MPI as a Programming Model for High-Performance Reconfigurable Computers

MANUEL SALDANA and ARUN PATEL
Arches Computing Systems

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

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

• Introduction
• Motivation
• Design Flow
• An Example of Portability and Scalability
• Architecture
• Current Functionality
• Performance
• Conclusion
Introduction

- Portability in a typical HPC depends on standardized Operating Systems and Software layers (Middleware) that abstract machine-specific hardware from the application.

- For HPRCs (High Performance Reconfigurable Clusters), there currently is no such abstraction for application hardware engines to access host-specific resources:
  - External memory
  - Host communication
    - Intel Front Side Bus (FSB)
    - AMD HyperTransport
Introduction (Cont.)

- TMD-MPI started as a subset MPI implementation targeting Multiprocessor System-On-Chip implementations across multiple FPGAs.  
  - This provided a well-known, high-level programming API for FPGAs

- TMD-MPI has now evolved to provide portable hardware and software communication middleware for embedded processors, specialized hardware computing engines, and x86 processors.

Fig. 1. Comparison between HPC and HPRC abstraction layers.
Overview

- Introduction
- **Motivation**
- Design Flow
- An Example of Portability and Scalability
- Architecture
- Current Functionality
- Performance
- Conclusion
Motivation

- TMD-MPI was developed to meet the requirements of the TMD machine at University of Toronto.
- The TMD machine is a multi-FPGA system designed to accelerate compute-intensive applications.
- Main Assumption: FPGAs have enough resources to implement entire applications versus small computing kernels.
  - Therefore, we can tightly couple implementations of embedded processors (for control) and specialized hardware engines (for compute) across one or many FPGAs.
Motivation (Cont.)

• In general, HPRCs have hosts with:
  • One or more x86 processors
  • One or more FPGAs
• FPGAs are either connected directly to the Interconnection System, or to another FPGA
• Within FPGAs are one or more Computing Elements (CE)
• **TMD-MPI guarantees portability as long as the host’s main memory is shared.**

![HPRC generic model](image)  
*Fig. 2. HPRC generic model.*
Overview

- Introduction
- Motivation
- **Design Flow**
- An Example of Portability and Scalability
- Architecture
- Current Functionality
- Performance
- Conclusion
Design Flow

• Two challenges:
  – High-level parallelization of the application for the HPRC
    • Data-partitioning, synchronization, and load-balancing
  – Low-level design choices for the FPGAs
    • Number of pipeline stages, type of floating-point support, and mapping a digital design to chip resources
• To meet these challenges, a step-by-step design flow was created to implement applications on HPRCs.
**Design Flow**

- **Step 1:**
  - Develop of prototype of the application in a high-level language such as C/C++
  - At this stage, the application is profiled to identify compute intensive routines
• Step 2:
  – Partition the program into simple, well-defined processes that can be replicated.
  – Use MPI for inter-process communication.
  – No FPGAs are involved yet.

Fig. 3. Design flow with TMD-MPI.
• Step 3:
  – Selected processes are recompiled for embedded processors
  – This is easy to do because of the portability of MPI
Design Flow

- **Step 4:**
  - Selected embedded processors are converted to specialized hardware engines.
  - This can be done either manually with HDL, or automatically with C-to-HDL tools.

Fig. 3. Design flow with TMD-MPI.
Overview

- Introduction
- Motivation
- Design Flow
- An Example of Portability and Scalability
- Architecture
- Current Functionality
- Performance
- Conclusion
An Example of Portability and Scalability

• University of Toronto has collaborated with Hospital for Sick Children to develop an FPGA-based machine to accelerate a Molecular Dynamics simulation.

• Development began with Amirix AP1000 PCI development boards
  – Soon into development, the more integrated BEE2 platform arrived
  – Then, the even newer Xilinx Accelerated Computing Platform arrived
An Example of Portability and Scalability

- In the MD application, two types of force calculations were done by CEs in the same FPGA.
- Switching from AP1000 to the BEE2 board, one type had to be moved to another to another FPGA because the BEE2 has smaller FPGAs.
- Moreover, the CE was a PowerPC processor, but then became a MicroBlaze soft-processor.
- TMD-MPI provided tolerance to these hardware changes without any changes to the MD application code.
Overview

- Introduction
- Motivation
- Design Flow
- An Example of Portability and Scalability
- Architecture
- Current Functionality
- Performance
- Conclusion
TMD-MPI Architecture: Reference Platform

- 4-Processor Server System
  S7000FC4UR motherboard
- One socket has an Intel Quad-core Xeon 7300 processor
- The other two sockets have Xilinx ACP M2 Stacks
- Fourth not used.
TMD-MPI Architecture: Tiered Communication

- Communication is divided into Tiers
- Tier 1 (Intra-FPGA):
  - On-chip network for xfer between CEs
  - FIFO queues for pt2pt communication
- Tier 2 (Inter-FPGA):
  - High-speed serial links or LVDS lines

Fig. 5. Inter-FPGA and Intra-FPGA communication.
TMD-MPI Architecture: Tiered Communication

![Diagram showing TMD-MPI Architecture](diag.png)

**Fig. 6.** Inter-FPGA and Intra-FPGA communication.
TMD-MPI Architecture: Shared Memory Comm.

- TMD-MPI provides two ways to do shared-memory transfers:
  1. Sender copies from his buffer to an intermediate buffer
     Receiver copies from the intermediate buffer to the dest. buffer
  2. Zero-copy directly from the source to dest. buffer
- TMD-MPI will select the protocol based on how the buffers were created
  - If MPI_Alloc_mem was used, Zero-Copy is selected
main() { 
    int x, ax[1000];
    int *px, *pax;
    ...
    MPI_Init();
    ...
    // ----- Send with implicit extra copy -----
    x = 23;
    init_with_data(ax);
    MPI_Send (ax,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(px);
    MPI_Send (px,1,MPI_INT,dest,tag,comm);
    MPI_Send (px,1000,MPI_INT,dest,tag,comm);
    MPI_Free_mem(px);
    MPI_Free_mem(pax);
    ...
    MPI_Finalize();
}

Fig. 7. Transfers with implicit extra copy and zero-copy.
TMD-MPI Architecture: MPI_FSB_Bridge

- x86-x86 communication is achieved via shared memory.
- FPGA—x86 communication must be done through an MPI_Bridge which implements in hardware the same protocol x86 uses for shared memory.
- The bridge takes Network on Chip requests and converts them to low-level, vendor-specific requests for the OCCC.
- The MPI_Bridge allows portability of the FPGA design to new HPRC architectures.

Fig. 8. The MPI Shared Memory Bridge is the interface between the shared memory and the Network-on-Chip.
On x86, TMD-MPI provides message passing capabilities.

Hardware engines must use the MPE which provides the hardware equivalent of MPI_Send and MPI_Recv.

Additionally the MPE:
- Handles the communication protocol
- Handles unexpected messages
- Divides large messages into smaller size packets to be sent through the network

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

- Introduction
- Motivation
- Design Flow
- An Example of Portability and Scalability
- Architecture
- Current Functionality
- Performance
- Conclusion
Current Functionality

- Functionality is added on an as-needed basis
- Currently, reduction collectives can only perform the MPI_SUM and MPI_MAX operations

<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_Aloc_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_Aloc_mem</td>
</tr>
</tbody>
</table>
Overview

• Introduction
• Motivation
• Design Flow
• An Example of Portability and Scalability
• Architecture
• Current Functionality
• Performance
• Conclusion
Performance

- Experiments on latency and bandwidth were performed
- Two ranks exchange round-trip messages of increasing size
- Each round-trip is repeated 10 times
- Elapsed time is measured with MPI_Wtime()
- Two ranks are implemented in software (x86) and hardware (hardware engine plus MPE)
- HW engine is hand-written in VHDL and follows the same algorithm as the C code
- The two ranks give following combinations: X86-X86, X86-HW and HW-HW
Performance: Latency

Fig. 14. Round-trip experiments between two ranks mapped to different types and locations.
**Latency**

- Latency is the predominant factor in comm. time of small messages
- Rendezvous protocol is always used in TMD-MPI
- Lowest to highest:
  - intra-m2b
  - x86-x86
  - x86-m2b
  - inter-m2b

Fig. 15. Latency and bandwidth performance measurements.
Bandwidth

- Three factors in performance:
  1. Physical structure of the communication channel:
     - x86
     - Speed of memory RW
     - NoC of FPGA
     - 32-bits wide at 133 MHz
     - Peak BW 533.33 MB/s
  2. Synchronization
     - Affects how fast saturation is reached
  3. Packetization/Depacketization overhead
Packetization/Depacketization overhead

- Once BW is saturated, the difference compared to peak is explained by the packetization/depacketization process.
- The smaller the packet, the higher the overhead.
- Larger packets achieve better BW, but are more resource intensive (memory and logic resources).
- x86-x86 numbers are noisy due to OS context switching and the usage of cache.

Fig. 15. Latency and bandwidth performance measurements.
Overview

- Introduction
- Motivation
- Design Flow
- An Example of Portability and Scalability
- Architecture
- Current Functionality
- Performance
- Conclusion
Conclusion

• This paper has done the following:
  – Introduce High Performance Reconfigurable Clusters (HPRCs), a mixture of x86 and FPGA accelerators
  – Introduce TMD-MPI, an MPI implementation which abstracts communication between hardware (FPGAs) and software (x86)
  – Obtain performance measurements of TMD-MPI on the University of Toronto TMD Machine, for which TMD-MPI was created
  – Prove the portability of TMD-MPI by explaining how an application was developed despite changing FPGA hardware
A Scalable Runtime for the ECOSCALE
Heterogeneous Exascale Hardware Platform

Paul Harvey, Konstantin Bakanov, Ivor
Spence, Dimitrios S. Nikolopoulos
Queen’s University Belfast University Road
Belfast, United Kingdom

paul@paul-harvey.org, {k.bakanov,
i.spence, d.Nikolopoulos}@qub.ac.uk
Outline

• Introduction
• Background
• Ecoscale Hardware
• Ecoscale Framework
• Conclusion
Introduction

- Applications are becoming more demanding of computing resources
- HPC systems need to keep up
- We are moving towards the exascale era: a billion billion FLOPs
- To get there, simply increasing the number of hardware devices is not an option
  - Non-linear scaling of power consumption, heat generation, and cost
- More power-efficient platforms are required
- HPCs are becoming more heterogenous to satisfy power-efficiency
- Therefore, newer programming models and runtimes are required
Introduction (Cont.)

• This paper introduces the Ecoscale Runtime for a new heterogenous platform.
• The runtime is based on OpenCL and expands upon it with the Partitioned Global Address Space (PGAS) model in order to break the one-to-one association between hardware devices and the current granularity of work within OpenCL.
• In addition, the runtime provides automated placement of data and computation across a heterogenous platform with a focus on FPGAs.
Outline

• Introduction
• Background
• Ecoscale Hardware
• Ecoscale Framework
• Conclusion
Background

• OpenCL Background
  – Programming framework for heterogenous computing
  – Programmers required to think in terms of host and device code
    • Devices may be CPUs, GPUs, FPGA, or a co-processor such as the Xeon Phi
  – The host will set up, dispatch, and collect results from the device

• OpenCL Configuration
  – The user queries for devices and creates a context for the device
  – A command queue is created for the device—these contain device info queries, memory management commands, and kernel invocation commands
  – A kernel object is created at runtime by compiling source file
  – The kernel is then invoked, and the host is blocked until completion
Background

• OpenCL Kernels
  – C-like piece of code run on the device
  – Memory hierarchy
    • global: accessible from all threads
    • local: accessible from within the thread block
    • private: accessible within only a single thread
    • constant: accessible from all threads, read-only

```
__kernel void square(__global float* input,
                      __global float* output,
                      const unsigned int count){
    int i = get_global_id(0);
    if(i < count)
        output[i] = input[i] * input[i];
}
```

Listing 1: OpenCL Kernel to Compute the Square of An Input Array
Background

• Location-Transparent Accelerators
  – OpenCL was designed for a single node with multiple accelerators
  – This will not work for a distributed system
  – SnuCL
    • Allows location-transparent (across multiple nodes) execution using MPI.
    • Downside: does not allow Shared Virtual Memory between hosts and devices
    • Also, requires each device to be manually configured, limiting scalability
  – VOCL
    • Similar to SnuCL
    • However, the mapping between OpenCL software devices and physical hardware devices is still one-to-one
Background

• Scheduling of Kernels
  – FluidiCL
    • Dynamically schedules kernels between CPU and GPU to improve performance
    • Finds the optimal device to run on
    • Downside: requires developers to add synchronization points manually within the kernel. Also, FluidiCL only works for CPU and GPU, so it is not suitable for the hardware in this paper
  – Shepard
    • Assumes access to a standard library of kernels that developers group into tasks
    • The runtime decides where to execute these kernels based on the data size and profiling of the kernel.
    • Downside: Shepard schedules at the kernel-level, so platforms with multiple devices may be under-utilized
Background

- **Data-Partitioning**
  - As HPC applications become more data-intensive, data placement across multiple hardware devices becomes increasingly challenging for the programmer to orchestrate.
  - Yan et al. have created an extension to specify data-partitioning:
    - Replicate: A copy of the data is sent to each device
    - Block: Data is divided into equal, contiguous blocks. Each device gets a block.
    - Cyclic: Every $i^{th}$ block maps to the $i^{th}$ device.
  - This is the approach taken by this paper.
Takeaways:

- There are no dynamic kernel schedulers for OpenCL that support FPGAs.
- There is no straightforward programming approach to take advantage of heterogenous hardware, distributed or otherwise.
Outline

• Introduction
• Background
• **Ecoscale Hardware**
• Ecoscale Framework
• Conclusion
**Ecoscale Hardware**

- The Ecoscale project is developing a novel system architecture with CPUs, memory, and FPGAs in a highly parallel manner.
- Hardware resources are divided into NODES which are further divided into WORKERS.

*Figure 2: ECOSCALE Hardware Architecture*
Ecoscale Hardware

- Each NODE has
  - UNIMEM: a shared partitioned global address space (PGAS) to communicate via regular load-stores without global cache coherence
  - UNILOGIC: a shared partitioned and reconfigurable set of resources which share the UNIMEM space with software tasks
Ecoscale Hardware

• Each NODE has
  – UNIMEM: a shared partitioned global address space (PGAS) to communicate via regular load-stores without global cache coherence
  – UNILOGIC: a shared partitioned and reconfigurable set of resources which share the UNIMEM space with software tasks

![Figure 2: ECOSCALE Hardware Architecture](image-url)
Outline

• Introduction
• Background
• Ecoscale Hardware
• **Ecoscale Framework**
• Conclusion
Ecoscale Framework

- OpenCL by default has a one-to-one mapping between accelerators (devices) and their respective OpenCL software abstractions
  - The current range of software abstractions are DEFAULT, CPU, GPU, and ACCELERATOR
- This paper introduces software abstractions for a one-to-many mapping:
  - Devices aggregated by the WORKER
  - Devices aggregated by the NODE
- The motivation behind this is to allow the runtime to best schedule kernels among the devices
Ecoscale Framework

• OpenCL Extensions:
  – With the WORKER and NODE level abstractions, the scheduler must decide the optimal kernel placement
  – The programmer can assist with this with the #pragma eco <device> directive
  – The programmer can also assist with data partitioning
    • For example, clCreateBuffer(context, REPLICATE,...);

• These extensions are optional—however, in certain cases application performance will improve with their use
Ecoscale Framework

• Software Architecture:
  – One WORKER per NODE is used as a controller
  – Each WORKER has an Operating System and an Ecoscale runtime
  – The runtime consists of:
    • A communication library
    • A scheduler
    • A log of performance metrics
    • Access to FPGA bitstreams and their associated metrics
    • An online machine learning component
Ecoscale Framework

• Unit of Schedulability
  – The programmer uses the new abstractions when submitting kernels, and the runtime will divide them into workgroups (which need not be from the same kernel)
  – The scheduler can run workgroups on whichever device it sees best fit

• No Preemption
  – Workgroups run until completion because they have a shorter lifetime and context switching is expensive
Outline

- Introduction
- Background
- Ecoscale Hardware
- Ecoscale Framework
- Conclusion
Conclusion

- This paper introduces the Ecoscale runtime which targets heterogenous high performance computing platforms
- By extending OpenCL, this work enables the performance and ease of programming offered by OpenCL as well as energy efficiency, compute power, and reconfigurable computing offered by the Ecoscale hardware platform
- The ability to transparently schedule OpenCL kernels at the level of workgroups across heterogeneous hardware platforms is a key innovation that will enable OpenCL applications to take advantage of current and next generation HPC platforms
Mellanox Innova™-2 Flex Open Programmable SmartNIC
Mellanox Innova™-2 Flex

- The Innova-2 Flex is “designed to accelerate a wide range of applications, including Security, Storage, Artificial Intelligence, Media & Entertainment, Cloud and more”
- The card has a Connect-X 5 network controller that enables RDMA
- The addition of an on-board FPGA provides maximum flexibility to offload the CPU by migrating portions of data-processing logic to the FPGA.
Thank You!

Network-Based Computing Laboratory
http://nowlab.cse.ohio-state.edu/

The High-Performance MPI/PGAS Project
http://mvapich.cse.ohio-state.edu/

The High-Performance Big Data Project
http://hibd.cse.ohio-state.edu/

The High-Performance Deep Learning Project
http://hidl.cse.ohio-state.edu/