

# TMD-MPI: AN MPI IMPLEMENTATION FOR MULTIPLE PROCESSORS ACROSS MULTIPLE FPGAS

*Manuel Saldaña and Paul Chow*

Department of Electrical and Computer Engineering  
University of Toronto  
Toronto, ON, Canada M5S 3G4  
email: {msaldana,pc}@eecg.toronto.edu

## ABSTRACT

With current FPGAs, designers can now instantiate several embedded processors, memory units, and a wide variety of IP blocks to build a single-chip, high-performance multiprocessor embedded system. Furthermore, Multi-FPGA systems can be built to provide massive parallelism given an efficient programming model. In this paper, we present a lightweight subset implementation of the standard message-passing interface, MPI, that is suitable for embedded processors. It does not require an operating system and uses a small memory footprint. With our MPI implementation (TMD-MPI), we provide a programming model capable of using multiple-FPGAs that hides hardware complexities from the programmer, facilitates the development of parallel code and promotes code portability. To enable intra-FPGA and inter-FPGA communications, a simple Network-on-Chip is also developed using a low overhead network packet protocol. Together, TMD-MPI and the network provide a homogeneous view of a cluster of embedded processors to the programmer. Performance parameters such as link latency, link bandwidth, and synchronization cost are measured by executing a set of microbenchmarks.

## 1. INTRODUCTION

In the high-performance computing (HPC) community, programming standards have been developed to make it possible to port applications between different multiprocessor systems. Most notable are MPI [11] for message-passing systems and OpenMP [13] for shared memory systems. For a System-on-Chip (SoC), the call for a standard has also been made [20, 18, 10], including the proposal of using MPI [8, 24] in Multiprocessor System-on-Chip (MPSoC).

The development of large-scale, reconfigurable computing systems implemented on FPGAs will also require a programming model and borrowing from the HPC community

is an approach that will make it easier to bring HPC applications into the reconfigurable computing domain. For the TMD project [15], MPI has been chosen as the model because it works well for a distributed-memory system. A key difference is that in the HPC world, all computations are done using high-end microprocessors, whereas in an FPGA, computations could be done in software on an embedded processor or with a hardware engine. For a hardware engine, a hardware MPI protocol engine is required.

In this paper, the focus is only on the functionality of a software implementation of MPI targeted at embedded processors. This implementation sets a reference model for the future development of a hardware MPI engine.

We show a proof-of-concept platform that gives the programmer a unified view of the system, hides hardware complexities and provides portability to application codes. To test the functionality of the communication system we develop a set of microbenchmarks to measure the link latency, link bandwidth and synchronization cost.

The rest of the paper is organized as follows. Section 2 contrasts existing research with our approach. Section 3 describes the hardware testbed. Section 4 explains the MPI software implementation. Section 5 presents the current functionality of the library. Section 6 shows the TMD-MPI performance tests results. Finally in Section 7 we conclude.

## 2. RELATED WORK

Message passing has proven to be a successful paradigm for distributed memory machines, such as the grid infrastructure, supercomputers, clusters of workstations, and now it is promising for MPSoC. At the on-chip level, research has been done to provide message-passing software and hardware, but some of the proposals rely on the existence of an underlying operating system [21], which is an overhead for memory and performance to each soft-processor. Other approaches use a CORBA-like interface and DCOM object model [16] to implement message passing. In Poletti [3] another message passing implementation is presented, but it does not use a standard application program interface, so the application code is not portable. There

---

We acknowledge the CMC/SOCRN, Xilinx for hardware and tools, and CONACYT in Mexico provided funding to Manuel Saldaña. Thanks to Chris Madill for his help with the tests, and Amirix Systems for help with their boards.

are other message-passing implementations, but they are designed for real-time embedded systems [12] adding quality of service, but still relying on an operating system. Furthermore, some approaches are partial middleware implementations that assume other layers of software, such as a hardware access layer on top of which the message-passing library is built [9].

One way of providing MPI functionality to embedded systems is to port a well-known implementation of MPI, such as MPICH [5], but that requires resources that may not be available on an *on-chip* embedded system. A commercial MPI implementation for high-end embedded systems with large memories is MPI/PRO [19]. A similar approach to ours, can be found in Williams [22], but it is limited to only eight processors implemented on a single FPGA.

In this work, we develop a new implementation of MPI targeted at embedded systems tailored to a particular architecture, but easy to port to other systems. TMD-MPI does not require an operating system, has a small memory footprint (8.7KB) and is designed to be used across multiple FPGAs to enable massive parallelism.

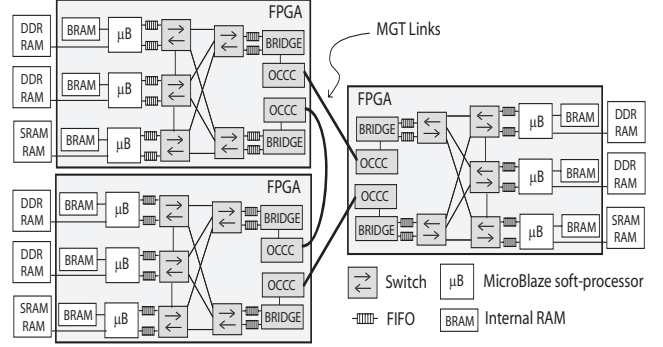
### 3. TESTBED SYSTEM

TMD-MPI has been developed as a result of the need for a programming model for the TMD machine being developed at the University of Toronto [15]. The TMD machine is a scalable Multi-FPGA configurable system designed to accelerate computing intensive applications, such as Molecular Dynamics.

Figure 1 shows the testbed hardware system used to develop TMD-MPI. We use three Amirix AP1000 development boards [1]. Each board has one Xilinx Virtex2-Pro FPGA (XC2VP100) [23], two 32MB DDR-SDRAMs and two 2MB SRAM. Each processor ( $\mu B$ ) is a Xilinx MicroBlaze soft-processor with an internal 64KB RAM memory (BRAM) connected to the Local Memory Bus (LMB) that stores the TMD-MPI and application code. The external RAM is used to store only data and is accessed by each MicroBlaze through an external memory controller that is attached to the On-chip Peripheral Bus (not shown in Figure 1).

Our multi-FPGA testbed system has two networks, one for intra-FPGA communications and one for inter-FPGA communications. We achieve this by developing a switch, a bridge, and an off-chip communications controller (OCCC) hardware block as shown in Figure 1.

The on-chip switch hardware block is a simple network interface that routes the ongoing packets according to the destination field (DEST in Figure 2). On the receiving side, the switch is a multiplexer controlled by channel priority logic. The switch is attached to the MicroBlaze using point-to-point communication channels implemented with Fast Simplex Links (FSL), which are essentially asynchronous FIFOs. Each MicroBlaze is attached to its own switch and the switches are fully interconnected as we want every node



**Fig. 1.** Testbed hardware system used to develop TMD-MPI

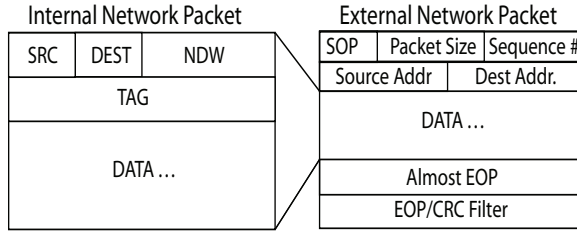
to communicate to all nodes on-chip.

The OCCC [2] is a hardware block that provides an interface between internal FSLs and the high speed serial links called the MultiGigabit Transceivers [23] (MGTs) in the Xilinx FPGAs. The OCCC provides a reliable communication link between FPGAs by checking CRC errors and handling packet retransmissions if necessary. The OCCC is seen as another network node and it has its own switch.

The internal network is more reliable than the external network because all data is interchanged within the chip, as opposed to the external network, which is exposed to signal interference and transmission errors. Therefore, the internal packet and external packet formats are different, and a bridge block is required to perform packet translation and enable the communication between processors across different FPGAs. The external packet format used by the OCCC includes the internal network packet as part of the payload. The internal network packet will be recovered as it passes through the receiving bridge. Both packet formats are shown in Figure 2. The external packet network format is not important to this work and is described in detail in Comis [2].

For the internal packet, the SRC and DEST fields are 8-bit values that identify the sender and the receiver respectively. In the current implementation only 256 processors are addressable, but this will be increased in future versions by expanding the width of the source and destination fields. NDW is a 16-bit value that indicates the message size in words (Number of Data Words); each word is four bytes wide. Consequently the maximum message size is  $2^{16} \times 4 = 256$  KBytes. The TAG field is the MPI message tag value that is used to identify a message.

The system has two clock domains, one clock at 125MHz for the OCCC, and a clock of 40 MHz for the rest of the system. Since the focus of this work is on functionality, we chose such frequencies to not complicate the place and route process. Additionally, each MicroBlaze requires an interrupt controller and a timer used for profiling purposes (not shown in the Figure 1). The resources required to implement each FPGA in Figure 1 are 26% of the total number of slices and 26% of the total BRAM available on the chip.



**Fig. 2.** Packet Formats

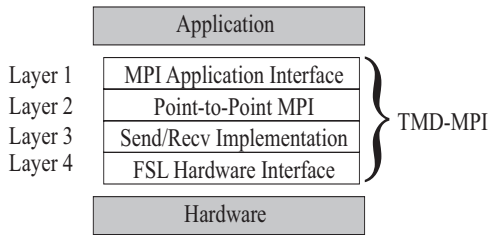
## 4. TMD-MPI IMPLEMENTATION

Currently, TMD-MPI is software-based, as it is easier to learn, develop, test and modify. Eventually, much of the functionality in TMD-MPI can be transferred to hardware to have a more efficient communication mechanism and decrease the overhead of the software implementation. The software-only version can be used where performance is not critical or where adding more hardware is not desired.

Elements that typically conform to an MPI implementation, such as communication protocols, pending message queues, packetizing and depacketizing large messages are design decisions discussed in this section.

### 4.1. A Layered approach

From the development perspective, an MPI layered implementation approach, such as used by MPICH, is convenient because it provides portability for the library. To port the MPI implementation to a new platform, or to adapt it to a change in the hardware architecture, it is required to change only the lower layers; the top layer and application code remain intact. Figure 3 shows the layered scheme implemented in TMD-MPI.



**Fig. 3.** Implementation Layers

Layer 1 refers to the API function prototypes and datatypes available to the application. This layer is contained in a C header file. In layer 2, collective operations such as MPI.Barrier, MPI.Gather and MPI.Bcast, are expressed in terms of simple point-to-point MPI function calls (MPI.Send and MPI.Recv). Layer 3 is the implementation of the MPI.Send and MPI.Recv functions that deals with the protocol processing, performs packetizing and depacketizing of large messages, and manages the unexpected messages. Layers 2 and 3 are both implemented in C code.

Layer 4 consists of four assembly-language macros that provide access to the MicroBlaze FSL interface. To port TMD-MPI to a PowerPC, a new Layer 4 would be required and some modifications to Layer 3 might be required, but Layers 1 and 2 would remain the same.

### 4.2. Rendezvous vs Eager Protocol

An important choice to make is between the Rendezvous and Eager message-passing protocols, which translates into the *synchronous* and *buffer* communication modes [11] in MPI, respectively. The eager protocol is asynchronous because it allows a send operation to complete without the corresponding receive operation being executed. It assumes enough memory at the receiver to store the entire expected or unexpected messages, which could be on the order of KBytes or even MBytes depending on the message's size, otherwise buffer overflows will occur. If substantial memory for buffering is allocated then it may lead to wasted memory in cases where the buffer is underutilized. In an embedded system, with limited resources this protocol may not scale well.

The rendezvous protocol is synchronous and the producer will first send a request to the receiver. This request is called the message envelope and it includes the details of the message to be transmitted. When the receiver is ready, it will reply with a clear-to-send message to the producer. Once the producer receives the clear-to-send message, the actual transfer of data will begin. This protocol incurs a higher message overhead than the eager protocol because of the synchronization process. However, it is less demanding of memory space and it is more difficult to incur buffer overflows because it only has to store message envelopes, which are eight bytes long, in the event of unexpected messages.

We decided to use the rendezvous protocol in TMD-MPI because of the smaller memory requirements. A mix of both protocols would be desirable because an eager protocol would be better for short messages reserving the rendezvous protocol for large messages. This is an improvement scheduled for future versions of TMD-MPI.

### 4.3. Message queues

In typical MPI implementations, there are two different queues: one for unexpected messages and one for ongoing or pending receives. In TMD-MPI, we only use one queue that stores the pending message requests at the receiver side. When a process wants to send a message to the receiver, it first sends a message envelope with the message information. At the receiver, the receive function will try to match the expected message with the envelope that has arrived. If there is a match, then the receiver replies to the sender with a clear-to-send message; otherwise the envelope would be from an unexpected message, and it would be stored in the pending messages queue for a possible future receive function call that does match the envelope. The receive function calls will always look into the pending message queue for a

message envelope that might match the receive parameters before starting to poll the FSL for the expected incoming envelope. The current search algorithm within the queue is linear for simplicity, but more efficient algorithms can be implemented to reduce the search time if needed.

#### 4.4. Packetizing and depacketizing

The OCCC supports a variable packet size between 8 and 2048 bytes. Each word has four bytes, therefore the packet size is 512 words maximum, including the control words. This restriction does not exist for the internal network because the MGTs are not used for on-chip communication. However, for simplicity of implementing the MPI\_Send and MPI\_Recv functions, the maximum packet size for the internal network was chosen to be the same as the packet size for the external network. A packetizing process divides large messages into packets no shorter than two words and no longer than 512 words. Similarly, the inverse process of joining the packets is called depacketizing and is performed by the MPI\_Recv function every time a message is received. Since there is only one path for every packet to travel through the network from one point to another, each packet is always received in order and there is no need to keep track of a packet number in the depacketizing process.

### 5. FUNCTIONS IMPLEMENTED

The MPI standard was originally developed for supercomputers with plenty of resources, and it was designed as a generic library making it unsuitable for embedded systems. TMD-MPI is a small subset of MPI. Only 11 functions are implemented and some of them have restrictions, but these functions are sufficient to execute a wide variety of applications. TMD-MPI functionality can be gradually increased as required for a particular application based on the functionality of lower layers.

Table 1 shows a list of the MPI functions implemented in TMD-MPI at this time.

**Table 1.** Functionality of TMD-MPI

MPI_Init	Initializes TMD-MPI environment
MPI_Finalize	Terminates TMD-MPI environment
MPI_Comm_rank	Get rank of calling process in a group
MPI_Comm_size	Get number of processes in a group
MPI_Wtime	Returns number of seconds elapsed since application initialization
MPI_Send	Sends a message to a destination process
MPI_Recv	Receives a message from a source process
MPI_Barrier	Synchronizes all the processes in the group
MPI_Bcast	Broadcasts message from root process to all other processes in the group
MPI_Reduce	Reduces values from all processes in the group to a single value in root process
MPI_Gather	Gathers values from a group of processes

A workstation-based MPI implementation would rely on the operating system to create, schedule, and assign an ID to the processes. This ID is used by the MPI implementation to assign a rank to processors. A rank is a unique and consecutive number within the MPI execution environment that identifies a process. In TMD-MPI, since we do not rely on an operating system, we assign a single process to each processor at compile time, and the rank is defined for each process by using the *-D* option, which is used to define constants for the C compiler. The constant defined is used by the MPI\_Comm\_rank function to return the process rank.

### 6. TESTING TMD-MPI

There exist benchmarks [4, 14, 7, 17] to measure the performance of an MPI implementation. However, they are not designed for embedded systems and they assume the existence of an operating system, which we do not use. ParkBench [14] has a Multiprocessor Low-level section that measures some basic communication properties. Unfortunately, this testbench is written in Fortran and there is no Fortran compiler for the MicroBlaze. Therefore, we have developed our own set of C-language benchmarks adapted for embedded processors called *TMD-MPIbench*. The same testbench code was executed on the testbed system shown in Figure 1 on a network of Pentium-III Linux workstations (P3-NOW) running at 1 GHz using a 100 Mbit/s Ethernet network, and on a 3GHz Pentium 4 Linux Cluster using a Gigabit Ethernet Network (P4-Cluster). This proves the portability that MPI provides to parallel C programs. The P3-NOW and P4-Cluster are using MPICH versions 1.2, and our multiple MicroBlaze system is using TMD-MPI. These tests are meant to demonstrate TMD-MPI functionality and to obtain an initial performance measurement of the current TMD-MPI implementation, and the network components. A more detailed performance analysis and benchmarking is scheduled for future work.

#### 6.1. Latency and Bandwidth

The objectives of the first test are to measure the link latency and link bandwidth under no-load conditions, i.e., no network traffic. We do this by sending round trip messages between two processors. The variables are the message size, the type of memory in which data is stored (internal BRAM or external DDR memory) and the scope of the communication (on-chip or off-chip). The results are shown in Figure 4.

By using internal BRAM, the tests are limited to short messages because in a single BRAM (64KB) there is also code and data. This limitation is not present when using DDR memory. However, for large messages, the systems with BRAM achieve 1.6x the measured bandwidth than those with DDR memory because of the overhead of accessing off-chip DDR memory.

For short messages, the multiple MicroBlaze system achieves higher link bandwidths than the P3-NOW and P4-

Cluster because our ad-hoc network has lower latency than the Ethernet network. It also has a lower overhead compared to the TCP/IP protocol. Note that, in this case, latency affects bandwidth because we are measuring round trip times, and for short messages this overhead is more evident. But as the message size increases, the frequency at which the system is running becomes the dominant factor in transferring the payload data. The P3-NOW and the P4-Cluster achieve 1.8x and 12.38x respectively, more bandwidth than the multiple MicroBlaze system with external DDR memory at 200KB message size, but the testbed is running only at 40MHz. For clarity in Figure 4, not all the results from the P4-Cluster are shown as they would compress the other plots.

Similarly, from Figure 4, we can see that on-chip communication is faster than off-chip communication because of the extra cycles required for the bridge to perform the network-packet format translation and the OCCC delay, which increases the latency and impacts short messages. For larger messages the bandwidth tends to be equal because the internal-link and external-link tests are both running at the same system frequency reaching the MicroBlaze's maximum throughput.

By using the internal BRAM and on-chip communications only, we achieved the highest link bandwidth, but still less than the Ethernet P3-NOW and the P4-Cluster. By doubling the frequency of the multiple MicroBlaze system we believe we could achieve higher link bandwidth than the P3-NOW. Moreover, even higher bandwidths would be achieved if faster hardware blocks are used as producers because the MicroBlaze throughput rate is less than the internal network throughput rate and the MGT throughput rate.

The zero-length message latency provides a measure of the overhead of the TMD-MPI library and the rendezvous synchronization overhead. Since there is no actual data transfer, this latency is practically independent of the type of memory. For on-chip, off-chip, P3-NOW and P4-Cluster communications, the latencies are 17 $\mu$ s, 22 $\mu$ s, 75 $\mu$ s and 92 $\mu$ s, respectively. These measurements are taken using the MPI\_Wtime function and subtracting its timing overhead, which is 96 $\mu$ s for the MicroBlaze, 3 $\mu$ s for the P3-NOW and 2 $\mu$ s for the P4-Cluster.

## 6.2. Measured Bandwidth With Contention

A different situation happens when there is congestion on the network. The test consists of half of the processors sending messages of varying size to the other half of the processors. Whereas in the P3-NOW and P4-Cluster the worst case link-bandwidth remained almost the same, in our network the worst case link-bandwidth dropped almost by half of the bandwidth previously reported under no-load conditions. We believe this is caused by the simple linear-priority channel selection logic in the switch block and the synchronization nature of the rendezvous protocol. That makes an unfair scheduling for short messages because a request-to-

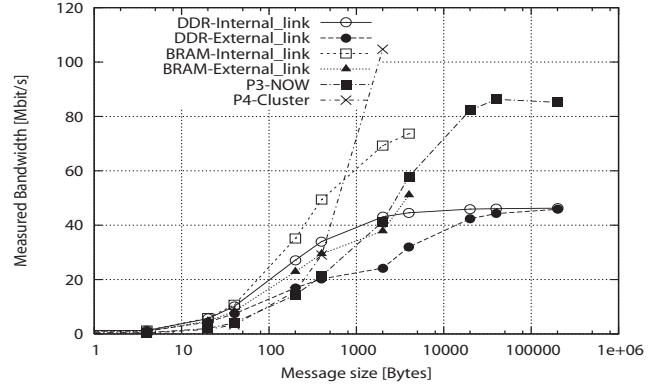


Fig. 4. Measured link bandwidth under no-traffic conditions

send message or a clear-to-send message from a channel lower in priority would have to wait for a long data message from a different channel with higher priority to finish. This would prevent the receiving MicroBlaze from overlapping communication and computation. The P3-NOW and P4-Cluster is using a buffered communication mode, which takes advantage of the absence of synchronization, and the Ethernet switch has a more advanced scheduling algorithm.

## 6.3. Synchronization performance

In parallel programs, barrier synchronization is a common operation and it is important to guarantee correctness. For example, the BlueGene Supercomputer [6] has an independent barrier network for synchronization purposes. To measure the synchronization overhead, 100 MPI\_Barrier function calls were executed; the variable is the number of processors in the system. The results are shown in Figure 5. It shows the number of barriers per second achieved as the number of processors is increased. Our testbed system and TMD-MPI provide low latency and low overhead, and since the synchronization is more dependent on latency than on frequency, our testbed system performs better than the P3-NOW, but not better than the P4-Cluster. As the number of processes is greater than the number of processors-per-FPGA, the off-chip communication channels are used and this means an increase in latency and more synchronization overhead. The barrier algorithm is another performance factor because as the number of nodes increases, a simple linear algorithm, such as the one used in TMD-MPI, becomes inefficient. A tree-like communication algorithm would be more scalable. Moreover, if the eager protocol is used instead of the rendezvous protocol, an increase of almost twice the number of barriers per second would be expected. Also we can see from Figure 5, the plot is smooth for the testbed, but not for the P3-NOW and the P4-Cluster. This happens because the testbed hardware is completely dedicated to run the testbench. For the P3-NOW and the P4-Cluster, the load of other users on the nodes may cause sudden changes.

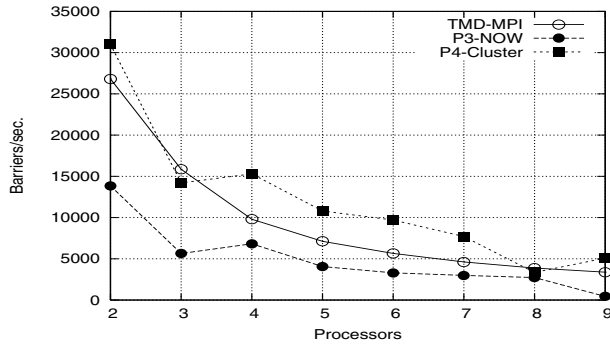


Fig. 5. Barrier Synchronization Overhead

## 7. CONCLUSIONS

This paper describes a lightweight subset MPI standard implementation called TMD-MPI to execute parallel C programs for a multiple processor System-on-Chip across multiple FPGAs. A simple NoC was developed to enable communications within and across FPGAs, on top of which TMD-MPI can send and receive messages. By executing the same C code in a Pentium Linux network of workstations and in our system, we showed the potential for MPI to provide code portability between multiprocessor computers and multiprocessor embedded systems.

Our experiments show that, for short messages, communications between multiple processors with an ad-hoc low-latency network and a simple packet protocol perform better than the network of Pentium 3 machines using a 100Mb/s Ethernet network and a Pentium 4 Cluster using a Gigabit Ethernet. For the current implementation, minimal latencies of  $22\mu\text{s}$  and maximum measured link bandwidths of 75Mbit/s are achieved with a clock frequency of only 40MHz.

TMD-MPI, does not depend on the existence of an operating system for the functions implemented. TMD-MPI is small enough that can work with internal RAM in an FPGA. Currently the library is 8.7 KB, which makes it suitable for embedded systems.

The tests show that the library works and provide some initial performance measurements. Future work will focus on tuning the library and developing a hardware block that performs the TMD-MPI tasks to enable more efficient communications and to be able to use specialized hardware engines instead of processors.

## 8. REFERENCES

- [1] Amirix Systems, Inc. <http://www.amirix.com/>.
- [2] C. Comis. A high-speed inter-process communication architecture for FPGA-based hardware acceleration of molecular dynamics. Master's thesis, University of Toronto, 2005.
- [3] P. Francesco, P. Antonio, and P. Marchal. Flexible hardware/software support for message passing on a distributed shared memory architecture. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 736–741, Washington, DC, USA, 2005. IEEE Computer Society.
- [4] W. Gropp and E. Lusk. Reproducible Measurements of MPI Performance Characteristics. In *Proceedings of Recent Advances in Parallel Virtual Machine and Message Passing Interface: 6th European PVM/MPI Users' Group Meeting*, Barcelona, Spain, 1999.
- [5] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, sep 1996.
- [6] IBM. BlueGene. <http://www.research.ibm.com/bluegene/>.
- [7] Intel, Inc. MPI Benchmark 2.3. <http://www.intel.com/>.
- [8] R. S. Janka and L. M. Wills. A novel codesign methodology for real-time embedded cots multiprocessor-based signal processing systems. In *CODES '00: Proceedings of the 8th international workshop on Hardware/software codesign*, pages 157–161, New York, NY, USA, 2000. ACM Press.
- [9] T. P. McMahon and A. Skjellum. eMPI/eMPICH: Embedding MPI. In *MPIDC '96: Proceedings of the Second MPI Developers Conference*, page 180, Washington, DC, USA, 1996. IEEE Computer Society.
- [10] Mobile Industry Processor Interface. <http://www.mipi.org>.
- [11] MPI. <http://www-unix.mcs.anl.gov/mapi/>.
- [12] MPI/RT Forum. <http://www.mpirt.org/>.
- [13] OpenMP Project. <http://www.openmp.org>.
- [14] ParkBench Project. <http://www.netlib.org/parkbench/>.
- [15] A. Patel, M. Saldaña, C. Comis, P. Chow, C. Madill, and R. Pomès. A Scalable FPGA-based Multiprocessor. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, California, USA, 2006.
- [16] P. G. Paulin et al. Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 48–53, New York, NY, USA, 2004. ACM Press.
- [17] SKaMPI Project. <http://liinwww.ira.uka.de/skamp/>.
- [18] Uniform Driver Interface. <http://www.projectudi.org>.
- [19] Verari Systems, Inc. <http://www.mpi-softtech.com/>.
- [20] VSI Alliance. <http://www.vsia.org/>.
- [21] J. A. Williams, N. W. Bergmann, and R. F. Hodson. A Linux-based Software Platform for the Reconfigurable Scalable Computing Project. In *MAPLD International Conference*, Washington, D.C., USA, 2005.
- [22] J. A. Williams, I. Syed, J. Wu, and N. W. Bergmann. A Reconfigurable Cluster-on-Chip Architecture with MPI Communication Layer. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, California, USA, 2006.
- [23] Xilinx, Inc. <http://www.xilinx.com>.
- [24] M.-W. Youssef, S. Yoo, A. Sasongko, Y. Paviot, and A. A. Jerraya. Debugging hw/sw interface for mp soc: video encoder system design case study. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 908–913, New York, NY, USA, 2004. ACM Press.