2018 International Conference on Field-Programmable Logic and Applications

Accelerating MPI Message Matching Through FPGA Offload

Qingqing Xiong* Anthony Skjellum† Martin C. Herbordt*
*Dept.of Electrical and Computer Engineering, Boston University
†Simcenter & Dept. of Computer Science and Engineering, University of Tennessee at Chattanooga

***Abstract*—The Message Passing Interface (MPI) is the *de facto* communication standard for distributed-memory High-Performance Computing (HPC) systems. Ultra-low latency communication in HPC is difficult to achieve because of MPI processing requirements, in particular matching requests and messages done by traversing the corresponding queues. Many researchers have addressed this issue by redesigning queues or by offloading them to hardware accelerators. However, state-of-art software approaches cannot free CPUs "from the misery" and hardware approaches either lack scalability or still leave substantial room for further improvement.**

With the emergence of numerous tightly coupled CPU-FPGA computing architectures, offload of MPI functionality to user-controlled hardware is now becoming viable; we find it productive to revisit hardware approaches. To maintain the generality necessary to support MPI while preventing high resource utilization, we design our MPI queue processing offload based on a recent analysis of performance characteristics in HPC applications. We propose a novel, two-level message queue design: a content addressable memory (CAM) coupled with a resource-saving hardware linked-list. We also propose an optimization that maintains high speed in the cases when the queue is long. To test our design, we create an SOC-based testbed consisting of softcore processors and hardware implementations of the MPI communication stacks. Even while using only a small fraction of the Stratix-V logic, our design can be one to two orders of magnitude faster than two well-known hardware designs.

***Index Terms*—MPI, message matching, message queue, FPGA communication processing, offload, middleware acceleration**

I. Introduction

Internode communication in non-trivial computer systems requires, from first principles, certain actions that are well understood and form the basis of communication protocols. A key problem of computer architectures is to consider how to accelerate the required semantics (*e.g.,* message matching). The Message Passing Interface (MPI) [1], as the dominant scalable parallel programming model, provides communication APIs with well-defined syntax and semantics, but does not mandate protocols (just API behaviors). Accelerating communication protocols amounts to improving the performance of MPI by choosing implementations that outperform software-only solutions. MPI's point-to-point message-matching requirements are quite general and so imply significant overhead, particularly on the receiver side.

MPI provides rich point-to-point communication primitives and collective communications that are process-group-scoped. These operations are widely used in HPC applications; they also take a large number of CPU instructions. Raffenetti et al. [2] analyze the MPI implementation MPICH and show that the average instruction count of 253 for an `MPI_Isend` call. For `MPI_Irecv`, the software path is similar. However, the number discussed is just the instruction count for posting a non-blocking send or receive request; it does not include the instructions for data send or receive processing. For instance, receive operations can consume a significant number of cycles for message-matching.

Message matching in MPI has drawn much of attention and has been called as "a misery" [3]. Why is it slow? First of all, MPI provides general matching criteria based on source, tag, wildcards, and the communicator. To address this requirement, MPI implementations such as MPICH and Open MPI use two major types of queues for receive operations. The posted receive queue (PRQ) is used for storing the unresolved posted receive requests, and the unexpected message queue (UMQ) is used for storing the arrived but not requested messages. These queues are usually implemented as linked lists. During the matching process, the linked lists are traversed for the match between a request and a message. Depending on the application and network behavior, the position of the matched element can vary, and each traversal can take $O(n)$ time, thus causing performance degradation.

Various approaches have been proposed to address the performance of message matching in the PRQ and UMQ. Software approaches seek to improve matching performance by breaking down the linked-list into smaller ones; this approach, however, provides limited speedup. Moreover, CPUs are still not free from the so-called misery. Another set of solutions is offloading message passing to an embedded processor on the Network Interface Controller (NIC); this reduces load on the host, but is still a CPU. Specialized hardware designs have also been proposed, but they either lack scalability or leave substantial room for performance improvement.

The thesis of this paper is this: As adding FPGAs to HPC clusters and their use for MPI is a growing trend [4]–[6], making full use of the reconfigurability and the degrees of design freedom of the FPGAs concomitantly becomes a cost-efficient choice. As part of the program to offload MPI functionality, we accelerate MPI message matching using pre-existing FPGAs in the cluster.

The contributions of this paper are: (1) We are the first to explore hardware MPI message queues on FPGAs, and the first to offload MPI message matching with both UMQ and PRQ

This work was supported in part by the NSF through awards CNS-1405695, CCF-1618303, and CCF-1821431; by the NIH through award 1R41GM128533; by a grant from Red Hat; and by Altera through donated FPGAs, tools, and IP.

on FPGA hardware. (2) Among the hardware MPI message matching designs, we are the only thus far to test the hardware designs by running the implementation on FPGAs instead of running simulations using CPUs. (3) We propose a novel two-level queue design for the UMQ and PRQ, which targets both speed and scalability. (4) We propose and implement an SOC design on Stratix-V FPGAs as the testbed. We test our design with two benchmarks and show performance improvement over two baselines [7], [8]. Our design outperforms both baselines by one to two orders of magnitude using about 10% of the on-chip logic resources.

II. Related Work

Improving MPI message matching performance has been much studied. Software approaches usually break down one linked list into multiple linked lists, each of which is entered through either hashing or indexes [3], [9]. To free up the CPU, message matching has been offloaded to the embedded processor on the NIC [10], [11]; but this approach does little to improve matching performance.

Offloading message matching using specialized hardware has been proposed using an associative list processing unit (ALPU) [7]; it provides a constant matching latency when the queue length fits in the ALPU, but consumes logic and does not have good scalability. They later proposed a microcoded processor that trades off performance for flexibility [12]. Mattheakis and Papaefstathiou [8] proposed a resource-saving ASIC design: the hash-based MPI processor. However, the performance gain is limited since it follows the same methods as used in software hash-based linked-lists.

Prior work on offloading MPI to FPGAs put little emphasis on message queues, while mostly concentrating on collective operations [13]–[15]. Saldaña et al. [16] use softcore processors to handle MPI primitives, including the matching process. Huang et al. [17] do not have PRQs in their design.

In distinction from the related work, we are, to the best of our knowledge, the first to explore hardware MPI message queues (UMQ and PRQ) on FPGAs, and the first to support nonblocking MPI operations with hardware stacks on FPGAs.

III. Design and Implementation

We consider message matching offload and use results from HPC queue analysis to evaluate queue design choices. We then describe a design based on two-level queues and discuss optimizations.

A. Message matching offloading

We design a hardware receive engine to perform message matching (Fig. 1). The PRQ stores the unresolved posted receive requests while the UMQ contains the headers of the unexpected messages and the data addresses (rather than the data); data is held in a separately. A completion unit (Comp Unit) keeps track of the number of completed requests.

When there is a receive request from the CPU side, e.g., an `MPI_Irecv`, the dataflow follows the red arrows in Fig. 1. The command finite state machine (FSM) polls for requests from the CPU and triggers the data matching FSM. The FSM in turn tries to find a matched message header from the UMQ. If none is found, the data finding FSM inserts the request into the PRQ. Otherwise it signals the Comp Unit and the data address contained in the matched entry is used to read the data memory. Finally, the output of the memory is moved to the CPU. If the request is a blocking request, such as an `MPI_Recv`, the PRQ and Comp Unit are bypassed and the receive call returns after finding a match.

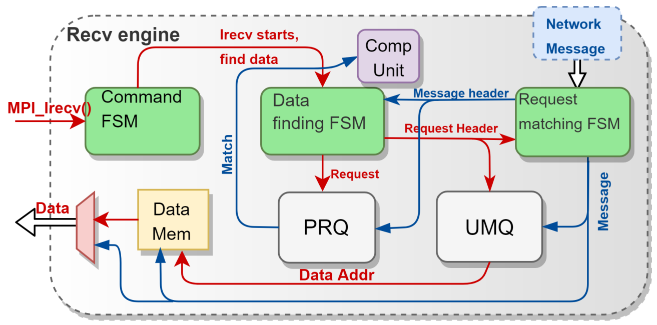

Fig. 1. Receive engine with message matching. Requests from the CPU flow along the red arrow; messages from the network flow along the blue arrow.

Now following the blue arrows, a newly arrived packet from the network triggers the request matching FSM to find a matched request from the PRQ. When a match is found, the request matching FSM removes the request from the PRQ and signals the Comp Unit. The received data is moved to the CPU. Otherwise the request matching FSM stores the data in data memory and performs an insert into the UMQ.

Non-blocking operations must test completion; blocking operations need not. When there is a *wait* call from the CPU testing completion, we pass the request count with it. The command FSM reads the counter in the Comp Unit, and waits until the counter threshold is reached.

B. Design choices of queues based on HPC queue analysis

Klenk and Fröning [18] have analyzed exascale proxy application traces, made available by the Department of Energy (DOE), to find characteristics of queue length and traversal depth. These results indicate that both the UMQ and the PRQ lengths are smaller than 128 elements in 50% of the various execution points measured, and smaller than 512 in 75%. Therefore, for both UMQ and PRQ, a small but fast storage unit is ideal for most applications. Meanwhile, they also show that the queue length can sometimes be much larger: a few thousand or even longer. Therefore, secondary storage unit is also needed that can be triggered in less common cases.

In MPI, wildcard matching may be required when a programmer specifies a receive request's source or tag as *Any*. We profile the NAS parallel benchmarks [19] and the Mantevo HPC proxy application suite [20] and find very few such instances. Only miniFE, HPCCG, and miniXyce in Mantevo have one or two receive requests with source rank wildcards and no tag wildcards are used. Furthermore, the wildcard receives can be potentially replaced by specifying the source rank. Therefore in this work, we do not provide wildcard

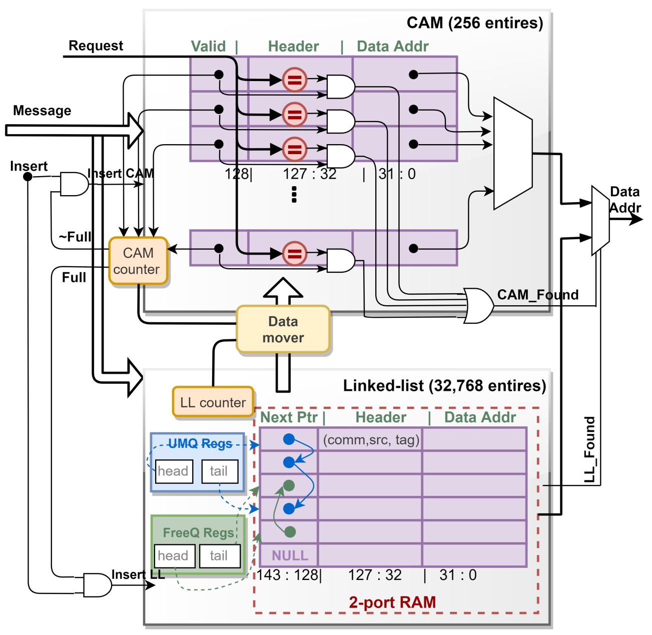

Fig. 2. Two level unexpected message queue(UMQ).

support. While such instances remain rare they can be handled elsewhere with little overall performance loss.

C. Two-level queue

We propose a two-level queue design based on the queue characteristics of HPC applications (Fig. 2). The first level is a content addressable memory (CAM) that can find a match in a single cycle; the second level is a resource-saving hardware linked-list. The CAM has a depth of 256 which is sufficient for most cases. Otherwise the linked-list with a capacity of 32,768 entries is used. As slots are freed up in the CAM, entries are replenished from the linked-list. Both levels have a counter unit to facilitate data movement. The counter units input the queue status to a data mover unit. Transfer takes one cycle for the linked-list, during which any insertion into the linked-list is stalled. Details are given in the next subsections.

1) First-level: Content Addressable Memory (CAM): CAMs are widely used in routers for network packet classification in the form of ternary CAM (TCAM). Here we eliminate wildcards and so implement CAMs instead of TCAMs. Xilinx and Altera both have CAM IPs designed based on RAM and/or shift registers. The throughput of these IPs is high but the insertion latency can be as high as 16 cycles.

We implement the CAM with one cycle of latency for both insertion and matching. As shown in Fig. 2, each entry has a valid bit, a 64-bit header, and a 32-bit address field. In the UMQ design, this 32-bit field is the address to the memory where the received data is temporarily stored. In the PRQ design, this field is the destination address in the CPU memory. The CAM counter unit keeps track of the valid fields and exposes the first empty entry for insertion. For matching, the CAM searches the request in parallel against all the header fields of the entries and produces the match as the result.

2) Second-level: hardware linked-list: The hardware linked-list (Fig. 2) is inspired by that in the hash-based MPI processor [8]. PRQ and UMQ are both mapped to an MPI buffer that is implemented with a single SRAM. It represents the empty entries as a free list. Each entry of the MPI buffer contains the source ID, tag bits, data address, and the next element's index. Each list has a register pair for storing head and tail. Search and insert requests of the two queues are processed sequentially via a list manager. A problem is that requests can arrive simultaneously from both processor and network. Moreover, queues need to process both traversals and insertions. Therefore when there are simultaneous read and write requests, to one or both queues, this serialization creates congestion and throttles performance.

Our linked-list is similar but we address the serialization problem by implementing it with two dual-ported BRAMs where each queue has two ports. With this modification, the design can handle simultaneously accesses and perform the operations in parallel. We also extend each entry with a communicator ID field.

When inserting an unexpected message into the linked-list of a UMQ, the linked-list first uses the head register in the free queue (freeQ) as an address to access the RAM, writes the new header value and the data address to the entry, and updates the freeQ head with the next pointer value returned by the RAM. Meanwhile, the next pointer field of the UMQ tail entry is updated with the inserted entry's index; the UMQ tail register is also updated. When searching, the linked-list reads the UMQ head register and starts traversing based on the next pointer value while comparing with the header field. When a match is found, a deletion in the UMQ and a push_back to the freeQ are performed.

D. Optimized two-level queue

The length and search depth of the queue can each be as high as a few thousand. Since search time increases linearly, this is too expensive; we therefore split the second-level linked-list into n smaller linked-lists. Insertion is done using a round-robin arbiter; searching is performed in parallel. The data mover moves an entry to the CAM from a single linked-list using a round-robin policy. By reducing the linked-list lengths to $1/n$ of the length and enabling parallel searching, the search latency in the linked-list for the last entry is also $1/n$ of the original case. In our experiments we configure n to be 4 or 64. We refer to these two designs as Parallel-two-level and Parallel-two-level-64.

IV. Experimental Setup

A. Testbed

To test the message matching hardware, we implement an SOC on a Altera Stratix-V 5SGSMD8 FPGA with hardware support for MPI point-to-point communication and using Altera tools for development (including Qsys for system integration). The SOC has three major components (Fig. 3). The CPU is a softcore 32-bit NIOS II IP [21], [22]; its peripherals are a JTAG UART, a phase lock loop (PLL), a performance

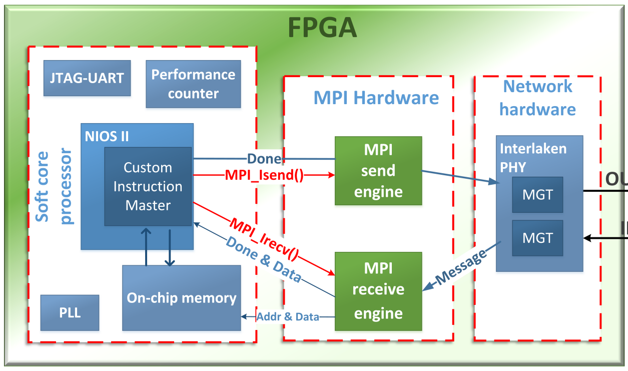

Fig. 3. Message matching testbed: MPI on SOC.

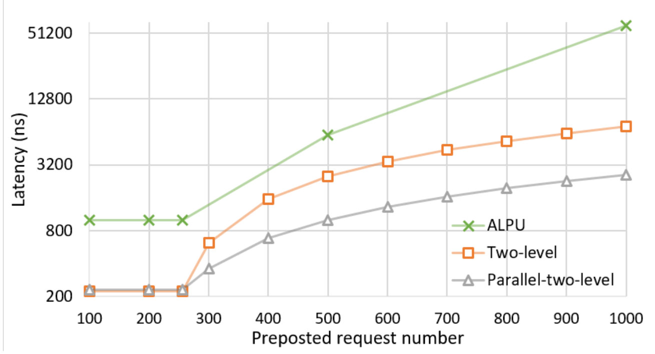

Fig. 4. Growth of latency of three posted request queue (PRQ) designs (logarithmic). The ALPU has 256 entries [7], the Two-level has one secondary linked-list, and the Parallel-two-level has 4 parallel linked-lists.

counter, and a 0.5MB on-chip memory. We wrap the MPI send and receive engines as two custom instruction slaves [23], and connect them to the custom instruction master port of the Nios II processor. The MPI engines use an eager protocol. We use the Altera Interlaken PHY IP to connect our MPI hardware with the Multi-gigabit Transceivers (MGTs). In our testbed four FPGAs are connected via the MGTs [4], [24].

After Qsys generates the system, we integrate the system into the overall Quartus II project. After compilation, the system is downloaded to the Stratix-V 5SGSMD8 FPGA. We port the benchmark code with the Nios II IDE. The NIOS II processor offloads the MPI operations in the benchmarks to the hardware design via custom instructions.

B. *Baselines and benchmarks*

We test our design with two benchmarks, and compare the performance results with two other similar hardware designs.

There is no hardware message matching design that combines state-of-art performance and scalability. For the cases where queue length is smaller than 1K, we compare our result with the 256-entry associative list processing unit (ALPU) result [7]. When the queue is larger, we compare with the 64-bucket, hash-based MPI processor [8]. It splits the single linked-list by hashing into 64 buckets based on source rank.

The baseline results are based on simulation. It is noteworthy that the ALPU simulates the network latency as 200ns, which is the same as our average network time via tightly connected MGTs. However, the hash-based MPI processor has not accounted for network time; therefore, for comparison we connect our MPI send and receive engines and use the loopback results.

The Preposted Latency Benchmark [25] is used by the ALPU; this benchmark is a standard for measuring the matching performance of MPI queues. The key idea is as follows: for testing the matching latency at depth N of the PRQ, it preposts $N-1$ eager non-blocking receive requests in the PRQ and then posts the latency measure request for matching. A *wait* call tests the completion of the receive.

Another important performance metric is time to unload when the queue is filled. We use an extension of the preposted latency benchmark introduced in the hash-based MPI processor [8]. After preposting N eager non-blocking receive requests, the benchmark dequeues the last element in the queue until the queue is empty. A *wait* call with a request count of N is used to finish.

V. Results and Analysis

A. *Performance results*

1) The preposted latency benchmark: With various queue lengths, we measure the average matching latency of the last entry in the queue. The results are shown in Fig. 4. From these results, we make the following observations.

- When the three designs are in the constant-time region, ALPU takes about 1 µs longer than our queue; this is because in the ALPU system, the offload happens twice: from the CPU, to NIC processor, and to the ALPU.
- When the traversal time dominates the latency, we outperform; e.g., when the preposed request count is 1000, our Parallel-two-level queue is 22x faster than ALPU.
- When the queue length is bigger than the CAM capacity (256), the traversal time of our designs increases linearly with the queue depth. This is because traversing the linked-list takes 3 cycles per element.

2) The unloading latency benchmark: We test the latency of emptying a queue; results are shown in Fig. 5.

When the queue length is below 256, then all of our designs outperform the hash-based MPI processor. When the queue length is 500 and 1,000, the hash-based MPI processor outperforms the two-level and the Parallel-two-level designs with hashing into 64 buckets. This hashing enables the possibility of reducing the traversal depth to 1/64, but there is bias when messages' source ranks are not evenly distributed. Our parallel linked-list resolves this bias, thus the speed benefits are shown when the queue lengths are 10,000 and 30,000. Furthermore, Parallel-two-level-64's performance is sustained at one order of magnitude over the hash-based MPI processor.

B. *Resource and frequency results*

We synthesize the designs using Quartus II. Resource utilization is shown in Table I. The maximum frequency of MPI hardware with the Two-level, Parallel-two-level, and Parallel-two-level-64 queues are 321.3 MHz, 234 MHz, and 168 MHz respectively. The frequency decreases as the parallelism of

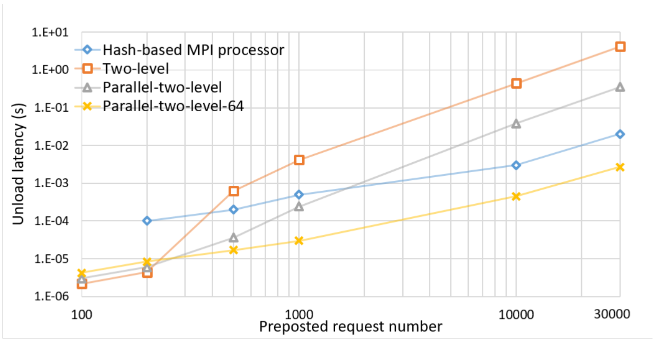

Fig. 5. The latency to unload queues with various posted request queue (PRQ) lengths. The hash-based MPI Processor has 64 hashing destination buckets [8]. The Parallel-two-level-bucket64 has 64 parallel linked-lists.

TABLE I
RESOURCE UTILIZATION OF DESIGN COMPONENTS

Components		ALMs	BRAMs	Registers
CPU		2151(<1%)	0.5MB	2596
Network hardware		6009(2.3%)	38.11KB	9404
MPI hardware	Two-level	21372(8.1%)	1.125MB	2816
	Parallel-two-level	21418(8.2%)	1.125MB	2862
	Parallel-two-level-64	21491(8.2%)	1.125MB	2934

the linked-list increases; this is because of the addition of an arbiter and a multiplexer.

C. Portability and scalability

For systems where the CPU and the FPGA are connected via PCIe, QPI, UPI, or Xilinx CCIX, our MPI hardware designs are portable. For instance, in a system where an X86 processor and an FPGA are connected by PCIe, our MPI hardware can be attached to the host CPU via PCIe IPs. The latency from user space to the FPGA hardware via PCIe-gen2 is 3.54 µs to 26.88 µs [26]; this latency is in addition to the matching latency reported.

The UMQ and the PRQ have a capacity of 33,024 entries each, which is sufficient to cover all the cases including on with 10,684 ranks [18]. Since FPGAs are configurable and the designs parameterized, the MPI hardware is scalable applications with up to one million processes.

VI. Conclusion

We address MPI message matching latency by offloading MPI to FPGA hardware. We propose and prototype a novel message matching design with two-level queues: a high-speed CAM and a scalable hardware linked-list. We implement an SOC on the Stratix-V FPGA as a testbed and evaluate our message matching designs with two standard benchmarks. As compared against two leading hardware designs, we provide one to two orders of magnitude speedup while consuming only a small part of the FPGA's resources.

Accelerating MPI middleware is essential for HPC systems. Future research includes enabling and demonstrating proxy application performance improvement based on message matching offload, and providing further support for offloading multiple MPI ranks per FPGA. We also plan to support the proposed persistent operations [27] for MPI-4.

References

[1] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," Knoxville, TN, USA, Tech. Rep., 1994.
[2] K. Rafenetti *et al.*, "Why Is MPI So Slow? Analyzing the Fundamental Limits in Implementing MPI-3.1," *Proc. Int. Conf. for High Performance Computing, Networking, Storage, and Analysis*, 2017.
[3] M. Flajslik, J. Dinan, and K. D. Underwood, "Mitigating MPI Message Matching Misery," *Int. Supercomputing Conf*, pp. 281–299, 2016.
[4] A. George, M. Herbordt, H. Lam, A. Lawande, J. Sheng, and C. Yang, "Novo-G#: A Community Resource for Exploring Large-Scale Reconfigurable Computing Through Direct and Programmable Interconnects," in *Proc. IEEE High Perf. Extreme Computing Conf.*, 2016.
[5] Q. Xiong, E. Ates, M. Herbordt, and A. Coskun, "Tangram: Colocating HPC Applications with Oversubscription," in *Proc. IEEE High Perf. Extreme Computing Conf.*, 2018.
[6] Q. Xiong, P. Bangalore, A. Skjellum, and M. Herbordt, "MPI Derived Datatypes: Performance and Portability Issues," in *Proceedings of the EuroMPI Conference*, 2018.
[7] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell, "A hardware acceleration unit for MPI queue processing," *Proc. 19th IEEE Int. Parallel & Distributed Processing Symp.*, 2005.
[8] P. M. Mattheakis and I. Papaefstathiou, "Significantly reducing MPI intercommunication latency and power overhead in both embedded and HPC systems," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, pp. 1–25, 2013.
[9] M. Bayatpour, H. Subramoni, S. Chakraborty, and D. K. Panda, "Adaptive and dynamic design for MPI tag matching," *IEEE International Conference on Cluster Computing, CLUSTER*, pp. 1–10, 2016.
[10] F. Petrini, W.-C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The Quadrics Network: High Performance Clustering Technology," *IEEE Micro*, vol. 22, no. 1, pp. 46–57, 2002.
[11] P. Shivam and P. Wyckoff, "EMP : Zero-copy OS-bypass NIC-driven Gigabit Ethernet," *SC*, November 2001.
[12] S. K. Hemmert, K. D. Underwood, and A. Rodrigues, "An architecture to perform NIC based MPI matching," *CLUSTER*, pp. 211–221, 2007.
[13] O. Arap, E. Kissel, and A. Shroyer, "Offloading Collective Operations to Programmable Logic," in *High Perf. Interconnects (HOTI)*, 2016.
[14] S. Gao, A. G. Schmidt, and R. Sass, "Hardware implementation of MPI barrier on an FPGA cluster," in *19th International Conference on Field Programmable Logic and Applications, FPL*, 2009, pp. 12–17.
[15] J. Stern, Q. Xiong, J. Sheng, A. Skjellum, and M. Herbordt, "Accelerating MPI_Reduce with FPGAs in the Network," in *Proc Workshop on Exascale MPI*, 2017.
[16] M. Saldaña, A. Patel, H. J. Liu, and P. Chow, "Using partial reconfiguration and message passing to enable FPGA-based generic computing platforms," *International Journal of Reconfigurable Computing*, 2012.
[17] L. Huang, Z. Wang, N. Xiao, Y. Wang, and Q. Dou, "Adaptive communication mechanism for accelerating MPI functions in NoC-based multicore processors," *TACO*, vol. 10, no. 3, pp. 1–25, 2013.
[18] B. Klenk and H. Fröning, "An Overview of MPI Characteristics of Exascale Proxy Applications," *ISC*, vol. 10266, pp. 217–236, 2017.
[19] T. Tabe and Q. Stout, "The use of the MPI communication library in the NAS parallel benchmarks," *Techincal Report*, pp. 1–16, 1999.
[20] M. A. Heroux *et al.*, "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.
[21] Altera, "Generic Nios II Booting Methods User Guide Hardware Abstraction Layer," 2016.
[22] Q. Xiong and M. Herbordt, "Bonded Force Computations on FPGAs," in *Proc. IEEE Symp. Field Prog. Custom Computing Machines*, 2017.
[23] Altera, "Nios II custom instruction user guide," *Altera*, vol. 95134, no. 408, p. 46, 2017.
[24] J. Sheng, C. Yang, and M. Herbordt, "High Performance Dynamic Communication on Reconfigurable Clusters," in *Proc. IEEE Conf. Field Prog. Logic and Applications*, 2018.
[25] K. D. Underwood and R. Brightwell, "The impact of MPI queue usage on message latency," *Proceedings of the International Conference on Parallel Processing*, pp. 152–160, 2004.
[26] J. Gong, T. Wang, J. Chen, H. Wu, F. Ye, S. Lu, and J. Cong, "An efficient and flexible host-FPGA PCIe communication library," *Proc. IEEE Conf. Field Prog. Logic and Applications*, 2014.
[27] A. Skjellum, "Persistent Point-to-point-channels for the 'point-to-point communication chapter'; ticket #88," 2018.