Exploring DataVortex Systems for Irregular Applications

Roberto Gioiosa, Antonino Tumeo, Jian Yin, Thomas Warfel
Pacific Northwest National Laboratory
{roberto.gioiosa, antonino.tumeo, jian.yin, thomas.warfel}@pnnl.gov

David Haglin
Trovares Inc.
david@trovares.com

Santiago Betelu
Data Vortex Technologies Inc.
santiago.betelu@datavortex.com

***Abstract*—Emerging applications for data analytics and knowledge discovery typically have irregular or unpredictable communication patterns that do not scale well on parallel systems designed for traditional bulk-synchronous HPC applications. New network architectures that focus on minimizing (short) message latencies, rather than maximizing (large) transfer bandwidths, are emerging as possible alternatives to better support those applications with irregular communication patterns. We explore a system based upon one such novel network architecture, the Data Vortex interconnection network, and examine how this system performs by running benchmark code written for the Data Vortex network, as well as a reference MPI-over-Infiniband implementation, on the same cluster. Simple communication primitives (ping-pong and barrier synchronization), a few common communication kernels (distributed 1D Fast Fourier Transform, breadth-first search, Giga-Updates Per Second) and three prototype applications (a proxy application for simulating neutron transport-"SNAP", a finite difference simulation for computing incompressible fluid flow, and an implementation of the heat equation) were all implemented for both network models. The results were compared and analyzed to determine what characteristics make an application a good candidate for porting to a Data Vortex system, and to what extent applications could potentially benefit from this new architecture.**

***Index Terms*—Data Vortex; irregular application; high-performance computing;**

I. Introduction

Knowledge-discovery applications in many fields (e.g. bioinformatics, cybersecurity, and machine learning), and the analysis of complex social, transportation, or communication networks typically employ algorithms processing enormous datasets that require large-scale clusters to provide sufficient memory and the necessary performance. These algorithms typically use data structures built on pointers or linked-lists (such as graphs, sparse matrices, unbalanced trees, or unstructured grids) that result in irregular, asymmetric, and unpredictable patterns for data access, control flow, and communications [1]. Data can potentially be accessed from any node with transaction sizes of only a few bytes, and very little computation is performed per access, making it difficult to partition datasets across nodes in a load balanced manner. Current clusters and supercomputers have been optimized for scientific simulations where communications are generally predictable or can be efficiently interleaved with computations. Their processors implement advanced cache hierarchies that exploit spatial and temporal locality to reduce access latencies and quickly execute regular computation patterns. Their network interconnects are optimized for bulk-synchronous applications characterized by large, batched data transfers and well-defined communication patterns. Clusters optimized for traditional HPC applications rarely demonstrate the same scalability for irregular applications.

Several custom large-scale hardware designs (e.g., Cray XMT and Convey MX-100) [2], [3] have been proposed to better cope with the requirements of irregular applications but are too expensive for general use. Software runtime layers (e.g., GMT, Grappa, Parallex, Active Pebbles) [4], [5], [6], [7] have also looked into better support requirements of irregular applications by exploiting available hardware resources. However, software runtime layers cannot fully compensate for network limitations. Conventional network interconnects only achieve high bandwidths with large data transfers, but are easily congested with unpredictable data transfers and remain one of the key limitations for efficiently executing these applications on current cluster architectures.

In this paper, we evaluate a full implementation of a novel network design, the Data Vortex architecture, in a real cluster system. We perform our evaluation from the viewpoint of a user wanting to explore a new system and understand whether an eventual porting of existing applications or design of new applications on this platform will bring performance and/or programming benefits. The Data Vortex architecture was originally proposed as a scalable low-latency interconnection fabric for optical packet switched networks [8], [9]. Designed to operate with fine-grained packets (single word payloads) to eliminate buffering, and employing a distributed traffic-control mechanism [10], the Data Vortex switch is a self-routed hierarchical architecture that promises congestion-free scalable communication. The initial analysis work on the optical version of the switch was limited to verifying the feasibility of the design for optical networks [11], verifying its reliability [12], [13], and verifying its robustness with a variety of synthetic traffic patterns [14], [15]. The architecture that we evaluate is built on an electronic implementation of the Data Vortex Switch [16]. Each node in the cluster contains a Vortex Interface Controller (VIC) PCI Express 3.0 card, and is accessed through a dedicated Application Programming Interface (API). Each VIC connects its node with the switch and provides hardware acceleration for certain API functions,

including a dedicated memory to collect and prepare network packets, and a Direct Memory Access (DMA) controller for moving data across the PCI Express bus between the node and the VIC. The architecture is fully functional.

Applications must be adapted to fully exploit the special features of an alternative network; in some cases the algorithm must be restructured. As we show in this work, several irregular algorithms easily map onto the Data Vortex programming model and achieve considerable performance improvement relative to a Message Passing Interface (MPI) implementation over Fourteen Data Rate (FDR) Infiniband, despite having a slower physical interconnect. Other algorithms show limited or no improvement compared to the reference MPI implementation. One of our key contributions is describing which algorithmic and implementation characteristics make an application a good candidate for improving performance on a Data Vortex system.

We performed our experiments on a 32-node cluster; each node has dual Intel E5-2623v3 Haswell-EP processors (2 CPUs, 4 cores/CPU, 2 threads/core), 160 GB of main memory, and both Data Vortex and Infiniband NICs. We ran a variety of benchmarks including low-level communication tests (ping-pong messaging, global barriers), basic kernels ("Giga Updates Per Second" (GUPS), an FFT, and Breadth-First Search (BFS)), and three representative applications (the SNAP mini-app,[1], an implementation of the heat equation, and a finite difference solver for modeling 2D incompressible fluid flows). To provide an "apples-to-apples" comparison, we compare results from our Data Vortex implementations to MPI implementations of the same algorithms running on the same cluster, but using a conventional MPI-over-Infiniband implementation. We show that traditional applications that are regular or that can be "regularized" through message destination aggregation show little to no performance improvements on the DataVortex network compared to MPI-over-Infiniband, but that irregular applications with small computation-to-communication ratios show considerable performance improvement when running on the Data Vortex network.

In summary, this paper makes the following contributions:

- We analyze the Data Vortex hardware and software architecture;
- We perform an analysis of the performance improvement provided by Data Vortex system for data analytics and irregular applications;
- We highlight which characteristics make an application a good candidate for improving performance on Data Vortex systems.

The rest of this paper is organized as follows. Section II and III describe the Data Vortex Switch and the VIC, and the programming model of the system. Section IV details our system setup and provides information about the benchmarks used. We show the test results of our evaluation in Sections V, VI, and VII. Finally, Section X concludes this work.

[1]http://portal.nersc.gov/project/CAL/designforward.htm#SNAP

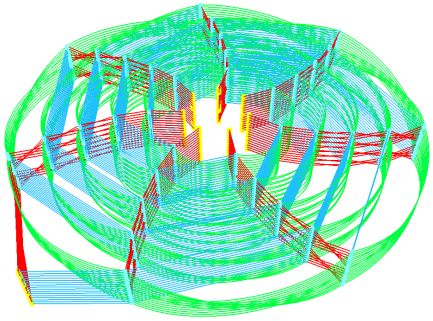


Fig. 1: Data Vortex Switch. Normal paths, connecting nodes from one cylinder to a nested one at the same height and, rotationally, an element (angle) forward, are denoted in blue. Deflection paths, connecting nodes in the same cylinder, an angle forward, are denoted in green. Deflection signal lines are denoted in red.

II. Data Vortex network architecture

The Data Vortex Network architecture is a congestion-free packet-switched network. All packets have a 64-bit header and carry a 64-bit payload. The system is composed of two interacting elements: the Data Vortex Switch and the Data Vortex VICs (Vortex Interface Controller - the network interface). The Data Vortex switch implements a high-radix interconnect that routes packets using a data flow technique based on packet timing and position. The switching control is distributed among multiple switching nodes in the structure, avoiding a centralized controller and limiting the complexity of the logic structures. The interconnect structure implements a *deflection* or *hot potato* design, whereby a message packet is routed through an additional output port rather than being held in storage buffers until a desired output port is available. The current implementation is an electronic version based upon an earlier optical switch design.

The Data Vortex switch is organized as a multilevel structure; each level is composed of a richly interconnected set of rings. With evenly-spaced levels and switching nodes on each ring, the interconnect forms a three-dimensional cylindrical structure composed of multiple nested cylinders, as shown in Figure 1. Each switching node has a location identified by three-dimensional cylindrical coordinates: radius, rotation angle, and height. The radius identifies the exact cylinder (routing level), the rotation angle identifies the position of the switching node among all the switching nodes in the same cylinder circumference, and the height is the position of the node among the nodes along the same cylinder height. Denoting with C the overall number of cylinders, with H the number of nodes along the cylinder height, and with A the number of nodes along the cylinder circumference, a node is identified by the triplet (c, h, a). C scales with H as $C = \log_2 H + 1$, and the number of nodes per cylinder is $A \times H$. Packets are injected into the nodes of the outermost cylinder, and are ejected from the nodes of the innermost cylinder. Therefore, a data vortex network with $N_t = A \times H$ input and output ports implements

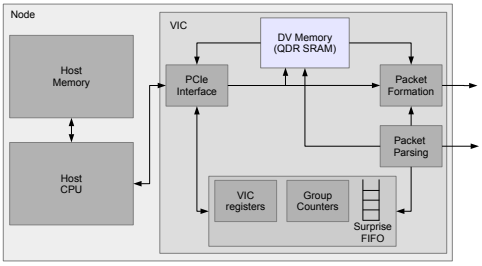


Fig. 2: The Vortex Interface Controller (VIC). The VIC interfaces the host to the Data Vortex Switch providing 32 MB of SRAM (DV Memory) and additional features such as the group counters and a FIFO queue for Surprise packets.

$N = A \times H \times C = A \times H \times (\log_2 H + 1)$ nodes, and the number of nodes scales with the number of ports as $N_t \log_2 N_t$. Normal paths connect nodes of the same height in adjacent cylinders, while deflection paths connect nodes of different heights within the same cylinder. When packets enter a node with coordinates (c, h, a), the c-th bit of the packet header is compared with the most significant bit of the node's height coordinate (h). If the bits are equal, the packet is routed to a node in an inner ring, and the rotation angle increments (i.e., at angle $a + 1$ in the inner cylinder circumference). If not, it is routed with the same cylinder and rotationally incremented at a different height. Once a packet reaches the innermost cylinder (at the desired height), it is then routed around to the target output port according to the angle. Contention is resolved using deflection signals. A node has two inputs: a node in the same cylinder and a node on the outer cylinder. Whenever this node receives a packet from the node on the same cylinder, the sending node also propagates a deflection signal that blocks the node on the outer cylinder from sending new data. Thus, contention is resolved by slightly increasing routing latency (statistically by two hops) without need for buffers.

Past research [9] has explored the use of the Data Vortex topology for optical networks, leading to prototypes with up to 12 input and 12 output ports [11]. Performance studies with synthetic and realistic traffic patterns showed that the architecture maintained robust throughput and latency performance even under nonuniform and bursty traffic conditions due to inherent traffic smoothing effects [14], [15]. Our system's Data Vortex Switch is an electronic (not optical) implementation on a set of Field Programmable Gate Arrays (FPGAs).

The Vortex Interface Controller (VIC) is a custom Network Interface Controller for interfacing a cluster node to the Data Vortex Switch; each cluster node has at least one VIC. Figure 2 shows the block diagram of the VIC. The current VIC is implemented as a PCI-Express 3.0 board that integrates an FPGA with 32 MB of Quad Data Rate Static Random Access Memory (QDR SRAM), also called "DV Memory". DV Memory is directly addressable for reading and writing from both the network and from the cluster node (across the PCI Express bus). While primarily used to buffer incoming network data, it also allows outgoing packet headers and payloads to be pre-cached across the PCI Express bus, as well as storing user-level data structures. Because every VIC can address every DV Memory location (local or remote) with the combination of VIC ID and DV Memory address, the DV Memory can also be used as a globally-addressable shared memory. The API allows a programmer to specify where the header and payload come from; the VIC FPGA handles the logic for getting packets to and from the switch. In addition to the control logic for the DV Memory interface, the VIC also provides hardware support for two other API features: (packet) group counters, and a network-addressable First In First Out (FIFO) input queue. Group counters provide a means of counting how many data words within a particular transfer are yet to be received by the receiving VIC. Network transfers occur as individual 8-byte words, and order of arrival is not guaranteed. To know when a transfer has completed, the application programmer must first declare the specific number of words that will be received in a transaction by writing that value into a group counter before the first data packet arrives. Incoming packets for that transfer must explicitly refer to that particular group counter, and as they arrive, the counter decrements toward zero. An additional API call allows a program to wait until a specific group counter reaches 0, or a timeout expires. The current VIC implementation provides up to 64 group counters in the current FPGA; one of these is presently reserved as a *scratch* group counter for those situations where the counter does not need to be checked, and another 2 are reserved for a group barrier synchronization implementation.

The FIFO queue (also referred to as the "surprise packet" queue in the Data Vortex API) allows each node within the cluster to receive and buffer thousands of 8-byte messages until ready to deal with them. DV Memory slots only store single words, and only the last-written value can be read. Specific addresses must be coordinated by the sender and receiver in advance, and multiple nodes sending to the same destination DV Memory slot must be coordinated to avoid overwriting data. The FIFO queue provides for incoming unscheduled ("surprise") packets to be non-destructively buffered, although message ordering across the network, even when sent from the same node, is not guaranteed. The developer is responsible for polling the status of the queue to know when messages have arrived, and for handling the potential out-of-order messages when reading from the queue. The VIC also provides two Direct Memory Access (DMA) engines for transferring data between the host memory, the VIC Memory, and the network.

III. Programming Model

The Data Vortex API is a low-level interface to the VIC components enabling multiple ways of sending packets across the network to any individual VIC, including your own. Each packet is composed of a 64-bit header and a 64-bit payload. The header specifies the destination VIC, an optional destination group counter, and and an address within the destination VIC. The destination address at the destination

VIC can specify a VIC DV Memory slot, the "suprise packet" FIFO, or a specific group counter. Individual packets (header plus payload) can be assembled within host memory and sent directly from host memory to the network; when sending multiple packets, one can either DMA headers and payloads from host memory, pre-cached headers from DM Memory and payload from host memory, or headers from host memory and pre-cached payload from DV Memory. Upon arrival at the destination, the header is normally discarded and the sent value is written to the appropriate address. A few special headers allow sending a "return header" as the payload, and also allow a particular query value to be encoded in the header. The receiving VIC reads the requested location encoded in the header, uses the original payload as the header for a new "reply" message, and uses the requested value as the payload for the new message. The "reply" destination VIC does not need to be the same as the original sending VIC.

Group counters are globally accessible, meaning that each VIC can set not only his own group counters, but also the group counters of other VICs. While this could potentially allow a sending VIC to set the destination group counter before sending a data transfer, the out-of-order packet arrival means that the first data packet could arrive before the "set group counter" control packet reaches the VIC. Even though the transfer would complete, the destination VIC group counter would never reach zero because the initial packet arrival is lost when the "set group counter" control packet arrived. Typically the developer will set up the communication by presetting a group counter to the number of expected packets and invoke a barrier. The current API provides such a synchronization primitive that, as previously discussed, employs two reserved group counters to implement a fast, whole system barrier.

As previously explained, the surprise FIFO provides another way to send messages to a VIC without a pre-defined target address on its DV Memory; this alone may be sufficient for simple communications, or can also form the basis for coordinating DV Memory usage between cluster nodes. This ability to use the VIC's DV memory for both addresses and payloads also allows mechanisms whereby one VIC can send a packet to another VIC such that the destination VIC's DV Memory content triggers assembly and transmission of a new packet to another destination without any host intervention.

The API provides specific commands to exploit the VIC DMA engines; the current DMA mechanism requires using Linux Huge Memory Pages. Either 4 MB or 1GB HugeTLB pages can be allocated for VIC DMA use at runtime (the 1GB pages must also be reserved at boot time); these HugeTLB pages can be used as the host-side source or destination for VIC DV Memory DMA operations. The VIC provides a DMA Table with 8192 entries to store the transactions to be executed between host memory and DV memory; a single DMA transaction may span multiple table entries. A separate background DMA process pulls incoming FIFO packet data from the VIC into a circular buffer on the host as it becomes available so that host-side checking for incoming FIFO data occurs quickly. The VIC also pushes a list of zero-valued group counters to a specific host memory location during idle PCIe cycles (via reverse bus-master DMA) so end-of-transmission states can be checked without incurring the latency of an explicit PCIe read. DMA transfers to the VIC run up to 4 times faster than direct writes, and DMA transfers from the VIC happen up to 8 times faster than direct reads. Incoming and outgoing DMA transfers can be overlapped, and multi-buffered DMAs enable better overlap of communication with host computations when compared to direct reads and writes.

Direct translation of classic MPI primitives to the low level primitives of the Data Vortex architecture is not always easy or possible. Conversely, the Data Vortex API provides capabilities not available in MPI. Thus, a developer may need to rethink the way an algorithm operates, or how it is implemented, to exploit the unique features of the interconnect.

IV. System Setup

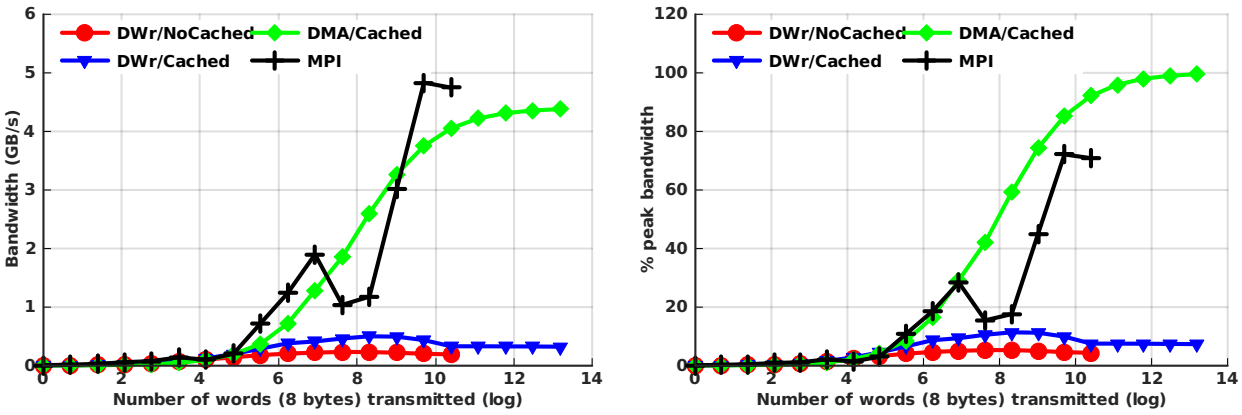
We tested a 32-node cluster equipped with two Intel E5-2623v3 Haswell-EP processors (4 cores with hyperthreading), 160 GB of main memory divided in two NUMA domains. The cluster is equipped with both DataVortex and Infiniband network adapters, as well as an Ethernet connection for system management purposes. The system distribution is Red Hat Linux with the Rocks 6.1.1 cluster environment, running the 2.6.32 64-bit SMP linux kernel. All applications are compiled with `gcc` version 4.9.2. Data Vortex applications use `dvapi` API library version 1.1, while MPI applications are linked against `openmpi` message passing library version 1.8.3.

Most of the MPI benchmarks belong to well-know benchmark suites. We developed those that were not publicly available. The Data Vortex applications have been developed by our team. This is an initial porting and not necessarily optimal. As we will see in the next sections, writing a Data Vortex application that performs well requires restructuring the algorithm to take advantage of Data Vortex hardware features. A simple replacement of MPI primitives with Data Vortex APIs does not generally yield satisfactory results.

V. Micro-benchmarks

We begin our evaluation of our Data Vortex system with a few low-level benchmarks that characterize system behavior. The goal of this section is to identify possible bottlenecks and drive the development of the kernels and applications presented in the next sections.

Ping-Pong Messaging This is a low-level exercise of fixed-length back-and-forth messaging to measure the network bandwidth visible to an application requiring round-trip communications. One node (sender) sends a fixed-length message to a second node (receiver). The second node sends a message from its memory back to the first node, while ensuring the entire received message gets copied from the network adapter into its local host memory.



(a) Absolute Ping-Pong bandwidth

(b) Percentage of peak bandwidth

Fig. 3: Network bandwidth as function of the message size.

We perform 1,000,000 ping-pong exchanges between two nodes with message lengths ranging from 1 word to 256k 64-bit words, directly writing message header and payload from main memory to the network (DVWr/NoCached), pre-caching message headers in part of the sending VICs DV Memory (DVWr/Cached), and using DMA to send from main memory with pre-cached headers and overlapping with DMA from the VIC DV Memory into host main memory (DMA/Cached). Note that, although the Data Vortex adapters always transfer fixed-size packets (64-bit header and 64-bit payload), data still need to be transferred over the PCI Express in large transactions to hide the I/O latency. Figure 3a shows that the Data Vortex network bandwidth when using direct write is generally low and limited by the PCIe lane read bandwidth (500 MB/s, only one lane is used). The bandwidth is a little higher when the destination headers are cached in the VIC's memory because the traffic over the PCIe bus is reduced. When using both DMA and caching headers in memory, the network bandwidth is much higher and close to the nominal peak bandwidth (4.4. GB/s) for large message transfers. We noticed that the Infiniband network bandwidth is higher than the Data Vortex bandwidth, especially for message sizes between 32 and 128 words, or larger than 512 words. We believe that there are two main reasons for these results. First, the Infiniband nominal peak bandwidth (6.8 GB/s) is about 50% higher than the Data Vortex peak bandwidth (4.4 GB/s). As shown in Figure 3b, the Data Votex implementation achieves 99.4% of the peak performance when transferring 256k words while the Infiniband network only achieves about 72% of the peak bandwidth. We believe that it is possible to achieve closer to peak bandwidth with Infiniband as well, but with much larger messages (that do not fit in the DV memory). Second, the MPI runtime performs many optimizations (e.g., double buffering) for well-known network transfer patterns such as Ping-Pong. The Data Vortex runtime does not (currently) perform such optimizations automatically. Even though VIC-to-VIC performance is good, other bottlenecks can impact application-level performance.

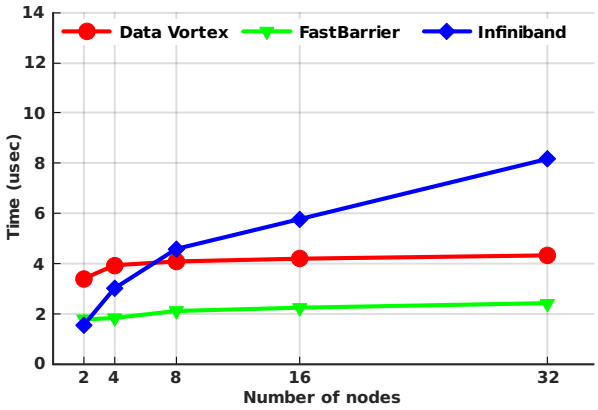
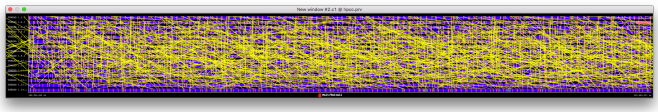


Fig. 4: Latency of global barrier at scale.

Global Barrier Global barriers are a challenge for bulk synchronous applications because of their limited scalability. Indeed, Figure 4 shows that MPI global barriers over Infiniband do not scale well, and that the time to complete a global barrier increases considerably with the number of compute nodes, especially when more than 8 nodes are involved.

The Data Vortex network provides hardware support for fast global and subset barriers: most of the communication (except for the initialization) is performed by the VICs without involving the host processor. We compare two implementations: the first ("Data Vortex") is an intrinsic within the current API and uses two group counters. The second implementation ("Fast Barrier") was developed in-house and relies on all-to-all communication. The results in Figure 4 show that, for Data Vortex, the time to complete the global barrier does not vary much with increasing numbers of compute nodes involved in the barrier. This indicates that barriers on Data Vortex network are very scalable operations.

(a) Complete execution trace



(b) Zoom

Fig. 5: GUPS execution trace.

Lesson learned Our experiments show that the Data Vortex network is not a simple replacement for MPI over Infiniband. The Infiniband driver and MPI runtime are very mature software, highly optimized over many years of research and development. In contrast, the Data Vortex software stack is new and lacks many optimizations. Nevertheless, this section's results show promise for applications that can be restructured to exploit the hardware features.

VI. Applications Kernels

In this section we move to more complex application kernels developed to stress particular computation or communication patterns. These kernels constitute basic blocks of larger applications, thus achieving good performance is predictive of the final application's performance.

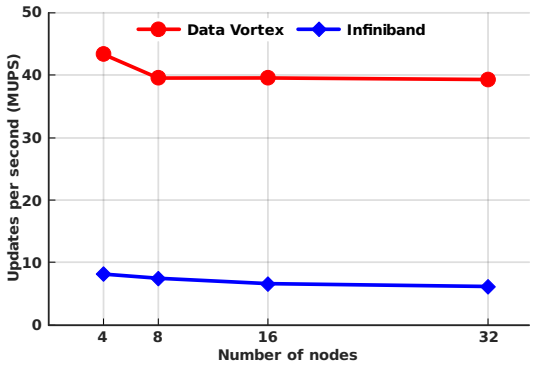
GUPS Current processors and networks are optimized for regular computation and memory/network traffic. As a result, even a small number of irregular or random accesses can severely limit overall application performance. The Giga Update Per Second (GUPS) benchmark has been designed to measure memory and network performance in the extreme case where all accesses are random. In GUPS, a distributed array of elements is randomly updated, and a system is characterized both by the size of the array that can be represented (total number of elements), as well as the sustained rate at which randomly selected elements can be updated. The initial value of each element is randomly generated, but does not affect the rate of updates. The update rate is measured as the average number of memory locations that can be updated in one second. In a single-node system, this benchmark measures memory performance. For parallel systems, this benchmark stresses the network because the elements are distributed over multiple machines and any individual node could update any particular element.

GUPS implementation rules are quite strict to avoid software implementations that attempt to artificially regularize irregular accesses. For example, the user is allowed to buffer at most 1,024 accesses. This constraint limits the amount of aggregation by destination often used to reduce network latency. Figure 5a shows a tracing of the HPCC GUPS implementation based on MPI [17], obtained using the Extrae [18] MPI instrumentation library. Figure 5b shows a close up in the central part of the applications. In both plots, blue represents computation, yellow lines depict messages between two nodes (remote accesses) and the other colors represent MPI functions. Figure 5b shows that there is no exploitable regularity for aggregating messages directed to the same destination.

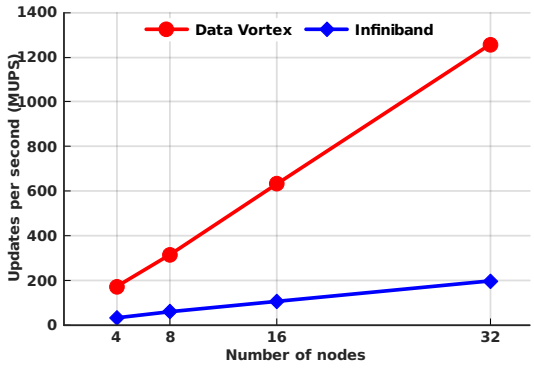
Our Data Vortex implementation takes advantage of the VIC's DV Memory to store the mapping between a range of global addresses and the compute nodes where they are physically stored. Another advantage comes from sending multiple small messages across the network; this plays to the strengths of the Data Vortex switch and does not congest the network. Finally, we implemented an "aggregation at source" scheme where multiple packets from the same compute node (source) that are directed to multiple destination nodes can be aggregated for transfer across the PCIe bus from host main memory to the VIC's DV Memory. Figure 6 presents the number of updates per second for our Data Vortex and the HPCC GUPS implementation. Figure 6a shows the number of updates per second per single processing element. Ideally, for weak scaling applications like GUPS, this value should remain constant with the increase in the number of compute nodes. Practically, however, network congestion and the higher probability of accessing a remote memory location often decreases this metric. This is evident for the HPCC GUPS implementation running over Infiniband, where the GUPS per processing element decrease constantly from 4 to 32 nodes. In contrast, the Data Vortex implementation shows much more scalable results, with constant performance, except for a reduction from 4 to 8 nodes. The net result is that the aggregated GUPS value (Figure 6b) is much higher for the Data Vortex implementation than for MPI-over-Infiniband. Moreover, the performance gap between the Data Vortex and the MPI implementation increases with increasing numbers of compute nodes.

FFT-1D The Fast Fourier Transform is a basic block of many scientific simulations. This kernel is very challenging because of the multiple matrix transpose operations ("butterflies") that need to be performed at each stage. Depending on the stage and how far data needs to be moved, these transpositions can introduce data access irregularities at each stage in the FFT. If a 2D or 3D FFT is performed, additional matrix transpositions may be required to optimize memory distributions. For the one-dimensional FFT benchmark, a discrete Fourier transform is performed on one-dimensional data distributed over multiple compute nodes. The size of the problem is defined by the number of discrete points in the FFT. In our experiments we use 2^{33} randomly initialized points.

We implemented a Data Vortex version of the FFT-1D benchmark and compare it with the HPCC MPI version [17]. In our Data Vortex implementation we take advantage of the natural scatter/gather capabilities of the network to perform the data transposition and redistribution operations. A partial row of points can be loaded in the VIC's memory and scattered to many destination nodes very efficiently. We also take



(a) GUPS per processing element.



(b) Aggregated GUPS

Fig. 6: GUPS

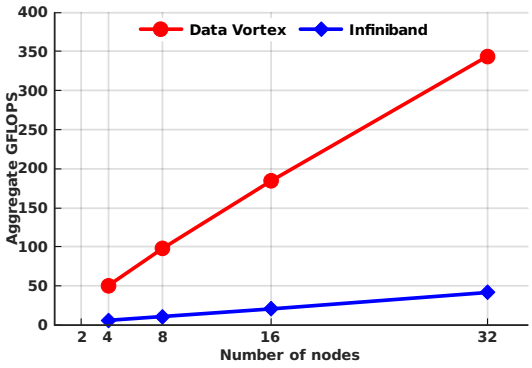


Fig. 7: FFT.

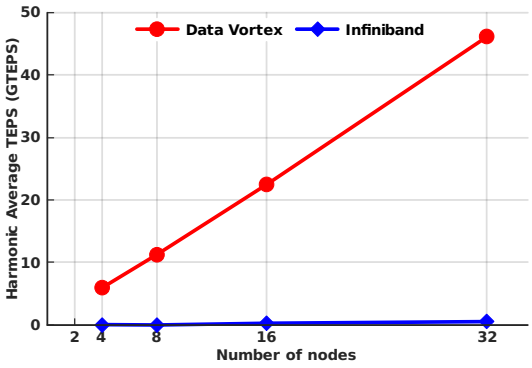


Fig. 8: Graph500.

advantage of the "aggregation at source" scheme to efficiently move data from host to VIC DV Memory over the PCIe bus.

We measure performance in terms of aggregated FLOPS. Figure 7 shows that the Data Vortex implementations not only performs better than the MPI-over-Infiniband counterpart, but also that, similar to GUPS in Figure 6b, the performance gap increases with the increasing numbers of nodes.

Breadth-first search The breadth-first search (BFS) is a very common kernel in emerging data analytics workloads. The benchmark allocates a very large random graph in a distributed memory system and performs 64 searches starting from random keys. We use the Graph500 [19] implementation as our baseline MPI-over-Infiniband reference. This benchmark is characterized by two key parameters: "scale" and "edge factor". The scale is the number of vertexes in the graph being searched, while the edge factor is the average number of edges per vertex (total graph edges divided by the number of graph vertexes). While the edge factor represents the average number of edges per vertex, the actual number of edges connecting to any individual vertex follows a power law distribution. Graph edges for the benchmark are generated with a Kronecker generator similar to the Recursive MATrix (R-MAT) scale-free graph generation algorithm. In this benchmark, we used the standard parameters for the Kronecker generator from the Graph500 documentation and tuned the scale factor to build the largest possible graph to store in the distributed memory. Performance can be measured in terms of traversed edges per second (TEPS), as well as the time to complete a search. The benchmark includes three phases: graph construction, search, and validation. The benchmark main metrics are extracted from the search phase.

Figure 8 shows the aggregate average TEPS for the Data Vortex and the MPI implementations versus increasing number of nodes. We observe that the performance of the Data Vortex implementation is consistently higher than the corresponding MPI implementation over Infiniband, and that the performance gap widens with increasing numbers of nodes. We believe that this kind of data-driven application is a strong fit for network architectures such as the Data Vortex. It is difficult to efficiently aggregate outgoing messages by their destinations (to assemble fewer, longer Infiniband messages). With the Data Vortex, we merely need a sufficient volume of outgoing messages from each node (that can be directed to different destinations) to ensure that host-to-VIC transfers across the PCIe bus happen efficiently. This "source aggregation", which is simpler to achieve than "destination aggregation", is sufficient to hide most PCIe latency.

Lessons learned The Data Vortex implementations of these kerenels can run faster than the reference MPI-over-Infiniband implementations for applications that cannot efficiently aggre-

gate outgoing messages by destination (e.g., GUPS). Furthermore, Data Vortex implementations that can send multiple messages in batches (even to different destinations) benefit from aggregating PCIe bus transfers and amortizing I/O latencies. Finally, our results highlight that applications which redistribute data (such as the FFT) can sometimes benefit by using the VIC's memory to fold redistribution operations (such as a transpose) into the application communications.

VII. Applications

We consider three prototype applications that have high communication cost per computation. All are iterative solvers for partial differential equations, but they have very different implementations, data transfer mechanisms, and resultant data costs. Our intent is to show how the main features of these problems can be addressed with simple methods that could be used by the readers as stepping stones to solve other, more complex problems.

Discrete Ordinates Application Proxy The SN (Discrete Ordinates) Application Proxy (SNAP) [20] is a proxy application for neutron transport applications. SNAP is designed to mimic the computational workload, memory requirements, and communication pattern of PARTISON[21], but most of the physics code has been removed to avoid distribution restrictions. SNAP is based on discrete ordinate methods. Six dimensions (three spatial, two angular, and one energy) are iteratively calculated over each time step. In the MPI reference implementation, the 3-D spatial mesh is distributed over a set of MPI processes. At each time step, the entire spatial mesh is swept along each direction of the angular domain, generating a large number of messages.

For the Data Vortex implementation, we performed a "best-effort" porting by first replacing the MPI primitives with equivalent Data Vortex API functions where possible, and re-implemented those other MPI primitives (where one-to-one substitution was not possible) using a combination of Data Vortex API calls. This simple porting, while quick, did not yield good performance. We then added an aggregation scheme (similar to those used in the prior sections) to minimize the number of PCIe transfers per message; this improved performance considerably.

Ideal Incompressible Flows Our second problem is modeling flow of an inviscid, incompressible fluid. The challenge lies in modeling the instability of the flow at small and decreasing spatial and temporal scales: as the flow evolves, it becomes more complex and finer grained, with a multiplication of eddies and vortex sheets. The best known manifestation of this phenomenon is the Kelvin Helmholtz instability, when adjacent regions of fluid with different tangential velocities develop complex patterns of vortexes. The equations describing this flow are derived from the Navier Stokes equations that describe a fluid flow [22] in the high Reynolds number regime $R = V\rho L/\mu \to \infty$, where L and V are the typical scales of length and velocity, respectively. In this case the

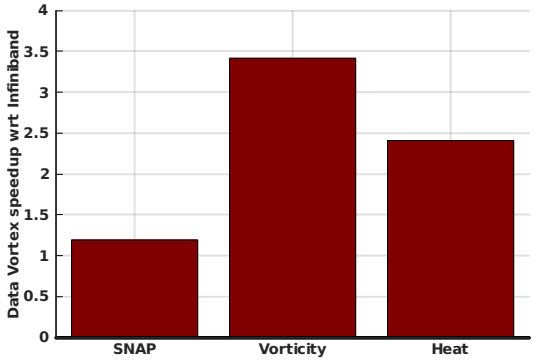


Fig. 9: Application speedup w.r.t. MPI-over-Infiniband.

Navier Stokes equation can be dimentionalized and reduced to Euler's equation:

$$\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} = -\nabla p. \quad (1)$$

For this algorithm, we significantly restructured the application to take better advantage of the Data Vortex network architecture. The majority of the communication cost is from computating five two-dimensional FFT's at each time step t. Assuming an entire row can fit into a single VIC's memory, the communication cost is equal to performing two matrix transpositions. The Data Vortex API enables delivering data to specific DV Memory locations within the VICs of specific nodes with very few API calls; this enables data reordering and redistribution to be integrated with "normal" data transfers without substatial additional overhead.

Heat Equation The heat equation is a parabolic partial differential equation describing the changing variations in temperature (heat flow) within a region over time. We solve the equation in three dimensions and employ domain decomposition over the processes involved in the applications. Thus, each process needs to communication with several neighbors, which results in a large number of small messages sent over the network. For the Data Vortex implementation, as in the previous case, we re-structured the algorithm to take full advantage of the underlaying hardware features.

Results We compare the Data Vortex implementation of SNAP, Vorticity, and Heat against MPI implementations of the same algorithms running over Infiniband. As we mentioned above, for SNAP we performed a "best-effort" porting with the addition of simple aggregation schemes for transferring data to and from the VIC's memory. For the Data Vortex Vorticity and Heat implementations, we performed a much more aggressive re-structuring of the algorithm to better align with the Data Vortex capabilities. The results are presented in Figure 9 in terms of relative speedup of execution time for the Data Vortex implementation over the MPI implementation. As we can see from the plot, SNAP "best-effort" porting provided a speedup of 1.19x over the reference MPI implementation;

we regard this as a very good result for the limited effort expended in porting the application. For the Vorticity and Heat applications we obtained speedups between 2.46x and 3.41x with the Data Vortex, but achieving this required a much more intensive re-structuring of the application to exploit the Data Vortex capabilities.

The experiments presented in this work indicate that there might be considerable benefits in using a Data Vortex network, especially for irregular and data intensive workloads. However, as exemplified by our experience with the SNAP application, attempting to use the Data Vortex system as a purely drop-in replacement for a traditional MPI-over-Infiniband implementation is unlikely to perform well.

VIII. Related Work

Many researchers have looked at ways to improve the performance, scalability, and ease of development of irregular applications on large scale systems. Those efforts span all levels of the stack, from architecture, to system software, to the algorithms and applications [23].

The Tera MTA, MTA-2, and their successors, the Cray XMT [2] and XMT 2, are well-known examples of systems custom-designed for irregular applications. The Cray XMT and XMT 2 employ the Cray Seastar2+ interconnect [24]; these network interfaces communicate through Hypertransport across a 3D-Torus topology. While now old, the Seastar2+ still provides a respectable peak message rate at around 90 Million. The XMT multithreaded processors (Threadstorms) implement a global address space across the whole system, and can generate up to 500M memory operations per second. A switch inside the processor forwards memory operations either to the node memory or to the network, applying only minimal packing before being sent through the network. However, the reference rate of the Seastar2+ decreases as the number of nodes increases, and the topology creates potential for contention on routers when there is a significant amount of non-local communication (i.e., messages directed to non-neighboring nodes in the topology). This is a typical situation of the unstructured traffic of irregular applications. The ThreadStorm handles these situations better than conventional cache-based processors by using extensive multithreading to hide the network latencies, but studies have shown scalability limitations [25]. The Urika-GD [26] data analytics appliance still employs an evolution of the Cray XMT design. Maintaining a fully-custom architecture is expensive, though, and may not be economically sustainable long term.

After Seastar2+, Cray developed Gemini and Aries network interconnects to use PCIe host interfaces. Gemini (still based on a 3D-Torus topology and used for the Cray XE6, XK6 and XK7 compute nodes) did not provide significant additional benefit for fine-grained unstructured traffic. Aries [27] (employed in the Cray XC series) was still based on the PCIe interface, but improved performance by using a Dragonfly topology [28] and including additional Fast Memory Access mechanisms for remote memory access.

IBM has done extensive development on their supercomputer interconnects. One of the major innovations for their PERCS (Productive Easy-to-use Reliable Computing System) was its network [29], with custom hubs integrated into the POWER7 compute nodes. The PERCS' hub module implements five types of high-bandwidth interconnects with multiple links fully-connected through an internal crossbar switch. The links provided over 9 Tbs/second bandwidth, and the internal switch supported a two-level direct-connect topology without external switching boxes. Additionally, the hub provides hardware support for collective communication operations (barrier, reductions, multicast) and a non-coherent system-wide global address space. Various analyses have shown the effectiveness of the PERCS network, provided that tasks are properly placed to maximize the links' utilization [30].

Bluegene/Q [31] systems implement a 5D torus interconnection among nodes; each BG/Q node provides 10 bidirectional links which operate at 2 GB/s in each direction. The network interface is directly integrated into the processor. The BG/Q network implements collectives and barriers over the same 5D torus interconnect. The BG/Q network has proven effective for collective all-to-all messages of about 8 KB. Several irregular applications, particularly graph kernels, have been optimized for the BG/Q. While they provide high performance [32], mapping the kernels to the architecture required significant development effort.

Infiniband is the most common supercomputer interconnect in current systems; new systems typically use Infiniband FDR (14.0625Gb/s per lane, 54.54Gb/s per 4-lane port). While Infiniband networks can be built in a variety of topologies (including dragonfly), fat-tree topology remains the most common. The reliance on fat-trees limits Infiniband effectiveness for unstructured traffic [33] and poses cost and scalability issues for large systems. Infiniband typically requires messages of several KBs length to reach peak bandwidth due to packet formation overheads. Even for the best FDR devices, Mellanox declares peak message rates of 100 Mref/s [34]. Infiniband also provides a low level API (verbs) for remote DMA operations, but this requires substantially higher coding efforts compared to MPI and has additional limitations. Irregular applications developed using MPI over Infiniband typically need to aggregate data and maximize bandwidth utilization with respect to the message rate.

IX. Limitations

We evaluated the performance of a real cluster using the Data Vortex network and compared it with the same cluster using MPI over FDR Infiniband. Comparisons of programmability, cost, power, and other factors are orthogonal and left as future work. Our present study is limited by the size of the system available and does not try to model performance for larger systems than currently available. To the best of our knowledge, no existing simulator can definitively predict the performance of an application running on a larger-scale Data Vortex system. Theoretically, network properties should be maintained when scaling up, as the same mechanisms and structures would be replicated. Each doubling of nodes would add an additional "cylinder" to the Data Vortex Switch, and

would double the number of layers for each cylinder. Those additional hops through the switch structure would (minimally) increase latency but should not change overall throughput per node. Developing and validating such a simulation is beyond the scope of this paper. Larger switch implementations would also depend on technical issues (e.g. pin counts, FPGA routing limitations, etc) that could impact practical scalability; not all can be accurately foreseen and modeled at this time.

X. Conclusions

Emerging applications for data analytics and knowledge discovery typically have irregular or unpredictable communication patterns that do not scale well on traditional supercomputers. New network architectures that focus on minimizing (short) message latencies, rather than maximizing (large) transfer bandwidths, are emerging as possible alternatives to better support those applications. In this work we explore a new network architecture designed for data-driven irregular applications, the Data Vortex network. This network architecture was designed to operate with fine-grained packets to eliminate buffering and to employ a distributed traffic-control to limit traffic logic.

We compared performance of several applications, from simple micro-benchmark to complex spectral formulation of inviscid, incompressible fluid flow. Our results are very promising and show that Data Vortex system can achieve considerable speedups over corresponding MPI implementations over FDR Infiniband. In particular, irregular applications for which it is difficult to aggregate messages directed to the same destination (such as GUPS and Graph500) show considerable performance improvements. Similarly, applications which redistribute data (such as our FFT) benefit from effective use of the VICs memory by combining redistribution operations within the communication operation.

References

[1] K. A. Yelick, "Programming models for irregular applications," *SIGPLAN Not.*, vol. 28, pp. 28–31, January 1993.

[2] J. Feo, D. Harper, S. Kahan, and P. Konecny, "ELDORADO," in *CF '05: Proceedings of the 2nd conference on Computing frontiers*, 2005.

[3] "Convey MX Series. Architectural Overview." http://www.conveycomputer.com.

[4] A. Morari, A. Tumeo, D. Chavarría-Miranda, O. Villa, and M. Valero, "Scaling irregular applications through data aggregation and software multithreading," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 1126–1135.

[5] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin, "Crunching large graphs with commodity processors," in *HotPar '11: the 3rd USENIX conference on Hot topic in parallelism*, ser. HotPar'11, 2011, pp. 10–10.

[6] G. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu, "Parallex: A study of a new parallel computation model," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007.

[7] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "Active pebbles: parallel programming for data-driven applications," in *ICS '11: the International Conference on Supercomputing*, 2011, pp. 235–244.

[8] O. Liboiron-Ladouceur, A. Shacham, B. A. Small, B. G. Lee, H. Wang, C. P. Lai, A. Biberman, and K. Bergman, "The data vortex optical packet switched interconnection network," *Journal of Lightwave Technology*, vol. 26, no. 13, pp. 1777–1789, July 2008.

[9] C. Hawkins, B. A. Small, D. S. Wills, and K. Bergman, "The data vortex, an all optical path multicomputer interconnection network," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 3, pp. 409–420, March 2007.

[10] C. Reed, "Multiple level minimum logic network, us patent 5 996 020."

[11] A. Shacham, B. A. Small, O. Liboiron-Ladouceur, and K. Bergman, "A fully implemented 12 times; 12 data vortex optical packet switching interconnection network," *Journal of Lightwave Technology*, vol. 23, no. 10, pp. 3066–3075, Oct 2005.

[12] R. G. Sangeetha, D. Chadha, and V. Chandra, "4x4 optical data vortex switch fabric: Fault tolerance and terminal reliability analysis," in *Fiber Optics and Photonics (PHOTONICS), 2012 International Conference on*, Dec 2012, pp. 1–3.

[13] R. G. Sangeetha, V. Chandra, and D. Chadha, "4x4 optical data vortex switch fabric: Component reliability analysis," in *Signal Processing and Communications (SPCOM), 2014 International Conference on*, July 2014, pp. 1–5.

[14] Q. Yang and K. Bergman, "Performances of the data vortex switch architecture under nonuniform and bursty traffic," *Journal of Lightwave Technology*, vol. 20, no. 8, pp. 1242–1247, Aug 2002.

[15] I. Iliadis, N. Chrysos, and C. Minkenberg, "Performance evaluation of the data vortex photonic switch," *IEEE Journal on Selected Areas in Communications*, vol. 25, no. 6, pp. 20–35, August 2007.

[16] "Data vortex technologies. the data vortex network system. available at: http://www.datavortex.com/architecture/."

[17] "The hpc challenge benchmark. available at: http://icl.cs.utk.edu/hpcc/."

[18] "Extrae instrumentation package," http://www.bsc.es.

[19] "The graph 500 list," http://www.graph500.org, November 2015.

[20] "SNAP," http://portal.nersc.gov/project/CAL/designforward.htm#SNAP.

[21] "PARTISN," http://portal.nersc.gov/project/CAL/designforward.htm#PARTISN.

[22] L. D. Landau and E. M. Lifshitz, "Fluid mechanics, 2nd edition, pergamon press."

[23] A. Tumeo and J. Feo, "Irregular applications: From architectures to algorithms [guest editors' introduction]," *IEEE Computer*, vol. 48, no. 8, pp. 14–16, 2015.

[24] D. Abts and D. Weisser, "Age-Based Packet Arbitration in Large-Radix k-ary n-cubes," in *SC '07: the 2007 ACM/IEEE Conference on Supercomputing*, 2007, pp. 5:1–5:11.

[25] O. Villa, A. Tumeo, S. Secchi, and J. B. Manzano, "Fast and accurate simulation of the cray xmt multithreaded supercomputer," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2266–2279, 2012.

[26] "Cray urika-gd graph discovery appliance. available at: http://www.cray.com/products/analytics/urika-gd," 2016.

[27] L. K. Bob Alverson, Edwin Froese and D. Roweth, ""cray xc series network. cray inc., white paper wp-aries01-1112"," 2012.

[28] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, June 2008, pp. 77–88.

[29] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony, "The percs high-performance interconnect," in *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, Aug 2010, pp. 75–82.

[30] D. J. Kerbyson and K. J. Barker, "Analyzing the performance bottlenecks of the power7-ih network," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, Sept 2011, pp. 244–252.

[31] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The ibm blue gene/q interconnection network and message unit," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 26:1–26:10. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063419

[32] F. Checconi and F. Petrini, "Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 425–434.

[33] T. Hoefler, T. Schneider, and A. Lumsdaine, "Multistage switches are not crossbars: Effects of static routing in high-performance networks," in *Cluster Computing, 2008 IEEE International Conference on*, Sept 2008, pp. 116–125.

[34] "Mellanox, inc. edr infiniband presentation. available at: https://www.openfabrics.org/images/ eventpresos/workshops2015/ugworkshop/friday/friday_01.pdf."