# Accelerating Iterative Algorithms with Asynchronous Accumulative Updates on FPGAs

Deepak Unnikrishnan, Sandesh Gubbi Virupaksha, Lekshmi Krishnan, Lixin Gao and Russell Tessier
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003

***Abstract*—Iterative algorithms represent a pervasive class of data mining, web search and scientific computing applications. In iterative algorithms, a final result is derived by performing repetitive computations on an input data set. Existing techniques to parallelize such algorithms typically use software frameworks such as MapReduce and Hadoop to distribute data for an iteration across multiple CPU-based workstations in a cluster and collect per-iteration results. These platforms are marked by the need to synchronize data computations at iteration boundaries, impeding system performance. In this paper, we demonstrate that FPGAs in distributed computing systems can serve a vital role in breaking this synchronization barrier with the help of asynchronous accumulative updates. These updates allow for the accumulation of intermediate results for numerous data points without the need for iteration-based barriers allowing individual nodes in a cluster to independently make progress towards the final outcome. Computation is dynamically prioritized to accelerate algorithm convergence. A general-class of iterative algorithms have been implemented on a cluster of four FPGAs. A speedup of 7× is achieved over an implementation of asynchronous accumulative updates on a general-purpose CPU. The system offers up to 154× speedup versus a standard Hadoop-based CPU-workstation. Improved performance is achieved by clusters of FPGAs.**

## I. Introduction

Iterative algorithms form an important workload for distributed computing, including cloud computing. These algorithms generally are structured to progress in a series of iterations where the results of the current iteration are derived from the results of the previous iteration using a fixed set of operations. Although simple, Conway's Game of Life, where the state of a grid cell in a given iteration is based on the states of its neighbors from the previous iteration, provides a familiar example of an iterative algorithm. Many contemporary search and data mining applications use iterative algorithms to refine and process large volumes of data. For example, PageRank [1] is used to refine the rank values of web pages in the world wide web. K-means clustering [2] is an iterative algorithm used to classify data in computational biology.

In most implementations of iterative parallel computation, such as the widely-used MapReduce [3], a master node, distributes input data to multiple worker nodes. Generally, each iteration is scheduled as a separate task and intermediate results from an iteration must be stored and synchronized in a distributed file system (DFS) before the next iteration begins. These types of barriers are particularly problematic for distributed systems for several reasons. First, a node must wait for every other node to finish its task before the next iteration can be scheduled. Synchronization requires repeated disk accesses between successive iterations. Synchronization barriers also introduce periods of traffic bursts within the cluster network, degrading application performance and response time.

In many data mining and machine learning algorithms, a selected portion of the data plays a critical role in the convergence of the overall computation towards the final result. For example, in the iterative formulation of Dijkstra's shortest path algorithm, a solution can be quickly determined by exploring the nodes with the shortest paths from any given node. Unfortunately, software frameworks such as MapReduce and Hadoop lack mechanisms to prioritize the data, restricting the ability to accelerate the computation. Several implementations of MapReduce on FPGAs and GPGPUs [4][5][6] spatially parallelize the computation across multiple hardware resources. Although these frameworks demonstrate improved speedup over previous CPU-based approaches, they inherit several limitations including the need to synchronize iterations using a global interconnect and the absence of mechanisms to prioritize data.

In this paper, we present *Maestro*, a first implementation of an FPGA-based distributed system that uses asynchronous updates [7] to break the synchronization barriers in iterative algorithms. In Maestro, computations for a piece of data progress as soon as an input which affects its value is received, regardless of the iteration in which it was generated. System-wide results are only synchronized at the end of many computations, after the results have converged. Intermediate results are stored in a memory close to the FPGA, eliminating the need for multiple reads and writes to a distributed file system. Further, Maestro dynamically refines the input data set during the computation allowing critical data to receive priority access to the available hardware resources. Our system has been evaluated using three popular iterative algorithms in a laboratory cluster consisting of four Altera DE4 FPGA development boards. Experiments show that our system provides over 2 orders of magnitude speedup over Hadoop and up to 40× speedup over a CPU-based implementation [7] of asynchronous accumulative updates.

## II. Related Work

### A. Previous Iterative Algorithm Implementations

A number of deployments of MapReduce and other implementations of iterative algorithms on distributed hardware have been demonstrated. MapReduce was introduced as a compute

model for FPGAs and GPUs in 2008 [8]. A set of libraries for a heterogeneous system containing a single component of each device was demonstrated. FPMR [5] demonstrated an implementation of iterative algorithms using an FPGA and external DDR memory. FPMR addresses synchronization issues for a single-FPGA system by allowing computation to start as soon as intermediate values are available for a specific element up until an iteration boundary. If the computation requires multiple iterations, data values must be written back to the global memory. Axel [4] is a heterogeneous cluster consisting of FPGAs, GPUs and CPUs which are interconnected using Ethernet links. This paper specifically mentions the challenge of balancing computation across heterogeneous resources to avoid waiting on barriers (Section 6.6, paragraph 2). Mars [6] implements iterative algorithms on GPGPUs. The individual map and reduce tasks, specified using APIs, are assigned to GPU threads. Although these frameworks mark important steps towards integrating special-purpose hardware with existing PC clusters, the need for synchronization barriers in all of them imposes inefficiencies in the distributed iterative compute model.

Several improvements have been proposed to accelerate iterative algorithms [7][9][10] using distributed groups of general-purpose CPUs. iMapReduce [9] transforms the map and reduce tasks into persistent tasks that store intermediate results from successive iterations in memory, eliminating the need for unnecessary reads/writes to the distributed file system. Each worker schedules the reduce phase as soon as intermediate map results for that worker are available, obviating the need for strict synchronization barriers. PrIter [10] identifies a subset of the input dataset that can lead to faster convergence towards the final outcome and performs iterations only on that subset. Maiter [7] proposes a completely asynchronous approach by allowing workers to independently update partitions of the input dataset and propagate these values through asynchronous updates. Although these frameworks greatly improve the efficiency of executing iterative algorithms on general-purpose CPU clusters, the feasibility of such optimizations in special-purpose hardware clusters has not been studied.

### *B. Asynchronous Accumulative Updates*

An iterative computation is performed on a data vector $v = \{v_1, v_2, ..., v_n\}$ by repetitively applying an update function $f$ to the vector $v$ - ie. the values of the data vector $v$ during the $k^{th}$ iteration, denoted by $v^k = \{v_1^k, v_2^k, ..., v_n^k\}$, is computed as

$$v^k = f(v^{k-1}) \tag{1}$$

The results of the current iteration are reused as inputs to the successive iteration until a termination criterion is met.

For example, PageRank is an iterative algorithm that is used to calculate the relative importance of the vertices (web pages) in a graph. The general PageRank algorithm iterates over a web address linkage graph $G(V, E)$, where $V$ represents the webpages (vertices/nodes of the graph), and $E$, the set of hyperlinks between web pages (edges of the graph). An edge exists between nodes $i$ and $j$ if a hyperlink exists from node $i$ to node $j$. To calculate the relative importance of web pages, each node $v$ in the graph is initially assigned an initial PageRank score $R(v) = \frac{1-d}{|V|}$, where $d$ is a constant dampening factor. The page rank of a node in the iteration $i+1$ is successively refined from its previous value in the $i^{th}$ iteration as:

$$R^{(i+1)}(v) = \frac{1-d}{|V|} + \sum_{u \epsilon N^-(v)} \frac{d \times R^{(i)}(u)}{|N^+(u)|} \tag{2}$$

where $N^-$ denotes the set of nodes which have directed edge connections towards node $v$ and $N^+$ denotes the set of nodes that have outgoing edges from node $v$. The iterative computation runs until the the difference in the PageRank values between successive iterations is less than $\varepsilon$.

In the synchronous compute model, the update function $f$ is applied only after a node collects the intermediate results of the previous iteration from *all* compute machines. Assuming that $n$ values are distributed equally among the compute machines, the synchronous approach requires each machine to possess $O(n)$ storage.

In the asynchronous accumulative compute model [7], updates for all $R(u)$ across all iterations are unrolled so that the updates in (2) can happen asynchronously. For example, for PageRank, since $\frac{d}{|N^+(u)|}$ is a constant, $R^{(i)}(u)$ in (2) could be replaced by a function of $i-1$. For example, $\frac{1-d}{|V|} + \sum_{u \epsilon N^-(v)} \frac{d \times R^{(i-1)}(u)}{|N^+(u)|}$, another series of additions, could be used. This type of replacement could be performed repetitively, effectively unrolling the computation until $R^{(\infty)}$, the converged value for a specific web page, is equal to $R^{(0)}$ (initial value) plus a group of summations of partial values. Effectively, these summations can be performed as soon as the input values are available, eliminating the need for iteration-based synchronization barriers. For generality, consider that the value of $R(v)$ at a given point in time is $v$. If a new input value arrives at a compute machine, it does not need to be added to $v$ immediately. Rather, it can be *accumulated* into a partial sum $\Delta v$ which can later be added to $v$.

In this asynchronous accumulate model, each compute node performs two operations:

**Accumulate:** When a compute node receives a message $m$ from any other worker, it is accumulated into a storage location $\Delta v$. The accumulation is specified using an abstract operator $\bigotimes$. Incoming values are accumulated in any order. There is one $\Delta v$ for each data value $v$.

$$\Delta v \leftarrow \Delta v \bigotimes m \tag{3}$$

In the PageRank example, $\bigotimes$ is an addition operation.

**Update:** $\Delta v$ is added to $v$, updating its value and messages are generated for other values which depend on $v$ as an input. The messages are sent to the compute nodes which contain those values. This update operation is performed according to a scheduling policy in three steps: (1) The node adds the accumulated value $\Delta v$ into its current value $v$, (2) an update function **g()** is applied to the *change* in its current value, $\Delta v$,

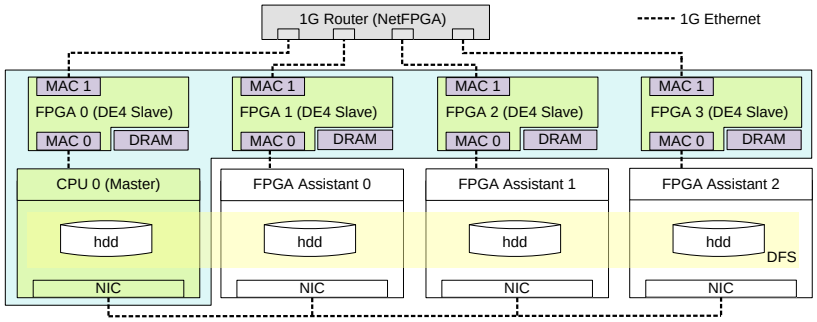


Fig. 1. Cluster setup for a four node $Maestro$ system. Data stored in the distributed file system is processed by four FPGA slaves coordinated by the master CPU node. FPGA assistants help interface the distributed file system (DFS) with the FPGA nodes.

and (3) the node propagates **m=g($\Delta v$)** to all neighboring nodes and resets $\Delta v$.

$$v \leftarrow v \bigotimes \Delta v, \tag{4a}$$

$$\text{if}(\Delta v \neq 0) \text{ send } m = g(\Delta v) \tag{4b}$$

$$\Delta v \leftarrow 0 \tag{4c}$$

For accumulative updates to guarantee correctness, the $\bigotimes$ operator must possess commutative, associative and identity property over $\bigotimes$ and **g()** must possess distributive properties. As an illustration, in the PageRank example, a node *accumulates* PageRank scores received from other nodes for a particular page $j$ in a variable $\Delta R_j$. The current PageRank score for the page $j$ is maintained in $R_j$. When the node updates page $j$, it adds $\Delta R_j$ to $R_j$, propagates the value $\frac{\Delta R_j}{|N(j)|}$ to other nodes and resets $\Delta R_j$ to 0. In this case, **g**($\Delta R_j$) is equal to $d \times \frac{\Delta R_j}{|N(j)|}$.

**Scheduling Updates -** A worker node that owns a partition of the input data set performs updates according to a user-defined scheduling policy. In a round robin scheduling policy, a worker iterates through its data partition updating each value in order one by one. Although simple, the round robin strategy is quite inefficient. For example, if all data receive equal priority, updates may be performed on many values that are insignificant to the overall progress of the computation. In many applications, it is possible to reduce the time to convergence (e.g. fewer iterations/operations) by prioritizing updates for the subset of data with higher importance.

## III. Design

The major contribution of this work is the scalable implementation of asynchronous accumulative updates (AAU) in a compute cluster consisting of FPGAs which contain the parallelism and specialization necessary to accelerate the customized computation versus a CPU-based cluster. The distinguishing features of this system that separate it from previous FPGA and GPGPU implementations of iterative algorithms (e.g. MapReduce and other implementations) include:

**Asynchronous updates:** Each computing node propagates results from its *updates* to other nodes as soon as they are generated without waiting for updates from other nodes. Updates received from other nodes are accumulated at the recipient node. Some updates may be used locally on the node which produces them.

**Scalable FPGA implementation:** A parallel hardware architecture which implements the accumulative-update computing model [7] has been developed and tested. The architecture allows users to scale the performance of individual FPGA nodes as well as the capacity of the cluster by attaching additional FPGA boards to the cluster network.

**Prioritized updates in the hardware implementation:** The intermediate results in our system are stored in DRAM during the computation, eliminating the need for frequent disk accesses. The use of accumulations limits the need to store numerous intermediate values. Effectively, intermediate results are combined using $\bigotimes$ operations (e.g. addition in the PageRank example). Updates are prioritized based on the size of $\Delta v$, where $v$ values with large $\Delta v$ are updated first. Prioritization is performed using a lightweight circuit within the programmable logic.

### A. Computing Cluster

Our *Maestro* asynchronous accumulative update model is implemented on a compute cluster consisting of FPGA worker nodes as shown in Fig. 1. The cluster consists of a single master (CPU 0) workstation and several slave FPGAs interconnected in a LAN configuration. The master is a CPU node responsible for coordinating the tasks running in other slaves and checking for termination conditions. Slaves are built from Altera DE4 development boards each of which includes a Stratix IV EPS230GX device. Slaves run in parallel to execute the computation as tasks and communicate via Gigabit Ethernet links attached to a NetFPGA[1] router in a star topology. The distributed file system (DFS) forms a logical storage that stores the input data used for iterative processing. DFS is implemented as a logical collection of hard drives located at separate workstations.

In order to simplify the process of accessing the distributed file system interface from the FPGA slave, in this prototype implementation each FPGA is attached to a CPU workstation

[1]NetFPGA provides a programmable low-cost 1 Gbps router for our lab prototype. In an actual implementation, the NetFPGA router may be replaced by a commercial off-the-shelf router.

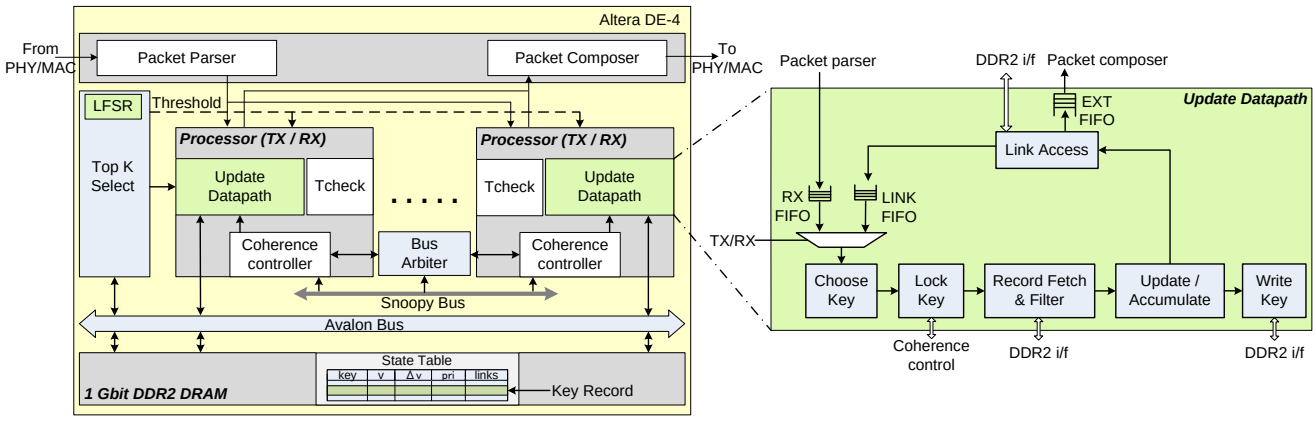


Fig. 2. Implementation of Asynchronous Accumulative Updates on an FPGA Node

(Master or FPGA Assistant in Fig. 1) which manages all distributed file accesses on behalf of the FPGA. Specifically, the CPU is responsible for tasks such as loading the data from the distributed file system into the FPGA, checking for termination conditions and writing the computed results back into the DFS. FPGA assistants exchange information such as termination check information with the master using standardized message passing interfaces (MPI) based on OpenMPI [11]. In future implementations, these functions could be performed by a soft or hard processor implemented on the FPGA. Each FPGA slave node implements a hardware architecture for performing accumulative updates and a network interface for communicating with other worker nodes.

Each data element in the input set (e.g. each web page in the PageRank example) is identified by a unique global key. A hash function of the key is used to make the node assignment. In the current implementation, a simple modulo (MOD) hash function is used, although more efficient functions could be considered in the future. Input data are organized as key-value pairs (KV pairs) and transferred to the appropriate node by the master at the beginning of the computation. A worker stores its partition of input data in **state tables**. The FPGA worker node stores state tables in a 1 Gbit DDR2 DRAM attached to the DE4 board. Messages **m** communicated between nodes during the computation also use a key-value pair structure.

### *B. FPGA Node Design*

The compute architecture in the FPGA slave provides dramatic performance advantages over microprocessor implementation due to customization of both the computation and the communication interface, optimizations that are not possible in a microprocessor or a GPU. The FPGA slave performs update and accumulate operations on a subset of key value pairs assigned by the master node. The architecture is shown in Fig. 2. Two hardware modules, **Packet parser** and **Packet composer**, handle communication with other slaves and the FPGA assistant. The packet parser, built by customizing the receive datapath of a NetFPGA reference router [12], parses incoming Ethernet packets and initiates appropriate actions

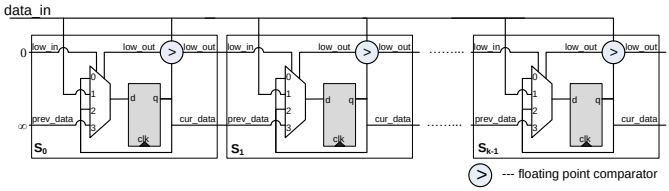


Fig. 3. Threshold selection circuit.

(e.g. load KV pairs into the FPGA, start the computation, etc.). The packet composer, built by modifying the transmit datapath of the NetFPGA reference router, constructs Ethernet packets from outgoing messages. Update/accumulate operations on KV pairs are performed in parallel by several **processors**. Processors access KV pairs from the state table using a shared 32-bit Avalon interconnect. Each processor owns an equal share of KV pairs assigned to the FPGA slave and is responsible for all operations on these KV pairs. Users can vary the number of processors to suit the needs of the specific application. During every iteration, the processor selectively refines KV pairs to prioritize the ones that are more relevant to the overall computation. This is achieved by comparing the priority of each KV pair with a threshold set by the **threshold selection** module. To prevent any memory inconsistencies caused by one or more processors performing update/accumulate operations on the same KV pair, processors negotiate exclusive access to a KV pair using the **coherency controller**.

Next, we discuss each component in greater detail.

*1) **State Table**:* KV pairs are stored using a state table within the DRAM. Since many scientific and web data mining applications involve processing on sparse graphs, the state table is designed to store KV pairs in a memory-efficient fashion. A state table entry is indexed by a hash of the key, and consists of five fields as shown in Fig. 2: the key, its current value ($v$), the accumulated change in the value between two consecutive update operations ($\Delta v$), priority field and the linkage information. The linkage information is a pointer to a linked list of keys whose results depend on the current key (e.g. in PageRank, other web pages which are referenced by the current page). The state of the key including $v$ and $\Delta v$

fields are constantly modified by the update and accumulate operations.

*2) **Threshold Selection**:* Prioritizing the updates to KV pairs during an iteration is critical to accelerating the algorithm convergence. A naive approach to select the $K$ most relevant KV pairs is to simply sort all KV pairs by their priority values and then choose the top $K$ KV pairs for update operations. While this approach is quite simple, it is quite inefficient since all keys must be sorted during each iteration. Instead, *Maestro* uses a threshold-based heuristic. The intuition of this heuristic is that the distribution of priority values in a statistical sample of the KV pairs provides a good approximation of the priority values in the state table. To refine the top $K$ pairs, a small subset of KV pairs ($S$) is randomly sampled. The sample is sorted by the value of priority fields. The *threshold* is set as the priority value of the top $K^{\text{th}}$ KV pair in the sorted sample. The threshold is then used by the processor during every iteration to measure a KV pair's relative importance to the computation. A KV pair is only chosen for update operations if its priority field has a value larger than the threshold.

In the customized FPGA implementation, a modified maximal-sequence linear feedback shift register (LFSR) circuit a length of $n$ bits ($n$ = $\lceil log_2(N) \rceil$, $N$ = number of keys in state table) is used to randomly select $S$ samples from DRAM. As KV pairs are fetched, they are prioritized by a threshold selection circuit, as shown in Fig. 3, using the values in the priority field. The circuit works on the principles of parallel insertion sort. A shift register chain of $K$ cells holds the KV pairs. Each cell stores a KV pair fetched from the DRAM. When a KV pair is read from the DRAM, a floating point comparator in the cell compares the priority field of the incoming key entry with the priority field of the key entry in the register. The $low\_out$ signal indicates whether the stored key's priority is lower than the priority of the incoming key. Additionally, each cell observes the comparison outcome of its left neighbor through the $low\_in$ port. Based on the two comparisons, the cell makes a decision as follows: (1) If the left neighbor's priority field and the cell's own priority are lower than that of the incoming key entry ($low\_in = 1$ and $low\_out = 1$), the cell shifts in the key entry from its left neighbor, (2) if the left neighbor's priority is higher than the incoming key entry's priority and the cell's priority is lower than that of the incoming key entry's priority ($low\_in = 0$ and $low\_out = 1$) the cell replaces its current key entry with the incoming key entry, (3) otherwise, the cell simply retains its current key entry.

*3) **Processor**:* The processor performs update and accumulate operations on a subset of KV pairs assigned to the slave. Each processor can be configured in two modes - transmitter (TX) or receiver (RX). A processor in TX mode performs both update and accumulate operations while a processor in RX mode only performs accumulate operations. The operation mode can be dynamically configured by the user through software configurable registers. Update/accumulate operations on KV pairs are sequenced using a five-stage pipelined datapath in order to maintain high throughput, The coherence controller ensures memory consistency for each key accessed by the processor during update/accumulate operations. The **Tcheck** module computes the progress of computation as measured by the sum of $v$ fields of KV pairs owned by the particular processor. Each processor uses three memory interfaces to access the state table in DRAM. During an iteration, a processor configured in TX mode performs update operations on all KV pairs it owns.

Accumulation messages generated locally or from other workers follow the pipelined datapath except that an Update/Accumulate operation only performs an accumulate on the KV pair. A processor configured in RX mode accepts messages for accumulation from other FPGA slaves via the RX FIFO. A transmitter processor (TX) prioritizes messages for local accumulation over updating new KV pairs.

*4) **Ensuring Memory Consistency during Updates**:* When multiple processors operate on KV pairs resident in a shared global memory, memory inconsistencies can occur due to one or more processors writing to the same KV pair entry. For example, consider two processors, each performing an accumulate and update operation on the same KV pair. While the update operation resets the $\Delta v$ field in the state table entry, the accumulate operation accumulates the incoming message $m$ into the $\Delta v$ field according to ($\Delta v \leftarrow \Delta v \bigotimes m$). Similarly, an inconsistency can also happen from two procesors trying to perform identical operations (update/accumulate) on the same KV pair. To avoid memory inconsistency, all operations on KV pairs must be strictly atomic.

To address this issue, *Maestro* implements a **snoopy coherency protocol** that borrows principles from cache coherency protocols in symmetric multiprocessor systems. The protocol is implemented within the coherence controller block attached to each processor. The snoopy coherency protocol guarantees that simultaneous accesses to the same KV pair are serialized, enforcing strict memory consistency on each KV pair. If accesses do not conflict, update/accumulate operations proceed in parallel in all processors.

*5) **Termination Check**:* Each slave FPGA measures and reports the progress of the local computation to the master node. Progress is defined as the sum of $v$ fields for all keys in the state table. Since update and accumulate operations are cumulative over the $v$ field of the KV pair entry, the rate of progress monotonically increases or decreases over time. Within the slave FPGA, **TCheck** modules attached to each processor compute local progress during every iteration. The results are aggregated and made available to the packet composer, which when requested, sends the estimated progress to the master node.

### *C. System Scalability*

The computing capacity can be scaled by adjusting the number of TX/RX processors within each FPGA or by attaching several FPGAs in a multi-node cluster configuration. In multi-node configurations, at least one processor must be configured as a receiver processor to process update messages from other slaves. The number of transmitter and receiver processors can

TABLE I
LIST OF ITERATIVE ALGORITHMS

| **Algorithm** | **Init**$_j$ | $\mathbf{g_j(x)}$ | $\bigotimes$ |
|---|---|---|---|
| Connected | $j$ | $x \cdot \Delta_j$ | $max$ |
| PageRank | $1-d$ | $d \cdot \frac{x}{\lvert N(j)\rvert} \cdot \Delta_j$ | $+$ |
| Katz metric [13] | 1 ($j$ = $s$) or 0 ($j \neq s$) | $\beta \cdot x \cdot \Delta_j$ | $+$ |

be dynamically varied by the user to suit the requirements of the application through software configurable registers.

## IV. CLUSTER OPERATION

To parallelize an iterative algorithm using *Maestro*, a user must specify three components: a data partitioner, an iteration kernel, and a termination checker. These interfaces are sufficiently general to describe any algorithm which meets the asynchronous accumulative update criteria described in Section II-B. The **partitioner** specifies the criterion to assign the keys to workers (e.g. the MOD operation in the PageRank example). The partitioner reads input key-value pairs from a file and assigns them to the individual worker nodes. The **iteration kernel** specifies the accumulate and update operations and the initial values for the keys. These operations are described as Verilog templates. The **termination checker** component is used to describe the criterion which must be satisfied to terminate the iterative computation.

The CPU node designated as the master (e.g. CPU 0 in Fig. 1) runs the partition function to distribute the input data according to the hash function specified in the partitioner. The computation executes in every worker in three steps. The master instructs all workers to load the data partitions from the local file system into the DRAM-based state tables. The FPGA assistant converts partition data into packets and sends them over to the FPGA. Slaves start the iterative computation process and exchange messages via Gigabit Ethernet links attached to 1G NetFPGA reference router. To amortize the communication cost of sending a KV pair outside a slave, messages are aggregated until there are enough to fill the maximum transmission capacity of an Ethernet frame (150 key-value pairs). The total progress in slaves is checked periodically (e.g. every 4 seconds) by the master. Once terminated, the results of the computation are retrieved by FPGA assistants from the slave nodes.

## V. EXPERIMENTAL APPROACH

**Setup:** To evaluate Maestro, we implemented a compute cluster with four CPUs and four FPGA nodes. Each CPU node has an Intel Core2 Quad processor running at 2.44 GHz with 4 GB RAM and an attached 1 Gbit/s network interface card. For Maestro cluster experiments, we fix the sampling size ($S$) as 1024, threshold selection circuit size ($K$) as 128. A termination check is performed by the master node every 4 seconds. The FPGA operates at a frequency of 125 MHz.

**Algorithms:** For each algorithm, Table I specifies the initial value for the $j^{th}$ key ($Init_j$), update ($g_j(x)$) and accumulate ($\bigotimes$) operators. The objective of the connected components (Connected) algorithm is to find all the connected nodes in a graph. In the iterative formulation of this algorithm, the $j^{th}$ key is initialized to a unique ID ($Init_j = j$). Next, every key propagates its ID to all its neighbors ($g_j(x) = x \cdot \Delta_j$). When a key receives an ID, it compares its ID with the incoming ID and chooses the maximum of the two ($\bigotimes = max$). Katz metric [13] finds the proximity measure of two nodes in a graph. It is computed as the sum over all the paths between two nodes exponentially dampened by the path length. In the iterative formulation of Katz, a key chosen as the source node (s). The source node is assigned an initial value of 1. All other nodes are initialized to 0. During an iteration, every key node multiplies its current value by a constant dampening factor $\beta$ and propagates the result to other nodes ($g_j(x) = \beta \cdot x \cdot \Delta_j$). When a key receives a message, it accumulates the message ($\bigotimes = +$).

TABLE II
SPEEDUP OF MAITER VERSUS HADOOP FOR 1, 2, AND 4 PROCESSOR CLUSTERS

| Configuration | Cluster | Graph | Execution time (sec) | | Speedup |
|---|---|---|---|---|---|
| | | | Hadoop | Maiter | |
| PageRank | 1 | 1.3M | 2505 | 114 | 22 |
| | 2 | 2.6M | 3639 | 467 | 8 |
| | 4 | 5.2M | 6673 | 717 | 9 |
| Katz | 1 | 1.3M | 4200 | 137 | 31 |
| | 2 | 2.6M | 4707 | 412 | 11 |
| | 4 | 5.2M | 10741 | 563 | 19 |
| Connected | 1 | 1.3M | 500 | 29.2 | 17 |
| | 2 | 2.6M | 1115 | 66 | 17 |
| | 4 | 5.2M | 1695 | 121 | 14 |

To evaluate Maestro, we also implement the three algorithms in Table I using Hadoop, an open source implementation of MapReduce [3], and Maiter frameworks, in addition to our heterogeneous system. Hadoop requires the use of strictly synchronous barriers and disk writes between successive iterations while Maiter provides an implementation of the asynchronous accumulative update-based computing model only using general-purpose CPUs. Both Hadoop and Maiter use a single-core, single-threaded implementation. Maestro improves Maiter by parallelizing the update and accumulate process on the FPGA and specializing the computation.

Evaluation is performed using graphs where in-degrees follow a log-normal distribution with parameters ($\sigma = 0.5$, $\mu = 2.3$). Graphs are sized to nearly fill the capacity of 1 Gbit DRAM memory on the Altera DE4 board.

## VI. EVALUATION

### A. Running time

In an initial experiment, we compare the execution time[2] of the asynchronous accumulative update based computing model implemented on a single FPGA versus a Maiter implementation on a general-purpose CPU. To illustrate the state-of-the-art nature of Maiter, the speedup of Maiter on up to four microprocessors versus a standard Hadoop implementation on up to four microprocessors is shown in Table II. Maiter executes 22×, 31× and 17× faster than the Hadoop version

[2]The time to load data into the FPGA slave is negligible relative to the computation time and hence it is ignored in the calculation of execution time.

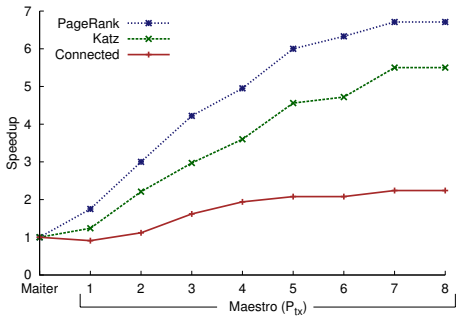


Fig. 4. Speedup of Maestro (1 FPGA) versus Maiter (1 microprocessor). The Maestro configuration is adjusted from 1 to 8 processors inside 1 FPGA.

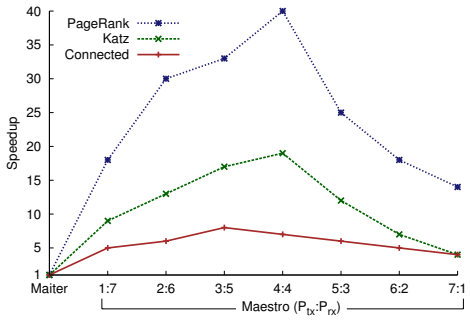


Fig. 5. Speedup of Maestro versus Maiter on 4 nodes for different transmitter to receive processor configurations. Graph size=5.2 million nodes

for PageRank, Katz and Connected benchmarks. The speedup results from the removal of synchronous barriers and disk writes between iterations. Our FPGA implementation makes further dramatic improvements on this Maiter speedup by using FPGA parallelism and specialization.

Fig. 4 shows the speedup of executing the three benchmarks on one Maestro FPGA node normalized against the execution time on Maiter on a single microprocessor. The input dataset is a 1.3 million node graph (900MB). In the experiment, the number of transmitter processors in the Maestro FPGA ($P_{tx}$) is varied from 1 to 8. With one transmitter processor ($P_{tx}$=1), Maestro is 77% faster than Maiter (39× faster than Hadoop) for the PageRank benchmark. Speedup linearly scales as more processors are added to the FPGA. With eight processors in the FPGA, Maestro executes approximately 7× faster than Maiter (154× faster than Hadoop). The Katz benchmark executes approximately 6× faster than Maiter on eight processors (186× faster than Hadoop). Connected components is a relatively low compute intensive application ($g_j(x) = x \cdot \Delta_j$) which yields only a modest speedup of 2.2× versus Maiter (38× vs Hadoop) with eight processors in the FPGA. In general, the performance gap between CPU and the FPGA implementation grows with the complexity of accumulate and update operations.

### B. Scalability

*1) Two nodes:* A two-FPGA cluster is setup according to the topology in Fig. 1. Each FPGA in the cluster includes eight processors. The ratio of transmitter to receiver ($P_{tx}$:$P_{rx}$) processors in the design is dynamically varied during the experiment. For each application, the problem size is doubled from that of the one node experiment (2.6 million nodes). The workload is evenly divided between all slaves using the MOD partition function. For comparison, Maiter and Hadoop are executed on two CPU workstations interconnected in a LAN configuration. We find that a balanced ratio of transmitter to receiver processors (4:4) yields the highest speedup in all benchmarks (26×, 16× and 4.1× for PageRank, Katz and Connected versus Maiter, or a speedup of 208×, 176× and 69.7× versus Hadoop). When the $P_{tx}$:$P_{rx}$ ratio is increased further, higher update rates and lower accumulation rates cause RX FIFOs in Fig. 2 to overflow. Thus, packet transmission must be throttled, reducing application performance.

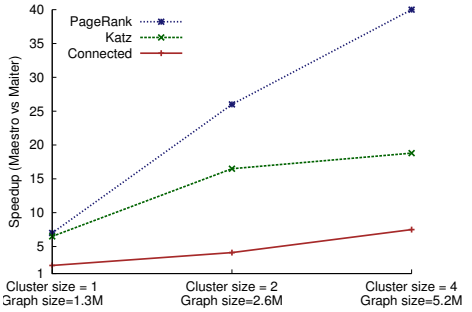


Fig. 6. Best case speedup of Maestro versus Maiter for different cluster configurations

*2) Four nodes:* For the four-node cluster, the problem size is doubled from that of the two-node experiment (5.2 million nodes). Maiter runs on four CPU workstations interconnected in a LAN configuration. Fig. 5 shows speedup of Maestro for different processor configurations versus Maiter. For the PageRank benchmark, Maestro is 18× faster than equivalent Maiter implementation when each FPGA is configured with one transmitter and seven receiver processors. As more processors are converted to transmitters, the speedup improves. With four transmitter and receiver pairs in every FPGA, the four-node Maestro executes 40× faster than the Maiter implementation and 360× faster than Hadoop. Speedup drops when transmitters exceed receivers.

Fig. 6 summarizes the best case speedup with Maestro in a scaling problem size/cluster configuration. The FPGA in the one-node Maestro cluster includes eight transmitter processors. For two- and four-node Maestro configurations, each FPGA was programmed with four transmitter and four receiver processors. In general, for all benchmarks Maestro demonstrates better speedup with larger problem sizes and cluster configurations. In the four-node configuration, Maestro offers 40×, 18.7× and 7.5× speedup versus Maiter for PageRank, Katz and Connected benchmarks.

### C. Resource consumption

Table III provides the logic and memory utilization of an eight-transmitter processor Maestro system on a Stratix IV FPGA. Each processor in our system requires 3,256 ALUTs and 3,375 registers. Table IV compares the energy

TABLE III
RESOURCE UTILIZATION ON A STRATIX IV EP4SGX230 FPGA

| Resource | Overall system | Resources per processor |
|---|---|---|
| Combinational ALUTs | 64,178 (35%) | 3,256 (1.7%) |
| Registers | 70,299 (39%) | 3,375 (1.8%) |
| Memory bits | 1,621,781 (20%) | 4,110 (0.03%) |

consumption and cost of executing three benchmarks in a four-node cluster using Hadoop, Maiter and Maestro frameworks. For these comparisons, we assume that a CPU workstation costs $500 and consumes about 120W. Each Altera DE4 board costs $3,000 and consumes approximately 10W power when attached to a x1 PCIe slot. Maestro consumes 238-342× less energy in comparison to Hadoop for PageRank and Katz for a 7× increase in the total system cost. Energy savings of approximately 35× and 13× are observed for these applications versus Maiter. As mentioned in Section III, the FPGA Assistant CPUs are provided in this experimentation for prototyping. These processor-based assistants could be replaced by FPGA-based soft processors. Hence, they have been omitted from the energy and cost analysis.

### D. Fixed problem size

Table V provides the total number of iterations and exection times $T_p$ for different problem sizes on a $p$ = 1, 2, and 4 FPGA Maestro configuration for PageRank. Each FPGA implementation has one transmitter and seven receivers. A linear speedup is observed when additional FPGAs ($p$ = 1-4) are used to solve a fixed size problem (e.g. problem size = 1200k). Each FPGA holds fewer state table entries as the problem is parallelized, resulting in a lower threshold for KV pair selection in larger cluster configurations. The drop in the threshold causes an overall increase in the number of iterations required to finish the computation.

### E. Comparison to Previous Work

FPMR [5] reports a speedup of 33.5× versus a CPU implementation of MapReduce for RankBoost, a machine learning application to rank web documents. In contrast, our implementation of PageRank, a similar machine-learning application demonstrates a speedup of 154× versus Hadoop on one FPGA. Further, Maestro can be scaled to yield higher speedups in larger configurations (up to 360× speedup for PageRank on a four-node system). Mars [6], an implementation of MapReduce on an NVIDIA G80 GPGPU, demonstrates a speedup of 5× versus a MapReduce implementation on an Intel Quad-core processor. Our work improves the speedup and scalability from these previous synchronous implementations by applying asynchronous accumulative updates and prioritized data refinement.

## VII. CONCLUSION

In this paper we have presented $Maestro$, an FPGA-based distributed system that utilizes asynchronous accumulative updates to execute iterative algorithms. This approach addresses the synchronization issue often found in distributed systems. Our work maps this approach to FPGA-based distributed systems, simplifying system scalability and demonstrating significant speedups due to FPGA parallelism and specialization. Prioritized computations accelerate algorithm convergence through dynamic data refinement. In the future, we plan to investigate better data partitioning strategies for $Maestro$. An open source implementation of $Maestro$ is available from https://github.com/deepakcu/maestro.

TABLE IV
ENERGY/COST ESTIMATE FOR 4 NODE CLUSTER

| Configuration | Energy (KWh) | | | Cost |
|---|---|---|---|---|
| | PageRank | Katz | Connected | |
| Hadoop | 0.89 | 1.43 | 0.23 | $2,000 |
| Maiter | 0.095 | 0.075 | 0.016 | $2,000 |
| Maestro | 0.0026 | 0.006 | 0.0023 | $12,000 |

TABLE V
ITERATIONS AND EXECUTION TIMES FOR PAGERANK

| Problem size (N) | Iterations (I) | | | $T_p$(sec) | | |
|---|---|---|---|---|---|---|
| | p=1 | p=2 | p=4 | p=1 | p=2 | p=4 |
| 200k | 239 | 291 | 795 | 10 | 5 | 2 |
| 400k | 155 | 206 | 462 | 20 | 9 | 5 |
| 600k | 131 | 181 | 232 | 30 | 12 | 6 |
| 800k | 115 | 162 | 201 | 40 | 19 | 8 |
| 1000k | 105 | 120 | 179 | 49 | 20 | 9.5 |
| 1200k | 97 | 119 | 145 | 60 | 25 | 11 |

## VIII. ACKNOWLEDGMENTS

This research was funded under NSF grant CNS-0831940. We thank Altera Corporation for the donation of the DE4 boards and supporting tools.

## REFERENCES

[1] S. Brin and L. Page, "The Anatomy of a Large-scale Hypertextual Web Search Engine," in *Proc. Int'l Conf. on World Wide Web*, Apr. 1998.
[2] N. Slonim, N. Friedman, and N. Tishby, "Unsupervised Document Classification using Sequential Information Maximization," in *Proc. ACM SIGIR Int'l Conf. on Research and Development in Information Retrieval*, 2002, pp. 129–136.
[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Comm. of the ACM*, vol. 51, pp. 107–113, Jan. 2008.
[4] K. H. Tsoi and W. Luk, "Axel: A Heterogeneous Cluster with FPGAs and GPUs," in *Proc. ACM/SIGDA Int'l Symp. on FPGAs*, 2010.
[5] Y. Shan, et al., "FPMR: MapReduce Framework on FPGA," in *Proc. ACM/SIGDA Int'l Symp. on FPGAs*, 2010, pp. 93–102.
[6] B. He, et al., "Mars: A MapReduce Framework on Graphics Processors," in *Proc. ACM Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2008, pp. 260–269.
[7] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Accelerate Large-scale Iterative Computation Through Asynchronous Accumulative Updates," in *Proc. Workshop on Scientific Cloud Computing*, 2012, pp. 13–22.
[8] J. Yeung, et al., "Map-reduce as a programming model for custom computing machines," in *Proc. IEEE Int'l Symp. on FCCMs*, Apr. 2008, pp. 149–159.
[9] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "iMapReduce: A Distributed Computing Framework for Iterative Computation," in *Proc. IEEE Int'l Symp. on Parallel and Distributed Proc.*, 2011, pp. 1112–1121.
[10] ——, "PrIter: A Distributed Framework for Prioritized Iterative Computations," in *Proc. ACM Symp. on Cloud Computing*, Oct. 2011.
[11] "OpenMPI," http://www.open-mpi.org/.
[12] J. Naous, et al., "NetFPGA: Reusable Router Architecture for Experimental Research," in *Proc. ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, Aug. 2008, pp. 1–7.
[13] L. Katz, "A new status index derived from sociometric analysis," *Psychometrika*, vol. 18, pp. 39–43, 1953.