sub-blocks are different, ForeGraph uses a *NULL* edge to fill in the blank position (gray blocks in Figure 7). We adopt DFR thus the destination sub-interval is replaced when all PEs finished updating. Figure 8 shows an example of the accessing order of sub-blocks in $B_{x\to y}$. K consecutive sub-intervals are loaded to the chip to update all sub-intervals. After loading sub-intervals, shuffled edges are loaded and dispatched to each PE.

4.3 Skipping Useless Blocks

In Algorithm 1, edges in all blocks are accessed in one iteration. However, previous work [30, 31] show that in algorithms like BFS, only some vertices are updated in one iteration. If a vertex is not updated, its neighbor vertices will not be updated in the next iteration. Thus, we do not have to access its outgoing edges.

Based on such observation, we can skip some edges if their source vertices are not updated in the prior iteration. Furthermore, if all vertices in one (sub-) interval have not been updated in one iteration, outgoing edges in (sub-) blocks with source vertices in the (sub-) interval do not need to be accessed in the next iteration. In this way, we can load fewer edges from the off-chip memory and skip (sub-) blocks which are unnecessary to be transmitted. In the implementation in ForeGraph, we use one bit in a bitmap [31] to represent if a sub-interval is updated in an iteration.

Table 3: Notations used in analysis

Notation	Meaning
BW_{mem}	bandwidth of the off-chip memory
BW_{int}	bandwidth of the interconnection
S_v	space used to store a vertex
S_e	space used to store an edge
M_{bram}	on-chip BRAM size
f	frequency of on-chip logic

5. THEORETICAL ANALYSIS

In ForeGraph, parameters like P and Q need to be set before implementation. In this section, we analyze how these parameters influence the performance of ForeGraph. Notations used in this section are listed in Table 3.

5.1 Modeling of ForeGraph

The processing time of an FPGA board mainly includes the following three parts:

- $T_{process}$, time of reading/writing sub-intervals from/to the off-chip memory before updating.
- T_{load}, time of loading edges from the off-chip memory when updating. We assume that on-chip throughput of all PEs is larger than off-chip bandwidth.
- $T_{transmit}$, time of loading intervals from other boards.

We assume that all n vertices and m edges are evenly divided into PQ and P^2Q^2 partitions (based on Table 8). Thus a sub-interval contains $n/(PQ)$ vertices and a sub-block contains $m/(P^2Q^2)$ edges. The first constraint is that on-chip BRAM size is sufficient to store all K source sub-intervals and destination sub-intervals.

$$M_{bram} \geq 2 \cdot K \cdot \frac{n}{PQ} \cdot S_v \rightarrow \frac{Q}{K} \geq \frac{2 \cdot n \cdot S_v}{P \cdot M_{bram}} \tag{3}$$

The second constraint relies on our vertex index compression method. We use no more than 16 bits to represent a vertex in a sub-interval. Thus a sub-interval contains less than $2^{16} = 65536$ vertices.

$$\frac{n}{PQ} \leq 65536 \tag{4}$$

The third constraint is that on-chip throughput is larger than off-chip bandwidth. We assume that the logic in PEs is designed in a pipelined architecture. Thus a PE can process one edge per clock cycle. The maximal on-chip throughput is $K \cdot S_e \cdot f$.

$$K \cdot S_e \cdot f \geq BW_{mem} \tag{5}$$

Formula (3)$\sim$(5) show the constraints in ForeGraph. Based on these constraints, we calculate the processing time of an FPGA board. Table 2 shows that in ForeGraph $(Q + 2Q^2/K)$ sub-intervals are read from/written to the off-chip memory when processing a block. A board needs to process P blocks in total thus we get $T_{process}$ in Formula (6).

$$T_{process} = \frac{(Q + \frac{2Q^2}{K})P\frac{n}{PQ}S_v}{BW_{mem}} = \frac{n \cdot S_v}{BW_{mem}} \cdot (1 + \frac{2Q}{K}) \tag{6}$$

Edges in P blocks are loaded to the FPGA chip. Thus, we get T_{load} in Formula (7).

$$T_{load} = \frac{\frac{m}{P} \cdot S_e}{BW_{mem}} \tag{7}$$

Intervals on other FPGA boards are transmitted to the board thus we get $T_{transmit}$ in Formula (8).

$$T_{transmit} = \frac{(P-1) \cdot \frac{n}{P} \cdot S_v}{BW_{int}} \approx \frac{n \cdot S_v}{BW_{int}} \tag{8}$$

Based on Formula (6)$\sim$(8), we can get the whole processing time of an FPGA board $T = T_{process} + T_{load} + T_{transmit}$.

5.2 Influence of Parameters on ForeGraph

Substitute Formula (3) into Formula (6), we get Formula (9).

$$T_{process} \geq \frac{n \cdot S_v}{BW_{mem}} \cdot (1 + \frac{4 \cdot n \cdot S_v}{P \cdot M_{bram}}) \tag{9}$$

We simplify $T_{process} \sim T_{transmit}$ and get T in Formula (10).

$$T = T_{process} + T_{load} + T_{transmit} \geq \alpha + \beta \cdot \frac{1}{P} \tag{10}$$

In Formula (10), α and β are two constants, show in Formula (11) and Formula (12).

$$\alpha = \frac{n \cdot S_v}{BW_{mem}} + \frac{n \cdot S_v}{BW_{int}} \tag{11}$$

$$\beta = \frac{4 \cdot n^2 \cdot S_v^2}{BW_{mem} \cdot M_{bram}} + \frac{m \cdot S_e}{BW_{mem}} \tag{12}$$

From Formula (10) we conclude that larger P (using more FPGA boards) leads to better performance in ForeGraph. However, in the real implementation, larger P leads to unbalance problem between partitions and decline of BW_{int}, thus simply increasing P cannot improve the performance when P reaches a threshold. Moreover, Q and K need to meet the condition for equality in Formula (3)$\sim$(5).

5.3 Comparison with Other Systems

We compare ForeGraph with two state-of-the-art FPGA-based large-scale graph processing systems, FPGP [18] and Shijie's work [23] (we call Shi in the following paper). We divide vertices into Q partitions. Both FPGP and Shi store vertices on the chip as many as possible. In FPGP, there is a *Shared-Vertex Memory* (SVM) connected to all FPGAs. Such implementation limits the scalability to multi-FPGA because the total bandwidth of SVM is limited. There are two source buffers on the chip thus it processes $K/2$ partitions equivalently. All edges are read once in FPGP. Shi uses the off-chip memory to store the temporary value of vertices, yet it cannot scale to the multi-FPGA architecture. All edges are read and written once respectively. On-chip BRAMs can store $2K$ partitions. Each partition needs to be read once when writing edge list and read/write once when reading the message list. Moreover, reading and writing of an edge in Shi are attached with a vertex value.

We compare the performance of the three systems on one FPGA board (FPGP does not scale well to multi-FPGA, and Shijie's work does not support multi-FPGA). The comparison result is shown in Table 4. Cells in the gray background show the system with the best performance from one perspective. ForeGraph outperforms the other two systems in terms of minimum data transmitting amount, maximal edges updated per cycle and scalability.

Table 4: Comparison between ForeGraph and other systems

	FPGP [18]	Shijie's work [23]	ForeGraph (ours)
read	$2Q/K \cdot nS_v + mS_e$	$2nS_v + mS_e + mS_v$	$(1 + Q/K)nS_v + mS_e$
write	nS_v	$nS_v + mS_v$	$Q/K \cdot nS_v$
read+write (assuming $m = 10n$ and $Q = 4K$)	$9nS_v + 10nS_e$	$23nS_v + 10nS_e$	$9nS_v + 10nS_e$
edges updated per cycle	two edges per cycle	at most K edges per cycle	K edges per cycle
multi-FPGA scalability	not scale well	no	scale well

6. EXPERIMENTAL RESULTS

Based on the system design and optimization methods, we conduct several experiments using different algorithms on different graphs. We also compare the performance of ForeGraph with state-of-the-art systems in this section.

6.1 Experimental Setup

We evaluate the performance of ForeGraph on the Xilinx Virtex UltraScale VCU110 evaluation platform with an xvcu190 FPGA chip. The target FPGA chip provides 16.61 MB on-chip BRAM resources. We verify the correctness of ForeGraph and get the clock rate as well as resource utilization using Xilinx Vivado 2016.2. All these results are from post-place-and-route simulations. The target off-chip memory is Micron MTA8ATF51264HZ-2G3 SDRAM (2GB, DDR4) and we use DRAMSim2 [32] to simulate the time consumption when accessing off-chip data. The memory runs at 1.2 GHz and provides a peak bandwidth of 19.2 GB/s. We simulate the time consumption of interconnection based on the Microsoft Catapult, it provides a stable latency around 400 ns and bandwidth around 12.25 Gb/s.

Table 5: Properties and acronyms of graphs

	# Vertices	# Edges
com-youtube (YT) [33]	1.16 million	2.99 million
wiki-talk (WK) [33]	2.39 million	5.02 million
live-journal (LJ) [33]	4.85 million	69.0 million
twitter-2010 (TW) [29]	41.7 million	1.47 billion
yahoo-web (YH) [34]	1.41 billion	6.64 billion

To evaluate the performance of ForeGraph, we implement it using three graph algorithms on five real-world graphs. Three graph algorithms include PageRank (PR), Breadth-First Search (BFS) and Weakly Connected Components (WCC). The properties of target graphs are shown in Table 5. The first three graphs can be implemented using one FPGA board while the latter two graphs need to be implemented on the multi-FPGA architecture. We use acronyms for these graphs and algorithms in our experimental results.

6.2 Resource Utilization

We use 8 bits to represent the depth of a vertex in BFS and 32 bits to represent the value of a vertex in PR and WCC. The average width of an edge is 32 bits (16 bits for the source vertex and 16 bits for the destination vertex) using our vertex index compression method. In this way, there are at most 65536 vertices in a sub-block. A sub-block uses 8 bits × 65536 = 64 KB BRAM resources in BFS and 32 bits × 65536 = 256 KB BRAM resources in PR/WCC. We implement 96 PEs when executing BFS and 24 PEs when executing PR/WCC. Detailed resource utilization, as well as clock rate, is shown in Table 6.

Table 6: Resource utilization and clock rate

	BFS	PR	WCC
# PEs	96	24	24
LUT	31.2%	33.4%	35.9%
Register	17.3%	20.6%	19.7%
BRAM	89.4%	81.0%	81.0%
Maximal clock rate	205 MHz	187 MHz	173 MHz
Simulation clock rate	200 MHz	150 MHz	150 MHz

6.3 Execution Time and Throughput

We implement three algorithms (BFS, PR, WCC) on four graphs (YT, WK, LJ, TW). Only one FPGA board is used when processing YT, WK, and LJ, while four FPGA boards are used when processing TW in our simulation.

Table 7: Execution time/throughput of ForeGraph

Algorithm	Graph	Execution time(s)	Throughput (MTEPS)
BFS	YT	0.010	897
	WK	0.027	929
	LJ	0.452	1069
	TW	15.12	1458 (364/board)
PR	YT	0.030	997
	WK	0.052	965
	LJ	0.578	1193
	TW	7.921	1856 (464/board)
WCC	YT	0.016	934
	WK	0.021	956
	LJ	0.307	1124
	TW	24.68	1727 (432/board)

Table 7 shows that the throughput of ForeGraph is around 1000 MTEPS when processing small graphs (e.g. YT, WK, and LJ). When processing larger graphs (e.g. TW) on the multi-FPGAs, the throughput decreases to 400 MTEPS because of the inter-FPGA communication and frequent substitution of on-chip data.

6.4 Benefits of Optimization Methods

6.4.1 Benefits of Vertex Index Compression

In ForeGraph, a vertex is indexed in its corresponding sub-interval using 16 bits. Table 5 shows the number of vertices in each graph with which we can calculate the required bit width to represent a vertex in each graph using the naive coding method. A vertex needs to be represented with $\log_2(1.1 \times 10^6) = 21$ bits in the YT graph and $\log_2(1.4 \times 10^9) = 31$ bits in the YW graph. Thus, ForeGraph reduces the edge data amount by 23.81%∼48.39%.

6.4.2 *Benefits of Edge Shuffling*

ForeGraph processes K source sub-intervals simultaneously and edges in corresponding sub-blocks are loaded to PEs. We use the edge shuffling method to avoid the case that few PEs are overburdened, while others are idle waiting edges. We execute the PR (10 iterations) on four graphs using two different methods, the randomized (randomizing edges in each K sub-blocks) and shuffled order. [1] [2]

Table 8: Benefits of edge shuffling

		YT	WK	LJ	TW
# edges[1]	randomized	2.99m	5.02m	69.0m	1.47b
	shuffled[2]	393m	704m	12.5b	334b
	shuffled	3.77m	5.58m	92.6m	1.72b
edge data increased[2]		131x	140x	181x	234x
edge data increased		1.26x	1.11x	1.34x	1.17x
T_{exe}	randomized	0.041s	0.077s	0.952s	16.55s
	shuffled	0.030s	0.052s	0.578s	7.921s
speedup		1.37x	1.48x	1.65x	2.09x

Table 8 shows the comparison between two edge arrangement methods. In the edge shuffling method, we need to insert several *NULL* edges (Figure 7) thus it increases 1.22x edge data amount on average (If we divide consecutive vertices rather than vertices with a constant stride into an interval, the edge size is 172x on average due to the unbalanced size of each sub-block. The reason is the $power-law$ characteristic of natural graphs, vertices with most in-edges are divided into one interval and sizes of its sub-blocks will be much large than other sub-blocks. Only one PE is working in this situation.). However, such increase of edges can lead to 1.66x performance improvement on average because we balance the workload of different PEs.

6.4.3 *Benefits of Block Skipping*

A source (sub-) interval, as well as corresponding out-edges, is skipped in an iteration when it is not updated in the prior iteration. Such optimization method leads to fewer data transmission between on-chip BRAMs and off-chip memories. We execute the BFS algorithm on four graphs and change the number of (sub-) intervals to show the ratio of transmitted (sub-) intervals/edges in Figure 9.

If we divide graphs into thousands of partitions, only 35%∼50% (sub-)intervals/edges are transmitted using our skipping method, shown in Figure 9. In this way, we reduce the transmitted data amount in ForeGraph.

6.5 Scalability

Section 5 shows that larger P leads to performance improvement. However, the interconnection scheme has the influence on the performance. We simulate four different interconnection schemes, including full interconnection (point-to-point connection), torus interconnection (implemented in Catapult), mesh (each FPGA board is connected to adjacent ones) and bus (all FPGA boards are connected using the bus). In these four schemes, the bandwidth and latency of a physical connection are all set to 12.25 Gb/s and 400 ns. We implement the PR algorithm on TW and YH.

[1] m stands for million and b stands for billion.

[2] We compare two partitioning methods. Results with this footnote divide consecutive vertices into an interval (e.g. $v_1, v_2, v_3 ...$). This partitioning scheme is widely used in many state-of-the-art systems like Gemini [7] and NXgraph [10]. Results without footnote are from our partitioning method which divides vertices with a constant stride into an interval (e.g. $v_1, v_{101}, v_{201} ...$). Actually, the partitioning method is heavily dependent on the given vertex labeling, while natural graphs follow *power-law* and vertices with higher degrees seem to have smaller indexes.

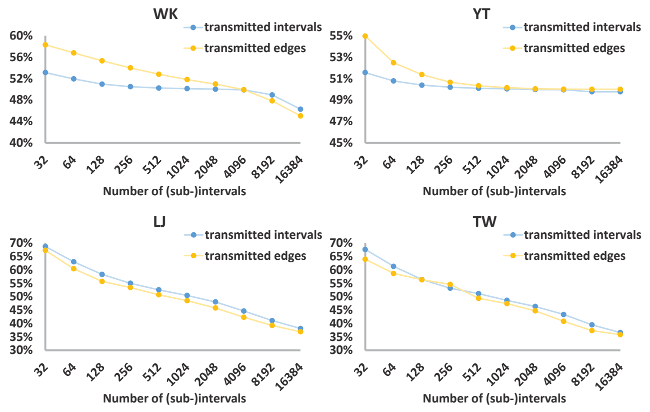


Figure 9: Transmitted (sub-)intervals/edges, executing BFS (varying number of (sub-)intervals).

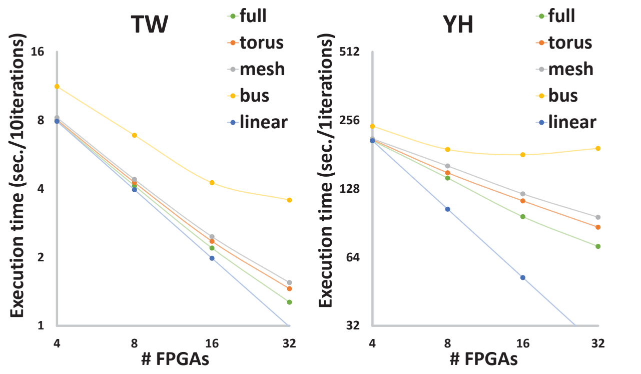


Figure 10: Scalability of ForeGraph.

We add a blue curve which provides a presumptive linear speedup (blue) in Figure 10. Both torus (orange) and mesh (gray) provide comparable performance to the full interconnection scheme (green). The comparison results of blue and green curves show that even the full interconnection cannot provide linear speedup because of the unbalanced workload of different FPGAs. Even so, ForeGraph scales well to the multi-FPGA platform.

6.6 Comparison with State-of-the-art Systems

We compare ForeGraph with state-of-the-art systems in Table 9. ForeGraph outperforms state-of-the-art systems on both execution time and throughput. Experimental results show that ForeGraph achieves 4.54x∼5.04x speedup to CPU-based systems and 8.07x speedup to the FPGA-based system. Moreover, the throughput of ForeGraph is 1.41x∼2.65x larger than previous FPGA-based systems.

7. CONCLUSION

In this paper, we propose a large-scale graph processing system, ForeGraph, based on the multi-FPGA architecture. ForeGraph provides larger on-chip BRAMs size and off-chip bandwidth which are essential to accelerate large-scale graph processing. Partitioning and communication scheme among FPGAs are also considered to ensure locality and reduce conflicts. ForeGraph achieves 5.89x speedup

Table 9: Comparison between ForeGraph and state-of-the-art systems

Algorithm	Graph	Metric	ForeGraph		Comparison system			Improve-ment
			Platform	Performance	System	Platform	Performance	
BFS	TW	execution time(s)	4 FPGAs	15.12	TurboGraph [13]	CPU	76.134	5.04x
BFS	TW	execution time(s)	4 FPGAs	15.12	FPGP [18]	1 FPGA	121.99	8.07x
PR	TW	execution time(s)	4 FPGAs	7.921	PowerGraph [2]	512 CPUs	36	4.54x
BFS	WK	throughput(MTEPS)	1 FPGA	1069	Shijie's work [23]	1 FPGA	657	1.41x
BFS	-	throughput(MTEPS)	4 FPGAs	1458	CyGraph [16]	4 FPGAs	550	2.65x

compared with state-of-the-art designs, and 2.03x average throughput improvement compared with previous FPGA-based systems. Using on-chip BRAMs with random access feature is a promising way to accelerate large-scale graph processing, but it is still limited by the size of BRAMs. Both theoretical analysis and experimental results show that larger BRAM size leads to better performance. In future, it is reasonable to use FPGAs with more on-chip memory resources (e.g. UltraRAM in Xilinx UltraScale$^+$ [35], Altera Stratix 10 using 3D-stacking technology [36], and *etc.*) to achieve a superior performance.

8. ACKNOWLEDGMENT

This work was supported by National Natural Science Foundation of China (No. 61373026, 61622403, 61261160501) and Huawei Innovation Research Program (HIRP). We are also thankful to reviewers for their helpful suggestions.

9. REFERENCES

[1] Graph 500. http://www.graph500.org/.

[2] Je Gonzalez, Y Low, and H Gu. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[3] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[4] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC Disk-based Graph Computation. In *OSDI*, pages 31–46, 2012.

[5] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *VLDB Endowment*, pages 716–727, 2012.

[6] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146. ACM, 2010.

[7] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.

[8] Nadathur Satish, Narayanan Sundaram, Md Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *SIGMOD*, pages 979–990. ACM, 2014.

[9] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471. ACM, 2013.

[10] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. Nxgraph: An efficient graph processing system on a single machine. In *ICDE*, pages 409–420, 2016.

[11] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph : Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *ATC*, pages 375–386, 2015.

[12] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488. ACM, 2013.

[13] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. In *SIGKDD*, pages 77–85. ACM, 2013.

[14] Farzad Khorasani. Scalable SIMD-Efficient Graph Processing on GPUs. In *PACT*, pages 39–50. ACM, 2015.

[15] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *ACM SIGPLAN Notices*, pages 117–128. ACM, 2012.

[16] Osama G Attia, Tyler Johnson, Kevin Townsend, Philip Jones, and Joseph Zambreno. Cygraph: A reconfigurable architecture for parallel breadth-first search. In *IPDPSW*, pages 228–235. IEEE, 2014.

[17] Brahim Betkaoui, Yu Wang, David B Thomas, and Wayne Luk. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *ASAP*, pages 8–15. IEEE, 2012.

[18] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. Fpgp: Graph processing framework on fpga a case study of breadth-first search. In *FPGA*, pages 105–110. ACM, 2016.

[19] Nina Engelhardt and Hayden Kwok-Hay So. Gravf: A vertex-centric distributed graph processing framework on fpgas. In *FPL*, pages 403–406. IEEE, 2016.

[20] Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomas E Uribe, F Thomas Jr, Andre DeHon, et al. Graphstep: A system architecture for sparse-graph algorithms. In *FCCM*, pages 143–151. IEEE, 2006.

[21] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C Hoe, José F Martínez, and Carlos Guestrin. Graphgen: An fpga framework for vertex-centric graph computation. In *FCCM*, pages 25–28. IEEE, 2014.

[22] Tayo Oguntebi and Kunle Olukotun. Graphops: A dataflow library for graph analytics acceleration. In *FPGA*, pages 111–117. ACM, 2016.

[23] Shijie Zhou, Charalampos Chelmis, and Viktor K Prasanna. High-throughput and energy-efficient graph processing on fpga. In *FCCM*, pages 103–110. IEEE, 2016.

[24] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*, pages 105–117. ACM, 2015.

[25] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges In Parallel Graph Processing. *Parallel Processing Letters*, pages 5–20, 2007.

[26] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel graph analytics. *Communications of the ACM*, 59(5):78–87, 2016.

[27] Brahim Betkaoui, Yu Wang, David B Thomas, and Wayne Luk. Parallel fpga-based all pairs shortest paths for sparse networks: A human brain connectome case study. In *FPL*, pages 99–104. IEEE, 2012.

[28] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*, pages 13–24. IEEE, 2014.

[29] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600. ACM, 2010.

[30] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A Bader. Scalable graph exploration on multicore processors. In *SC*, pages 1–11. IEEE, 2010.

[31] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *PACT*, pages 78–88. IEEE, 2011.

[32] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[33] Stanford large network dataset collection. http://snap.stanford.edu/data/index.html#web.

[34] Yahoo! altavisata web page hyperlink connectivity graph, circa 2002. http://webscope.sandbox.yahoo.com/.

[35] https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html.

[36] https://www.altera.com/solutions/technology/next-generation-technology/overview.html.