MPI as a Programming Model for High-Performance Reconfigurable Computers

MANUEL SALDAÑA, ARUN PATEL
Arches Computing Systems

CHRISTOPHER MADILL, DANIEL NUNES, DANYAO WANG, PAUL CHOW
University of Toronto

and

RALPH WITTIG, HENRY STYLES, ANDREW PUTNAM
Xilinx, San Jose

High Performance Reconfigurable Computers (HPRCs) consist of one or more standard microprocessors tightly-coupled with one or more reconfigurable FPGAs. HPRCs have been shown to provide good speedups and good cost/performance ratios, but not necessarily ease of use, leading to a slow acceptance of this technology. HPRCs introduce new design challenges, such as the lack of portability across platforms, incompatibilities with legacy code, users reluctant to change their code base, a prolonged learning curve, and the need for a system-level Hardware/Software co-design development flow. This paper presents the evolution and current work on TMD-MPI, which started as an MPI-based programming model for Multiprocessor Systems-on-Chip implemented in FPGAs, and has now evolved to include multiple X86 processors. TMD-MPI is shown to address current design challenges in HPRC usage, suggesting that the MPI standard has enough syntax and semantics to program these new types of parallel architectures. Also presented is the TMD-MPI Ecosystem, which consists of research projects and tools that are developed around TMD-MPI to further improve HPRC usability. Finally, we present preliminary communication performance measurements.

Categories and Subject Descriptors: C.1.3 [Computer Systems Organization]: Processor Architectures—Adaptable architectures; C.0 [Computer Systems Organization]: General—Hardware/software interfaces; D.1.3 [Software]: Programming Techniques—Parallel Programming

General Terms: Design, Experimentation, Performance, Standardization

Additional Key Words and Phrases: MPI, FPGA, Parallel Programming, Co-design, Programming Model, Reconfigurable, High-Performance

We acknowledge the CMC/SOCRN, NSERC and Xilinx for the hardware, tools and funding provided for this project. Thanks to Vincent Mirian for his collaboration and feedback.

Author’s address: Manuel Saldaña, Arun Patel, Arches Computing Systems, Toronto, Canada, email: {ms,ap}@archescomputing.com; Christopher Madill, Daniel Nunes, Danyao Wang, Paul Chow, University of Toronto, Toronto, Canada, email: {cmdill,dnunes,wangda,pc}@eecg.toronto.edu; Ralph Wittig, Henry Styles, Andrew Putnam, Xilinx, San Jose, USA, emails: {wittig,henry,s,andrewrp}@xilinx.com

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20YY ACM 0000-0000/20YY/0000-0001 $5.00
1. INTRODUCTION

Portability in a typical High-Performance Computer (HPC) is achieved by using standards-based operating systems and software layers (Middleware) that abstract machine-specific hardware from the software application, as shown in Figure 1 on the left. The UNIX operating system (and its variants) and the MPI [The MPI Forum 1993] library are examples of de-facto standards that enabled portable parallel applications across HPCs. For HPRCs designs to be truly portable, these concepts must be extended to the FPGAs, as they are now part of the application. Figure 1 shows how portability for HPRCs must come from two sides: application software and application hardware.

For HPRC’s, the hardware operating system and hardware middleware should provide a standard abstraction for application hardware engines to access host-specific resources, such as external memory and communications. Examples of these host-specific communication resources are the X86-FPGA communication mediums, such as Intel’s Front Side Bus (FSB) or QuickPath [Intel 2009a], AMD’s HyperTransport [HyperTransport Consortium 2009], Cray’s Rapid Array Transport [Cray Inc. 2005], SGI’s Scalable System Port–NUMA link connection [SGI 2008], or even PCI Express. Companies such as Cray [CRAY, Inc. 2009], SGI [SGI 2009], Intel [Intel, Corp. 2009], Nallatech [Nallatech, Inc. 2009], XtremeData [Xtreme Data Inc. 2009], DRC [DRC computer 2009] and SRC [SRC Computers, Inc. 2009] provide their own low-level software APIs and their own hardware interfaces for application hardware engines. HPRCs are now in a similar stage as HPCs were before the appearance of MPI [The MPI Forum 1993] when every vendor had their own message-passing API to program their own supercomputers causing a lack of portable designs. Currently, there is no implemented standard API for a high-level parallel programming model that includes the interaction between X86 processors and FPGAs. There is, however, interesting progress done by the OpenFPGA [2009] organization on this matter, which is compared against this work in Section 5.

Previous work proposed TMD-MPI [Saldaña 2006; Saldaña and Chow 2006; Saldaña et al. 2006; Patel et al. 2006] as a subset implementation of the MPI standard targeting Multiprocessor System-on-Chip implementations across multiple FPGAs to abstract hardware details from the user as well as to provide a well-known, high-level parallel programming API. In three years, TMD-MPI has

ACM Journal Name, Vol. V, No. N, Month 20YY.
evolved in a bottom-up approach to the point where it provides software and hardware middleware layers of abstraction for communications to enable the portable interaction between embedded processors, specialized hardware computing engines and now X86 processors; all programmed under the same message passing paradigm making it a unified programming model for HPRCs.

This paper provides a summary of TMD-MPI's philosophy and evolution, followed by a discussion of the new additions, current work and supporting tools and research projects called the “TMD-MPI ecosystem”. It is hoped that these research efforts generate discussion in the community regarding whether MPI is a good model for HPRCs or not, influencing perhaps existing MPI implementations to include FPGA support. It is time to consider how to include acceleration technologies, such as FPGAs, GPUs and CELL processors into a potential MPI-3 accelerator-aware standard because accelerator technologies also have data movement and synchronization requirements.

The rest of the paper is organized as follows. Section 2 explains the motivations that led to the development and use of TMD-MPI. In Section 3 an updated version of the design flow based on recent work is presented. Section 4 provides a real example of portability based on experiences using TMD-MPI. Section 5 contrasts TMD-MPI with related work regarding X86-FPGA communication. Section 6 describes TMD-MPI's architecture and HPRC reference platform. In Section 7, the current functionality of TMD-MPI is described. Section 8 presents a synopsis of the projects that constitute the TMD-MPI Ecosystem. Preliminary communication performance measurements are discussed in Section 9. Finally, Section 10 and Section 11 mention future work and some concluding remarks, respectively.

2. MOTIVATION

TMD-MPI was initially developed to address the need for a programming model for the TMD machine being developed at the University of Toronto [Patel et al. 2006]. The TMD machine is a scalable Multi-FPGA configurable system designed to accelerate computing intensive applications. The basic assumption is that FPGAs have enough resources to implement entire applications, as opposed to only parts of computations, such as small computing kernels. The objective is to have a tightly coupled mix of embedded processors (for control intensive parts of the application) and specialized hardware engines (to accelerate the compute-intensive parts of the application) all within the same FPGA or distributed among many FPGAs.

With multiple embedded processors and hardware engines (collectively referred to in this paper as computing elements, or CEs) interacting across multiple FPGAs, there is a need for a networking infrastructure and a distributed programming model. Most multi-FPGA systems have distributed memory (usually, each FPGA has its own external memory), and inside each FPGA there might be multiple CEs, each with their own local memory; hence, a programming model such as MPI seems adequate. With the appearance of tightly-coupled FPGAs and X86 processors sharing the main system memory (see [Cray, Inc. 2009; SGI 2008; Nallatech, Inc. 2009; SRC Computers, Inc. 2009; DRC computer 2009; Xtreme Data Inc. 2009]), TMD-MPI has been extended to include communication with X86 processors, marking TMD-MPI’s entrance to the HPRC world.
Figure 2 shows the generic model of a HPRC machine. It is possibly a multi-host machine, where each host can have one or more X86 processors and one or more FPGAs, each processor with one or more X86 cores, and each FPGA with one or more computing elements inside. A computing element can be an application-specific hardware engine or an embedded processor, such as a Xilinx MicroBlaze or a PowerPC. Note that in this model two or more FPGAs can indirectly share the same connection point to the host’s internal interconnection system (FPGAs 1, 2 and 3 share a connection). TMD-MPI proves that the MPI standard is a reasonable abstraction for addressing the communication and synchronization of such architectures.

There are two challenges when programming an application for this HPRC model. The first is the high-level, coarse-grain parallelization of the application, where decisions inherent to the parallel algorithm, such as data partitioning, synchronization and load balancing are made. The second challenge is the efficient use of the available FPGAs, where low-level decisions are taken, such as the number of pipeline stages, type of floating-point support and mapping a digital design to chip resources. A fully-automated flow for HPRCs that automatically parallelizes an application and at the same time converts it into hardware is a double challenge. This is especially true if a non-parallel-friendly, non-hardware-friendly language is used, such as C.

Perhaps at some point projects on high-productivity parallel programming languages, such as Chapel [Callahan et al. 2004], X10 [Ebcioğlu et al. 2005] and Fortress [Project Fortress Community 2009] can cooperate with tools that automatically generate HDL, such as ImpulseC [Impulse Accelerated Technologies, Inc. 2009], HandelC [Mentor Graphics 2009], Synfora [Synfora, Inc. 2009], MitrionC [Mitrionics, Inc. 2009], etc., to provide a fully-automated flow that overcomes the extraordinary challenges of programming HPRCs.
In the meantime, efficient applications for HPRCs have to be developed by a multidisciplinary team: a hardware engineer, a software engineer, and the application expert. The proposed approach is to deal with the two HPRC programming challenges independently. By parallelizing the code using MPI, the entire application is partitioned into smaller and simpler pieces of code, leaving the C-to-HDL tool (or hardware engineer) the easier task of turning into hardware more confined pieces of code. Writing high-level parallel code is still easier than writing low-level HDL code, so this approach is of great help to the non-hardware user. The experience of using TMD-MPI shows that MPI applied to FPGAs offers a convenient and clean interface between hardware (HDL) designers and software developers that allows independent progress of both due to the standard interface and well-defined semantics. Pragmatically, HPRCs exist now and the user community cannot wait for fully automated tools to magically appear and make them trivial to use. TMD-MPI and its surrounding ecosystem are examples of tools that exist today.

3. DESIGN FLOW

As previously discussed, the two major challenges to implementing an application on HPRCs are coarse-grain parallelization and HDL coding. To address these challenges, a top-down, step-by-step design flow was developed to implement applications using HPRCs. Figure 3 illustrates the updated version of the flow that was first introduced by Patel et al. [Patel et al. 2006]. This new version includes the X86-FPGA interaction.

Step 1 begins by developing a prototype of the application in a high-level programming language such as C/C++. The resulting code is sequential in nature and is only intended to provide a prototype solution to the computing problem. At this stage, the application can be profiled to identify computationally-intensive routines. The results of this prototype design can be compared to the results of the final system and validate the correctness of the implementation at any step during the flow.

Step 2 refines the prototype application by partitioning it into simple, well-defined processes that can be replicated to exploit the implicit parallelism of the application. At this point the user should have at least a hint about what processes are the most demanding, and therefore candidates for hardware engines. The user should avoid the use of any operating system calls or any feature that a hardware engine could not support. Inter-process communication is achieved using MPI, allowing the application to be developed and tested at the X86 level. No FPGA is involved yet. This approach has the advantage of allowing the programmer access to standard tools for developing, profiling and debugging. As an alternative to using TMD-MPI, other MPI implementations, such as MPICH [Gropp et al. 1996] or OpenMPI [Graham et al. 2005] can be used, however, the programmer must use only the available MPI functions implemented in TMD-MPI (see Section 7). Validation of the parallel algorithm can be done at this point, by comparing against the sequential version.

In Step 3, selected software processes developed in Step 2 are recompiled for embedded processors. The rest of the processes are still executed by the X86. The portability of MPI allows the software to be recompiled and executed on the em-
embedded processors with the code practically unchanged, there is no need to rewrite the application. This is an optional intermediate step where hardware engines are emulated by embedded processors in the FPGA. At this stage, execution of the entire application using the FPGAs is possible. This allows the interaction between X86 processors, embedded processors and emulated computing engines to be tested and validated. This step gives the user the opportunity to better exploit communication locality by putting together in the same FPGA control-intensive processes and compute-intensive processes that have an intense communication pattern.

The final step of the programming flow substitutes compute-intensive processes executing on embedded processors with hardware engines. Translating the compute-intensive processes into hardware engines is done manually in the current flow using typical FPGA design tools. However, since the system has already been partitioned into individual computing tasks and all communication primitives have been explicitly stated at this stage, C-to-HDL tools may also be used to perform this translation automatically, or as discussed in Section 8.3, MPI-to-HDL tools are feasible.

Once a hardware engine has been designed, an additional message-passing engine (TMD-MPE, Section 6.5) is used to perform message-passing functionality in hardware (e.g., protocol processing, handling message queues, divide large messages into packets), which simplifies the design of the hardware engine. Special versions of the TMD-MPE can also be used by the embedded processors as a communication coprocessor enabling a more efficient data exchange.

Fig. 3. Design Flow with TMD-MPI
4. AN EXAMPLE OF PORTABILITY AND SCALABILITY

This section gives an example of the portability that TMD-MPI provides to a reconfigurable computing application being developed with the rapidly evolving technology of FPGAs.

For the last three years, research has been conducted at the University of Toronto in collaboration with the Hospital for Sick Children to develop an FPGA-based machine to accelerate Molecular Dynamics (MD) simulations. These are atomic-level simulations of biomolecular systems used in biomedical research to study protein behaviour, such as when designing more effective drugs. The development of this machine began using the Amirix AP1000 PCI development boards [Amirix Systems, Inc. 2009]. Each board contains one XC2VP100 FPGA, 64 MB of external memory and five serial I/O links, which allowed five of these boards to be fully interconnected. The MD application coding was in progress when the new BEE2 platform [Chang et al. 2005] arrived with five XC2VP70 FPGAs per board, 4 GB of external memory per FPGA, using parallel LVDS lines and 18 high-speed serial links. This platform is obviously more integrated, provides more resources and potential for acceleration. Consequently, the development of the MD application continued on the BEE2 platform. The MD code was still in progress when the even newer Xilinx Accelerated Computing Platform (Xilinx ACP) arrived. This platform has Virtex 5 FPGAs that can be connected directly to the Intel FSB by inserting them into one of the CPU sockets in the motherboard. The FPGAs are tightly coupled to X86 processors sharing up to 256 GB of memory, opening up more opportunities for acceleration. Currently, the development continues for this platform.

The question is how to develop an application if the hardware keeps changing. Furthermore, how can existing code be isolated from future hardware changes. The impact of these hardware changes on the MD application has been minimized by using TMD-MPI. Indeed, there were changes, but all of them in the system infrastructure (what is described as SW/HW OS and SW/HW Middleware in Figure 1), not at the SW/HW application levels. The MD software code (written in C++) and the MD hardware engines (HDL) have been mostly the same, changing only in their natural evolution towards a more complete MD simulator.

One of the necessary changes was to regenerate some of the core netlists to take advantage of the newer Virtex 5 architecture. Another change was computing element reallocations. The bonded and non-bonded force calculations (particular types of forces in MD) were performed by two different computing elements that were located in the same FPGA, but when switching from the AP1000 boards to the BEE2 boards the bonded-forces compute engine had to be reallocated to another FPGA because the BEE2 board has smaller FPGAs than the AP1000 board. Moreover, the bonded-force computing element was originally a PowerPC processor, then it became a MicroBlaze soft-processor, but most likely it will end up being an X86 processor. Through all these changes, TMD-MPI provided tolerance to hardware changes and made it possible to perform architectural explorations and propose changes to achieve better performance without impacting the MD application code base.

Another advantage of using TMD-MPI was the abstraction of the FPGA con-
figuration process, which is usually not as easy to run as a typical program, and it is also a completely different procedure between the AP1000 boards, the BEE2 boards and the Xilinx ACP. Furthermore, in a HPRC, starting the execution of a large number of X86 software processes and configuring a large number of FPGAs must be automated. By using mpirun/mpiexec scripts, a difference like this can be minimized and hidden from the user perspective.

5. RELATED WORK

TMD-MPI was created with the influence of the Multiprocessor System-on-Chip [Jerraya and Wolf 2004] and Network-on-Chip research fields. Related work of those areas to TMD-MPI is discussed in previous work [Saldaña and Chow 2006; Saldaña et al. 2006] and briefly mentioned in Section 6.2. In this Section, other APIs for X86-FPGA communication are compared with TMD-MPI.

Currently, the most common usage of MPI on HPRCs is at the host level, where X86 processors exchange messages using MPI and then forward the received data to the FPGAs, which are mere slaves to the X86 processor (see [Aggarwal et al. 2004] as an example). This uses the X86 as a message relay introducing latencies and complicating the programming model because the user has to be aware of two different APIs: one for message-passing and another for interacting with the FPGA.

This section shows how MPI’s high-level API can be used instead of low-level API calls to perform X86-FPGA communications, with the added benefit of a broader communication context, more abstraction and a familiar API, providing a unified programming model. Note that many features that are available in the MPI standard are not implemented in TMD-MPI yet, and that certain features are not defined in the MPI standard but left open to the implementation. Here references to MPI mean the API itself, and references to TMD-MPI means our particular implementation of MPI.

At the time of writing this paper, OpenFPGA has released its first General API Specification 0.4 (GenAPI) draft [OpenFPGA 2009]. Its objective is to propose an industry standard API for high-level language access to reconfigurable computing resources in a portable manner. The draft is an excellent compilation of common X86-FPGA communication requirements for HPRCs. GenAPI shares practically all the same design goals as TMD-MPI, however, there is a fundamental difference: the scope of the standard. GenAPI only focuses on the low-level X86-FPGA interaction, not making any assumptions on higher level forms of communication, such as FPGA-initiated transfers, inter-host communications, inter-FPGA communications and intra-FPGA communications. In contrast, the MPI standard is agnostic of the communication media and requires that every rank (a task in the MPI jargon) be able to initiate transfers at any time regardless of its physical location. In this sense, the FPGA is no longer a passive slave in the system.

Similar to GenAPI, HPRC vendors provide their own APIs, but with the same limited scope: X86-FPGA communication, which is a subset of the communications that TMD-MPI is able to provide. Table I summarizes these communication capabilities. TMD-MPI will be able to provide communications at the host-to-host level, X86-to-X86 level, FPGA-to-FPGA level and intra-FPGA level (CE to CE); and in between these categories as well. That is, a hardware engine inside an FPGA in one
TMD-MPI's communication capabilities

<table>
<thead>
<tr>
<th>Master</th>
<th>Slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>X86 (core)</td>
<td>X86 (core)</td>
</tr>
<tr>
<td>X86 (core)</td>
<td>FPGA (CE)</td>
</tr>
<tr>
<td>FPGA (CE)</td>
<td>X86 (core)</td>
</tr>
<tr>
<td>FPGA (CE)</td>
<td>FPGA (CE)</td>
</tr>
<tr>
<td>Host (X86 core or CE)</td>
<td>Host (X86 core or CE)</td>
</tr>
</tbody>
</table>

host can exchange data with another hardware engine inside another FPGA in a remote host, without the need of an X86 to start or negotiate the communications.

In TMD-MPI, because of its on-chip origins and its distributed nature, achieving multi-level communications is straightforward. However, to make TMD-MPI portable across HPRC platforms, the low-level APIs are needed exclusively for X86-FPGA communication. For example, on the Xilinx ACP platform a custom low-level API is being used, but in principle other APIs could be used, such as Intel's AAL [Intel 2009b]. The GenAPI standardization efforts would help TMD-MPI by not having to implement a different version for each HPRC low-level API. By looking at some HPRC vendor APIs [Cray Inc. 2005; SGI 2008; Intel 2009b] and GenAPI, four general categories can be distinguished in the functions available: initialization and termination, FPGA management, data transfer, and memory allocation. The APIs are driver-like interfaces to the FPGA (open, close, read, mmap, etc.), but a system-level design of parallel applications that include FPGAs, needs a higher-level of abstraction.

Some of the low-level API calls can be easily mapped to MPI calls. For example, MPI function MPI_Init() is used to initialize internal data structures, to open the FPGA device and to signal the FPGA to start running. The opposite, MPI_Finalize() is used to signal the end of the application to the FPGA, remove the internal data structures and close the FPGA device. MPI_Send(), MPI_Isend(), MPI_Recv() and MPI_Irecv() are used to send/receive data to/from the FPGA in a blocking and non-blocking manner. For memory allocation, MPI provides the MPI_Alloc_mem() and MPI_Free_mem() functions to allocate/deallocate memory that the FPGA can access. Of course the MPI standard does not define how to download a bitstream to an FPGA, but this can be left open to the implementation. Currently in TMD-MPI, downloading a bitstream is part of the mpiexec or mpirun commands (but it will be part of TMD-MPI's process manager in future versions).

An MPI example using some of these functions is shown in Section 6.3.

Another interesting effort trying to bring portability to HPRCs is Vforce [Moore et al. 2007]. The authors provide a high-level object-oriented framework based on the VSIPL++ standard, which is a vector and image processing library API. Their approach is quite similar to ours in two fundamental points. First, one layer in their framework interacts with the low-level API calls rather than the final user’s code, insulating the application from hardware changes. TMD-MPI uses a low-level driver to access the FPGA, not the user’s code, which uses MPI calls. The second similarity is that they allow domain experts, such as FPGA designers, to create specialized hardware library components that can be interfaced with their framework. MPI allows the creation of third-party parallel libraries, and with
TMD-MPI it is possible to have software-hardware parallel libraries that interact based on an application-level message-passing protocol.

6. TMD-MPI ARCHITECTURE

In this section the Xilinx ACP is first presented as an example of the HPRC model described in Section 2. Then the mechanism that TMD-MPI uses to transfer data in the Xilinx ACP through a mix of shared and distributed memory is explained.

6.1 HPRC Implementation Platform

TMD-MPI has been used in a variety of platforms, but the focus here will be on the Xilinx ACP as it is the newest platform and the current target.

The base system is an instance of the HPRC model shown in Figure 2. It is based on a 4-Processor Server System S7000FC4UR motherboard. One of the processor sockets has an Intel Quad-core Xeon 7300 processor and the other two sockets have Xilinx ACP M2 Stacks [Nallatech, Inc. 2009]. Currently, the fourth socket is not being used, but more X86 processors or Xilinx M2 Stacks could be added. The Xilinx M2 Stack consists of a stack of up to three PCB layers (M2 modules). The first layer (bottom-up) contains the base module known as M2B, which contains a XC5VLX110 FPGA. This module is physically placed in one of the X86 sockets on the motherboard. Its function is to provide a connection point to Intel’s FSB and provide communications for the upper-level layers. The second and third layers are known as compute modules (M2C). Each M2C module contains two XC5VLX330 FPGAs in the same PCB. All the FPGAs are directly connected by LVDS lines in the same layer and between layers. The Xilinx M2 Stacks and the X86s can share up to 256 GB of RAM through the FSB, and the machine is running a standard 64-bit SMP Linux distribution. Figure 4 shows a system with three M2 Stacks and one quad-core processor. In this paper this machine is used as a reference implementation platform for TMD-MPI, but most of the concepts can be applied to other HPRCs as long as the FPGAs have access to main system memory.

6.2 Intra-FPGA and Inter-FPGA communication

A generic example for a multi-FPGA board is shown in Figure 5. The figure could physically represent five AP1000 boards, one BEE2 board or one stack of Xilinx M2 modules with one M2B module and two M2C modules. The group of FPGAs is called an FPGA_cluster. The communication is logically divided in Tiers based on its scope.

Tier 1 is the on-chip network that is used to exchange data between multiple computing elements within the same FPGA. This network is a set of point-to-point channels (a pair of FIFOs, one for transmission, one for reception), usually in a fully-connected topology, but other topologies can be used as well, depending on the communication needs. The network latency is usually a few clock cycles depending on the topology. The bandwidth is limited by the FPGA frequency and the width of the point-to-point channels. More about this is discussed in Section 9.

The Tier 2 network is used for inter-FPGA communication within the same FPGA_cluster. The gateways are connection points (bridges) between Tier 1 and Tier 2. Usually this network uses high-speed serial links or parallel LVDS lines. An important requirement is that this network has CRC and packet retransmission or
good signal integrity that guarantees error free transfers. One of the FPGAs, has a special gateway that connects Tier 2 to Tier 3 and Tier 4.

Tier 3, not shown in the picture, would be all the communications within the same host: FPGA\_cluster-X86, X86-X86, and FPGA\_cluster-FPGA\_cluster. Finally, Tier 4, also not shown in the figure would be the host-host communication network.

The Tier 1 network uses small distributed on-chip network interfaces (NetIfs), that forward packets to the right channels based on a routing table. The NetIfs use a buffer-less cut-through routing approach to reduce latency and to save on-chip resources. Some degree of buffering is implicit by using FIFOs as channels. Since all communication is FIFO-based, full and exists signals are used for flow control back-propagating the full condition to the sender.

Figure 6 shows an example of a connection between two computing elements: a MicroBlaze in FPGA 1 and a hardware engine in FPGA 2. Tier 1 and Tier 2 connections are established through a gateway node that acts as a bridge between the different tiers.
networks might have different protocols, in which case the Bridge block will handle network packet translations as they pass through the bridge. The bridge block can be as simple as just wires when packet formats between Tier 1 and Tier 2 are the same. However, in the case of FPGA_cluster-X86 communication the bridge is considerably more complicated, as will be seen Section 6.4.

Finally, the Off-chip Communication Controller (OCCC) is the block handling the physical access to the communication media (MGTs, LVDS, FSB, etc.). This block should deliver error-free data to the FPGA from the outside world and vice-versa. In the case of the AP1000 boards, this block is the FSL_Aurora core [Comis 2005], which uses the Xilinx MGT and Aurora link layer protocol for serial I/O. In the case of the BEE2 board, the blocks used are the FSL_interchip core for the LVDS interface (parallel link) and the FSL_XAUI core for the 10Gbit XAUI interface (serial link). In the case of the Xilinx ACP platform, the block is the Intel-Xilinx FSB protocol hardware core, which allows the FPGA to access the host’s shared memory.

6.3 Shared memory communication
TMD-MPI requires a distributed memory mind set from the user, but it takes advantage of the shared memory to perform the message passing for X86-X86, X86-FPGA and FPGA-FPGA communication in the same host. TMD-MPI provides two ways of performing shared-memory transfers. One is the typical approach that performs an extra copy from the user source buffer to an intermediate shared buffer, and lets the receiver copy the data from the intermediate buffer to the user destination buffer. The second approach available in TMD-MPI is a zero-copy technique, in which the sender or the receiver can copy the data directly from the user source buffer to the user destination buffer, without incurring the extra copy to the intermediate buffer.

TMD-MPI will automatically select one or the other based on how the buffer is created. If the source or destination (or both) buffers are allocated using the MPI_Alloc_mem() function then MPI will automatically select the zero-copy approach, otherwise TMD-MPI will use the intermediate buffers. This works for both X86-X86 and X86-to-FPGA communication (currently the FPGA-to-X86 zero-copy implementation is not yet operational). However, it does not mean that the user...
must allocate a buffer to be able to transfer it to the FPGA or to the X86. It just means that one way will be faster than the other depending on the situation.

This is convenient because at the programming model level the user does not have to allocate every buffer to be sent, even if it is a small payload. This is exemplified in Figure 7. The user can declare a variable or an array in a typical way and just send them (the extra copy approach will be selected by TMD-MPI), or the user can allocate two buffers and send them (zero-copy will be selected by TMD-MPI). A zero-copy transfer is convenient when transferring large amounts of data, usually critical to the performance of the application and justifying a memory allocation. But a simple intermediate copy would suffice for a non-critical small transfer, or when it is necessary to transfer large data sets at initialization stages, again not critical for performance; otherwise, extra memory allocations might be tedious for the programmer. In Figure 7, the variable dest can be the rank of an X86 processor, a MicroBlaze, a PowerPC or a hardware engine. The same API is used.

```c
main() {
  int x, ax[1000];
  int *px, *pax;
  ...
  MPI_Init();
  ...
  // ----- Send with implicit extra copy -----  
  x = 23;
  init_with_data(ax);
  MPI_Send(&x,1,MPI_INT,dest,tag,comm);
  MPI_Send(ax,1000,MPI_INT,dest,tag,comm);

  // ----- Send with zero-copy -----  
  MPI_Alloc_mem(1*sizeof(int),MPI_INFO_NULL,&px);
  MPI_Alloc_mem(1000*sizeof(int),MPI_INFO_NULL,&pax);
  *px = 23;
  init_with_data(pax);
  MPI_Send(px,1,MPI_INT,dest,tag,comm);
  MPI_Send(pax,1000,MPI_INT,dest,tag,comm);
  MPI_Free_mem(pax);
  MPI_Free_mem(px);
  ...
  MPI_Finalize();
}
```

Fig. 7. Transfers with implicit extra copy and zero-copy

6.4 The MPI_FSB_Bridge

The communication at the X86-X86 level is achieved using shared memory, and the computing elements inside the FPGA exchange messages using the Network-on-Chip. To exchange messages between X86s and computing elements, the data must travel through a shared memory MPI bridge (MPI_Bridge), which implements in hardware the same shared memory protocol that the X86 processors use, as shown in Figure 8. This bridge takes data to/from the NoC and issues read or write memory requests to the vendor-specific low-level Off-Chip Communication Controller (OCCC), which executes the request. The MPI_Bridge effectively abstracts the vendor-specific communication details from the rest of the on-chip network. In this way, the FPGA design can be ported to another HPRC as long as there is
a corresponding MPI_Bridge for the new architecture. This is true at least from
the communications perspective, because external memory interfaces might still be
different from one HPRC to another.

Fig. 8. The MPI Shared Memory Bridge is the interface between the shared memory and the
Network-on-Chip.

For this paper, the Intel FSB bus is used as the communication medium but the
same concepts can be applied to other communication media, such as AMD’s Hy-
perTransport [HyperTransport Consortium 2009], Intel’s QuickPath [Intel 2009a],
Cray’s Rapid Array Transport [Cray Inc. 2005], SGI’s Scalable System Port-NUMA
link connection [SGI 2008], or even with a standard PCI Express core because they
all provide a physical connection to the main system memory. The communi-
cation medium determines what OCCC to use, and in this case the Intel-Xilinx
FSB communication core handles the low-level protocol to read and write to mem-
ory as well as the memory coherence control. But most importantly, TMD-MPI’s
shared-memory, message-passing protocol should be the same across HPRCs or with
only minor variations. The only change is the physical interconnection between
the MPI_Bridge and the vendor-specific OCCC. By implementing a MPI_Bridge
for each type of OCCC the system is portable. For example, in this paper a
MPI_Xilinx_FSB_Bridge is used, but an MPI_Cray_Bridge could be implemented
to use a Cray HPRC machine.

An extension of this approach to a multi-host distributed-memory machine
is natural since message-passing assumes no shared memory. In this case, an
MPI_Ethernet_Bridge could be used, or any other point-to-point communication
interface, to allow the connection of multiple hosts through the FPGAs.

6.5 TMD-MPE

For processors, TMD-MPI is a software library that provides the message passing
capabilities, but for hardware engines to have message passing capabilities they
must use a block called TMD-MPE (TMD-Message Passing Engine), which encapsu-
lates in hardware some of the TMD-MPI functionality. The TMD-MPE provides
the hardware equivalent to MPI_Send and MPI_Recv to a computing element in
the FPGA, handles unexpected messages and the communication protocol, and di-
vides large messages into smaller size packets to be sent through the network. As
shown in Figure 9, the TMD-MPE is connected between the computing element
and the on-chip network interface (NetIf). The TMD-MPE receives the message
parameters and data from the computing element, such as the operation (whether it is sending or receiving a message), the destination node id (rank of the process in the MPI environment), the length of the message, and an identification number for the message (the tag parameter in a normal MPI send/receive operation). After this, the TMD-MPE will handle the communications through the network with the destination rank. The interfaces between the TMD-MPE and the computing element are four FIFOs (two for commands and two for data) making integration easy. These FIFOs can be asynchronous FIFOs allowing the computing elements to operate at different frequencies than the rest of the system.

The main application pipeline, or computing engine is wrapped inside a block that has a user-defined state machine whose purpose is to issue send and receive commands using the command FIFOs. The actual data should pass through the data FIFOs to and from the application pipeline. In this way, communications and computation are cleanly separated. The synchronization between the state machine and the application pipeline can be achieved through user-defined simple control signals, such as busy or done signals, or even register values that may act as input parameters to the application pipeline or return codes from the application pipeline that may change the communication pattern.

![Connection of the application hardware engine and the TMD-MPE.](image)

Fig. 9. Connection of the application hardware engine and the TMD-MPE.

7. CURRENT FUNCTIONALITY

TMD-MPI is in constant development because the MPI standard is very broad and there are still many features in the standard that are not implemented in TMD-MPI. Additionally, there are requirements in HPRCs that are not addressed in the standard, and some additions, such as the bitstream configuration and software resets to FPGAs, are functions that need to be placed somewhere during the execution of an MPI program. Our policy to expand TMD-MPI is on an as-needed basis. The current functions in TMD-MPI are listed in Table II.

Among other restrictions in TMD-MPI, currently collective operations can only perform MPI\_SUM and MPI\_MAX on scalar values. There is only one communicator (MPI\_COMM\_WORLD) and only the rendezvous message-passing protocol is implemented (no eager protocol for scalability reasons). However, some of these limitations can be relatively easy to overcome, and they are scheduled to be done.
### Table II. Functionality of TMD-MPI

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Init</td>
<td>Initializes TMD-MPI environment</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>Terminates TMD-MPI environment</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>Get rank of calling process in a group</td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>Get number of processes in a group</td>
</tr>
<tr>
<td>MPI_Wtime</td>
<td>Returns number of seconds elapsed since application initialization</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>Sends a message to a destination process</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>Receives a message from a source process</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>Non-blocking send</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>Non-blocking receive</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>Non-blocking send using synchronous protocol</td>
</tr>
<tr>
<td>MPI_Test</td>
<td>Tests if a non-blocking request is complete</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>Blocks the calling process until a non-blocking request is complete</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>Blocks the calling process until all the non-blocking requests in an array are complete</td>
</tr>
<tr>
<td>MPI_BARRIER</td>
<td>Synchronizes all the processes in the group</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>Broadcasts message from root process to all other processes in the group</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>Reduces values from all processes in the group to a single value in root process</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>Reduces values from all processes in the group, and broadcast the result to all the processes in the group</td>
</tr>
<tr>
<td>MPI_Gather</td>
<td>Gathers values from a group of processes</td>
</tr>
<tr>
<td>MPI_Alloc_mem</td>
<td>Allocates memory for Remote Memory Access (zero-copy transfers)</td>
</tr>
<tr>
<td>MPI_Free_mem</td>
<td>DEALLOCATES MEMORY THAT WAS RESERVED USING MPI_ALLOC_MEM</td>
</tr>
</tbody>
</table>

in the TMD-MPI project timeline. A future article will discuss TMD-MPI performance measurements as it requires more in depth analysis; however, Section 9 presents some preliminary performance results.

8. TMD-MPI ECOSYSTEM

There is ongoing research at the University of Toronto to create a supporting framework around TMD-MPI. The objective is to facilitate the application development for HPRCs by providing automation, debugging, verification and profiling tools. In this section, the goals and status for each of these projects is briefly summarized.

ACM Journal Name, Vol. V, No. N, Month 20YY.
8.1 Profiling TMD-MPI

In a parallel application, it is very important to understand how much time an application spends in communication and computation to identify potential improvements. Situations, such as communication bottlenecks and workload imbalance can be detected by profiling the application. The MPICH distribution includes the MPI Parallel Environment (MPE) [MPI Parallel Environment 2008], which provides the ability to profile different sections of the user application code as well as the MPICH code itself. This is done by code instrumentation using a predefined profiling API. The MPE includes a tool called JumpShot [Zaki et al. 1999] that visualizes the profiler logs, allowing a user to see what each rank is doing over a common timeline.

Inspired by this, a similar profiler for TMD-MPI was implemented, which works with hardware computing elements in FPGAs. The TMD-MPI profiler [Nunes et al. 2008] connects hardware blocks called tracers to the TMD-MPE, which provides all the information needed to profile the message-passing activity over the TMD-MPI network. A similar tracker is attached to a hardware engine or to an embedded processor to log computing events, such as pipeline busy signals that can help to determine the hardware engine utilization. The logs created by the TMD-MPI profiler are compatible with the logs created by the MPE allowing the same JumpShot GUI to be used; therefore, reducing the learning curve by not asking the user to learn a different tool, if already familiar with JumpShot. In the case of processors, the TMD-MPI profiler uses part of the same profiling API as the MPE.

Figure 10 is an example of the JumpShot GUI reporting data gathered by the TMD-MPI profiler. It shows a non-blocking data exchange between eight computing engines, implemented in an FPGA. The horizontal bars represent the time to receive a message (hash bar), compute (solid bar) and send a message (checker bar), and the arrows are control messages in the rendezvous protocol (request to send, clear to send and the payload data transfer). An overlap can be seen between communications and computation, which is not seen in the blocking version (not shown). For more details on this particular project see the work presented by Nunes et. al. [Nunes et al. 2008].

The TMD-MPI profiler has been implemented and tested only on the BEE2 platforms, and is able to use multiple boards. Future work for this project includes adding support for the X86-FPGA interaction, making the profiler suitable for profiling HPRC architectures.

8.2 Message Passing Simulation Framework

Compared to profiling, simulation is usually performed at the initial stages of the development to get a working FPGA-based system. A profiler is used to help improve the performance based on long-running, full-speed tests with real data sets, whereas a simulator is used to develop a functionally correct system, and run tests with size-reduced data sets at simulation speed. A simulation allows full visibility into the FPGA design and a fast code-debug-recompile cycle because there is no place-and-route process after a change in the hardware.

In the case of HPRCs, there are two components: the X86 software and the FPGA hardware. To design, debug and verify such architectures, co-design tech-
niques are taken from the embedded world but applied to the HPC world. These techniques are based on a system-level view of the application, and focuses on the interaction between hardware and software simultaneously, rather than designing two independent entities that will not coordinate efficiently. For this reason, the Message-passing Simulation Framework (MSF) [Saldaña et al. 2008] was created, which allows X86 software processes (R0 and R1 in Figure 11) to exchange MPI messages with the computing elements implemented in the FPGA (R2 and R3), which is being emulated by a simulator.

In a typical MPI parallel program, an MPI rank is not tested in isolation from the other MPI ranks. It has to be tested with all ranks running at once to verify the correct collective operation and synchronization between them. With FPGAs as containers of MPI ranks, they must be part of the system-level testing process. The MSF’s objective is to test and debug user-defined MPI-based application-level protocols as well as the correct functionality of hardware engines, all at once.

Figure 11 shows how the FPGA is being simulated using ModelSim [Mentor Graphics 2009], but from the X86 processor perspective it appears like a slow FPGA (simulation speed). The key element in the MSF is the use of Modelsim’s Foreign Language Interface (FLI), which allows a C program to have access to Modelsim’s simulation information, such as signals, register values and simulation control parameters. The MSF FLI module (C code) replaces the vendor-specific Off-chip Communication Controller (explained in Section 6.4) by providing the required functionality directly to the MPI Shared Memory Bridge. The MSF FLI accepts the MPI bridge memory requests (address and data) and performs the reads and writes directly to the shared memory buffers.

The MSF has been used to develop and debug the TMD-MPI infrastructure itself, but it can also be used in applications as well. A simulation of a LINPACK benchmark core example interacting with X86s and more details on this project can be found in the work presented by Saldaña et. al. [2008]. Additionally, the performance test application presented in Section 9, was developed using the design flow as described in Section 3 and simulated using the MSF before its actual implementation on the real FPGAs.
8.3 Not just C-to-HDL, but MPI-to-HDL

Currently, much research is being done on C-to-HDL tools because they will allow software developers, with no knowledge of hardware, to create FPGA-based hardware accelerators. However, this field is challenging because of the complexities of modern software and algorithms. A research project at the University of Toronto focuses on mixing MPI and a C-to-HDL tool (Synfora’s PICO Express FPGA [Synfora, Inc. 2009]). The resulting tool flow is called AUTO [Wang 2008]. The objective is to take an MPI program written in C as input and produce a hardware computing engine that can be interfaced with the TMD-MPI network.

The basic assumptions are that a parallel MPI code (at least for the rank that is a candidate to be turned into hardware) is simpler than the entire sequential application to be parallelized, and that explicit statement of the communication requirements (MPI_Send() and MPI_Recv() function calls) can be easily converted to stream-like reads and writes to FIFOs. Remember from Section 6.5 that FIFOs are used as interfaces to the TMD-MPE, which takes care of the message-passing protocol.

Figure 12 shows the AUTO flow. First the MPI code input is run through a preprocessing script that inserts codes that translate MPI function calls to stream operations in addition to hints and directives into a copy of the source code. The annotated code is then passed to PICO Express (Synfora’s compiler), which translates it into Verilog hardware files. These files are used by a packing script, which adds some control logic code and returns a complete pcore directory structure and files ready to be integrated into the Xilinx EDK tool.

A hardware engine was created from a simple MPI program that computes the Jacobi Iterations method to solve the Heat Equation using the AUTO flow. Results show that PICO Express works well for a subset of the C language; future additions to support pointers, more flexible looping structures, and floating-point will be welcome. An MPI program as a superset of a C program also suffers from the same lack of features. However, the extra work required to parse the MPI part over a plain C program is feasible. At this moment, the flow is designed to use Synfora, but a similar approach could be applied to ImpulseC, Handel-C, Mitrion-C, or any other C-to-HDL tool. See Wang [2008] for more details on this project.

9. PERFORMANCE MEASUREMENT

This section discusses some preliminary performance results using the HPRC described in Section 6.1. The focus of the experiments is on two basic communication
performance metrics: latency and bandwidth. A full and more extensive performance measurement and analysis that includes some computation is left for future work. However, these preliminary results are used to identify initial improvements that need to be addressed to increase the performance.

9.1 Experiments

To measure the latency and bandwidth, a simple 2-rank MPI application is created. These two ranks exchange round-trip messages of increasing size (steps). Every round-trip message (for a given size) is repeated ten times (samples) and the elapsed time is measured using the MPI function MPI_Wtime(), which returns the number of seconds since the application started to run. Then the total time accumulated is averaged over the number of samples. The MPI pseudo-code for the test is shown in Figure 13.

The two ranks synchronize each other by exchanging a couple of synchronization messages before the main loop test begins. This is done for every step to avoid having one of the two ranks start timing before the other, otherwise it may result in misleading bandwidth and latency measurements. As a validation resource, both ranks report the results and they must be approximately the same to confirm that the same measurement is observed from two different perspectives.

The two ranks in the test are implemented as a software rank (X86) and as a hardware rank (HW: hardware engine plus message-passing engine). The hardware engine is hand-written in VHDL and it implements a state machine that follows the same algorithm as in the C code for the X86. Having these two ranks gives the following combinations: X86-X86, X86-HW and HW-HW. In the cases where HW is involved, an X86 process will send initial configuration messages to the hardware ranks, such as number of samples, number of steps, and initial data values, which are compared to test the integrity of the data after the round-trip test completes.

In the case of the HW-HW test, there are two additional possibilities based on their physical location. Both HW ranks can be in the same FPGA or they can be in different FPGAs. In a system using the Xilinx-ACP platform these two FPGAs can be two base FPGAs (two M2B modules) or two compute FPGAs (one M2C module), or even one base FPGA and one compute FPGA (one M2B and one M2C module). Since the FPGAs are all connected through a high-speed LVDS core, we don’t expect to see any significant difference (4 to 6 additional cycles) compared to the option where both HW ranks are implemented in the same FPGA. More interesting is to measure two FPGAs communicating through FSB. For these reasons, the
MPI as a Programming Model for High-Performance Reconfigurable Computers

main() {
    ...
    MPI_Init();
    ...
    if ( my_rank == rankA ) {
        for(i=0; i<STEPS; i++) {
            // ----- Synchronize -----
            MPI_Recv(NULL, 0, MPI_INT, rankB, SYNC_TAG, MPI_COMM_WORLD, &status);
            MPI_Send(NULL, 0, MPI_INT, rankB, SYNC_TAG, MPI_COMM_WORLD);
            // ----- Main loop -----    
            t1 = MPI_Wtime();
            for(sample=0; sample < SAMPLES; ++sample) {
                MPI_Send(buf,msg_size[i],MPI_INT, rankB, sample, MPI_COMM_WORLD);
                MPI_Recv(buf,msg_size[i],MPI_INT, rankB, sample, MPI_COMM_WORLD, &status);
            }
            t2 = MPI_Wtime();
            delta_t = (double)((t2-t1)/(2.0*SAMPLES));
            BandWidth = sizeof(int)*msg_size[i]/delta_t;
            ...
        }
    } else if ( my_rank == rankB ) {
        for(i=0; i<STEPS; i++) {
            // ----- Synchronize -----    
            MPI_Send(NULL, 0, MPI_INT, rankA, SYNC_TAG, MPI_COMM_WORLD);
            MPI_Recv(NULL, 0, MPI_INT, rankA, SYNC_TAG, MPI_COMM_WORLD, &status);
            // ----- Main loop -----    
            t1 = MPI_Wtime();
            for(sample=0; sample < SAMPLES; ++sample) {
                MPI_Recv(buf,msg_size[i],MPI_INT, rankA, sample, MPI_COMM_WORLD);
                MPI_Send(buf,msg_size[i],MPI_INT, rankA, sample, MPI_COMM_WORLD);
            }
            t2 = MPI_Wtime();
            delta_t = (double)((t2-t1)/(2.0*SAMPLES));
            BandWidth = sizeof(int)*msg_size[i]/delta_t;
            ...
        }
    }
    ...
    MPI_Finalize();
}

Fig. 13. MPI pseudo-code for the round-trip message testbench

HW-HW test is performed for intra-M2B module communication (same FPGA) and inter-M2B module communication (different FPGAs on different modules). Figure 14 depicts the configuration tests for the experiment.

Note that from the HW rank’s perspective it is transparent where they are located; the HDL code of the HW engine does not change. It is the routing tables for the on-chip network that changes to let the network route the messages to the appropriate destination. The routing tables are stored in BRAM memory and their content in the bitstream can be updated with no need to rerun place and route.

The hardware engine is hand-written in VHDL and it implements a state machine that follows the same algorithm as in the C code.
9.2 Latency Results

We define message latency as the time it takes to send a zero-length message from one rank to another. It is an important metric because for small messages and synchronization operations, such as barriers, the latency is the predominant factor in the communication time. The latency will measure the overhead of calling the MPI functions (for software ranks) and calling the MPE (for hardware ranks) as well as the network latency.

In TMD-MPI, a zero-length message means there is synchronization (i.e., synchronization packets are exchanged) but no actual data transfer. TMD-MPI uses the rendezvous communication mode to transfer messages, which is a 3-way synchronous protocol. There is always a request-to-send packet and a clear-to-send packet before the actual data is transferred.

Figure 15(a) shows the latency plot (time vs message size) obtained for small-size messages between all the rank combinations. We can see that the lowest latency was obtained for the intra-M2B configuration because all communication takes place in the same FPGA. The lowest latency was 160 ns (approximately 21 clock cycles at 133MHz) to send the request-to-send packet and receive the clear-to-send packet. The second lowest latency is the X86-X86 latency with around 1 µs, which accounts for the TMD-MPI software stack to do the synchronization using shared memory; mostly pointer accesses and flag checking, but no memory copy occurs because it is a zero-length message. In third and forth place are the X86-M2B and inter-M2B latencies with around 1.8 µs and 2.1 µs, respectively. The rendezvous synchronization over the FSB bus incurs considerable overhead therefore increasing the latency.

In general, we can see that for message sizes greater than zero the time the round trip takes increases very slightly compared to the time of the previous message. This is because once the synchronization is done and the first word is sent through, the actual data transfer can happen very quickly at one word per clock cycle, or even take less time as in the case where an X86 is involved (X86-86 and X86-M2B tests) caused probably by cache effects, such as cache pre-fetching or already warmed-up caches. Flushing the X86 caches before every step would be an interesting experiment, but it is kept for future work.

By using the rendezvous protocol there is an increased latency, but it has its advantages. For example, there is no need to buffer entire unexpected messages, which otherwise would require more memory. It also avoids potential buffer overflows due to packet growth and synchronization delays.
to a lack of memory. This makes the rendezvous protocol more scalable memory-wise. There are other communication modes in MPI, such as Buffered mode and Ready mode that can be implemented in TMD-MPI to reduce the latency, but explaining them is beyond the scope of this paper.

9.3 Bandwidth Results

As opposed to a latency test where the messages are small, the bandwidth test requires large messages to measure the transfer rate; however, the latency also affects the message size by the total elapsed time divided by the number of samples. Note that this time also considers the synchronization time, which is more realistic to what one would experience in a normal application compared to just measuring the actual data transfer phase.

We identify three key factors for performance: synchronization, physical structure of the communication channel and packetization/depacketization overhead. The later will be discussed in Section 9.4. The physical structure of the channel determines its peak bandwidth. In the case of the X86-X86 it would be determined by how fast we can transfer data to/from memory. In the case of the on-chip communication, it would be determined by the product of the frequency times the width of the communication channel. In our case, the on-chip channels are 32-bits wide FSLs clocked at 133 MHz, which gives a peak on-chip bandwidth of 533.33 MB/s.

In Figure 15(b) we can see how the tests that involve the HW rank are limited to be under 533.33 MB/s, whereas the X86 achieves considerably higher bandwidths, at just under 1 GB/s. This probably happens because there are cache-to-cache transfers of data in the Xeon quad-core chip, avoiding access to main memory, which is not the case in the HW test configurations, which always have to access main memory. In addition, there is of course the frequency difference of 1.6 GHz of the X86 core compared to the 133.33 MHz at which the FPGA is clocked.

The effect of synchronization is reflected in the bandwidth plot as a measure of how fast the saturation bandwidth point is reached. With an ideal zero-time synchronization the bandwidth plot would be a horizontal line. From Figure 15(a), we know the intra-M2B latency is lower than the X86-X86 latency, therefore, the intra-M2B bandwidth is higher than the X86-X86 bandwidth for small messages as shown in Figure 15(b). After a certain size, the frequency and width of the channel becomes a more predominant factor causing the intra-M2B to reach the saturating bandwidth point earlier.

For inter-M2B and X86-M2B configurations the bandwidth is still limited by the on-chip bandwidth in addition to the extra latency of accessing main memory, which accounts for the actual latency of the memory requests passing through the North Bridge, to the actual memory chips and back to the FPGA, as well as the latency for the Intel-Xilinx FSB core plus the MPI_FSB_Bridge. Once the bandwidth is saturated the difference compared to the peak bandwidth is explained by the packetization/depacketization process, as explained in the next section.

9.4 Packet Size Impact on Bandwidth

Large messages in TMD-MPI are split into packets of smaller size to be able to use the packet-based network. However, the packetizing/depacketizing process in-
Fig. 15. Latency and Bandwidth Performance Measurements
introduces an overhead in performance decreasing the achievable bandwidth. This
effect can be seen in the bandwidth plots (Figures 15(d), 15(e) and 15(f)) as the
difference between the peak ideal bandwidth and the saturation bandwidth point.
Therefore, the packet size is an important factor in communication performance.
Large packet sizes will result in higher bandwidths, but also will incur more resource
usage (memory and logic resources in the FPGAs).

To analyze the impact of packet size in bandwidth, the same round trip message
experiment as in Section 9.3 is repeated but with different packet sizes: 128, 256,
512 and 1024 words per packet (each words is 32-bits wide).

Figures 15(c), 15(d), 15(e) and 15(f) show the bandwidth plot for the X86-X86,
X86-M2B, Intra-M2B and Inter-M2B tests, respectively. The x-axis uses a log
scale and it represents the message size. Notice that in all the cases the packet
size increases the bandwidth considerably, however, this increase is not directly
proportional. By doubling the packet size there is not a corresponding doubling in
bandwidth. For example, in Figure 15(f), with a packet size of 128 words/packet the
bandwidth saturates at 227MB/s, and with a packet size of 256 words/packet the
bandwidth saturates at 304 MB/s; this is only a 33.9% improvement. Furthermore,
as the packet size keeps increasing by a factor of two, from 128 to 256 to 512 to 1024
the observed improvement in bandwidth compared to the previous packet sizes are
33.9%, 19%, 10.6%, respectively; the benefits in additional bandwidth decreases as
it asymptotically approaches the peak bandwidth.

In the particular case of intra-M2B, the packetizing/depacketizing overhead is
minimal achieving 531.2 MB/s, which means 99.6% of the peak bandwidth. Keep
in mind that the peak bandwidth is 533.3 MB/s (32-bits wide channels @133
MHz). In contrast, the X86-M2B and inter-M2B achieve 410 MB/s (77% of peak
bandwidth) and 400 MB/s (75% of peak bandwidth), respectively. This happens
because in addition to the 533MB/s limitation, there is an additional overhead
to read and write to shared memory buffers to exchange data, which can only be
written or read at one time. A double buffering technique should alleviate this
problem, of course at the expense of more memory requirements.

Finally, we can see that in all the plots in Figure 15 the tests in which the X86 is
involved are noisier than those with only FPGAs. This is because FPGAs do not
use a cache, making their communication more predictable as there are no jumps
due to cache artifacts or operating system interference, such as time slicing and
process scheduling, even though the measurements were taken when nothing else
was running in the system.

10. FUTURE PERSPECTIVES

All together, TMD-MPI and its ecosystem are here and they provide a usable
environment. The main focus now will be on expanding TMD-MPI’s functionality
as we use it for more and more applications. Unfortunately, the Molecular Dynamics
application is not complete yet and performance results will be reported in a future
publication as it would require further discussion, which is beyond the scope of his
paper.

Naturally, performance improvements are scheduled targeting reduced latency
and increased bandwidth, while keeping in mind the application requirements to
maintain TMD-MPI as lightweight as possible. To do this, we will use more complete benchmarks, closer to existing MPI benchmarks, but adapted to use reconfigurable hardware. Existing benchmarks are designed to run on X86 processors with operating systems, which is not always the case with embedded processors and definitely not the case with hardware engines.

As shown in Section 9, the primary limitation of the bandwidth is due to the 32-bit, on-chip channels, however, results are promising given the current configuration. Future versions of TMD-MPI will provide 64, 128 and 256-bit wide channels, that should increase the bandwidth by two-fold, four-fold or eight-fold, respectively. In addition, increasing the frequency from 133 MHz to 266 MHz should also add another factor of two to the performance. The raw bandwidth available to the M2B module from the FSB bus is 8.5GB/s allowing plenty of opportunities to improve the performance by architectural enhancements and the use of more efficient protocols.

Additionally, we have the difficult task of convincing two different audiences, Software people and Hardware people, of the benefits of using TMD-MPI. Sometimes, hardware people do not understand how a typical software coarse-grain parallel programming API can help in hardware design, and software people sometimes do not understand how MPI can be used in a hardware environment. For this, future work includes writing documentation for both audiences and future papers with case examples of applications that use TMD-MPI.

11. CONCLUSION

TMD-MPI opens new research opportunities to implement an efficient MPI programming model for HPRCs, including hardware support for message-passing taking advantage of FPGA flexibility. In this paper, the ideas and tools that support MPI as a feasible programming model that abstracts the communications in a HPRC have been presented. The fact that MPI is based on a well-known and successful standard makes it even more appealing for parallel software developers.

TMD-MPI is not an exclusive idea; quite the opposite, it builds on top of other projects and vice-versa. It can benefit from OpenFPGA’s API standardization efforts, C-to-HDL tools while encouraging their improvements, new simulation techniques, new software/hardware profiling methods, and nothing prevents the use of a mixed shared memory plus MPI programming approach, if that helps to raise the programming abstraction. MPI provides a standard API and a common programming model for both software and hardware people while promoting scalability, portability and collaboration across HPRCs.

The performance measurements obtained in this paper, while still preliminary, are valuable metrics to drive continued improvement of TMD-MPI and assess the impact of such improvements. TMD-MPI has been used and developed as a proof-of-concept implementation. Although it is still work in progress, it is currently being used to code an MD application in an HPRC.

ACKNOWLEDGMENTS

We acknowledge the CMC/SOCRN, NSERC and Xilinx for the hardware, tools and funding provided for this project. Thanks to Vincent Mirian for his collaboration.
and feedback.

REFERENCES


Received Month Year; revised Month Year; accepted Month Year