

**A Parallel Programming Model for a Multi-FPGA Multiprocessor
Machine**

by

Manuel Alejandro Saldaña De Fuentes

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Sciences
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2006 by Manuel Alejandro Saldaña De Fuentes

A Parallel Programming Model for a Multi-FPGA Multiprocessor Machine

Manuel Alejandro Saldaña De Fuentes

Master of Applied Sciences

Graduate Department of Electrical and Computer Engineering

University of Toronto

2006

Abstract

Recent research has shown that FPGAs can execute certain applications significantly faster than state-of-the-art processors. The penalty is the loss of generality, but the reconfigurability of FPGAs allows them to be reprogrammed for other applications. Therefore, an efficient programming model and a flexible design flow are paramount for FPGA technology to be more widely accepted.

In this thesis, a lightweight subset implementation of the MPI standard, called TMD-MPI, is presented. TMD-MPI provides a programming model capable of using multiple-FPGAs and embedded processors while hiding hardware complexities from the programmer, facilitating the development of parallel code and promoting code portability.

A message-passing engine (TMD-MPE) is also developed to encapsulate the TMD-MPI functionality in hardware. TMD-MPE enables the communication between hardware engines and embedded processors. In addition, a Network-on-Chip is designed to enable intra-FPGA and inter-FPGA communications. Together, TMD-MPI, TMD-MPE and the network provide a flexible design flow for Multiprocessor System-on-Chip design.

Dedication

I would like to dedicate, not just this thesis, but the last two years of my life of perseverance and hard work to my grandmother (Memé) for having me in her prayers and because I love her so much. First and foremost, I would like to thank to my dear Claudia, for giving me wings and encouraging me all this time we have been apart. I also would like to thank to my parents and my family for all their love and support, because I never felt alone. Special thanks to my supervisor Prof. Paul Chow for his patience, guidance, technical feedback and giving me the confidence for writing papers. Thank you for giving me the opportunity to be part of this wonderful project. I also thank others in my research group, including Chris Comis that gave me the abc of the lab, Lesley, Arun, Chris Madill, Andrew, Daniel, Emanuel, Richard and Tom for their insightful comments and help. It was a great working environment. I would like to thank to my Mexican friends in Toronto Laura, Gaby, Flor, Jessica, Jorge, Miguel, Daniel, Juan and Misael; also to my Latin-American friends Jenny, Abby, and David for being there when I needed them the most. I am particularly grateful to Patricia Suárez for reminding me of my social responsibility as Engineer. Finally, thank you to my friends at Massey College because they made these last two years more enjoyable and an invaluable life-learning experience.

Acknowledgements

I would like to acknowledge to the institutions that supported me and my research with funding and infrastructure. In particular to the Mexican National Council for Science and Technology (CONACYT), the National Science Foundation for Education in Mexico (FUNED), the Edward S. Rogers Sr. Graduate Scholarship, the University of Toronto, the Canadian Microelectronics Corporation (CMC) through the System-On-Chip Research Network (SoCRN), Xilinx, Amirix, Massey College, my supervisor Prof. Paul Chow, and my family.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Research Contributions	2
1.3	Thesis Organization	3
2	Background	4
2.1	Classes of Machines	4
2.2	Parallel Programming	5
2.2.1	Message Passing	6
2.2.2	MPI	7
2.2.3	MPICH	9
2.3	The Abstraction of Communications	10
2.4	Related Work	11
3	Hardware Testbed	14
3.1	System Overview	14
3.2	The Processing Elements	16
3.3	The Network Components	17
3.3.1	The Network Interface	21
3.3.2	The Network Bridge	25
4	Message Passing Implementation	27
4.1	The Programming Model	27
4.2	Design Flow	28
4.3	TMD-MPI: Software Implementation	31
4.3.1	A Layered Approach	33
4.3.2	Rendezvous vs Eager Protocol	34
4.3.3	Message Queues	36

4.3.4	Packetizing and Depacketizing	37
4.4	TMD-MPE: Hardware Implementation	38
4.5	Computing Node Configurations	41
5	Testing the System	43
5.1	Testing TMD-MPI Communication Characteristics	43
5.1.1	Latency and Bandwidth	44
5.1.2	Measured Bandwidth with Contention	46
5.1.3	Synchronization Performance	47
5.2	The Heat Equation Application	47
5.2.1	Performance Results	49
5.2.2	Performance Test	49
5.2.3	Scalability Experiment	51
6	Conclusions	53
6.1	Future Work	54
	Appendices	56
A	Tier 2 Network for the 45-Processor System	57
B	Physical location of the MGT connectors on the AP1100 boards	58
C	TMD-MPI Manual	59
D	TMD Configuration using the PCI Bus	68
D.1	Extension to the EDK makefile	68
D.2	Commands to use the PCI interface	69
	Bibliography	71

List of Tables

5.1 Experiment configurations 50

List of Figures

2.1	Three Classes of Computing Machines	5
2.2	Simplified image of the MPICH architecture.	10
3.1	Interconnection of AP1100 boards in the Class 3 machine testbed	15
3.2	Multiprocessor System on Multiple FPGAs	18
3.3	Path of a packet from one FPGA to another	19
3.4	Packet Formats	20
3.5	Broadcast-based Network Interface	23
3.6	Selective Network Interface	24
3.7	The Network Bridge block diagram	26
4.1	Design Flow with MPI	30
4.2	Implementation Layers	34
4.3	Difference between Rendezvous and Eager protocols.	35
4.4	Type of packets used by TMD-MPI	36
4.5	Queue of envelopes from unexpected messages.	37
4.6	Packetizing process of a large message into smaller packets.	38
4.7	TMD-MPE block diagram	40
4.8	TMD-MPE communication protocol with the associated computing unit.	41
4.9	Different node configurations based on the use, or not, of TMD-MPE	42
5.1	Experimental testbed hardware system	45
5.2	Measured link bandwidth under no-traffic conditions	46
5.3	Barrier Synchronization Overhead	48
5.4	Data decomposition and point-to-point communication in Jacobi algorithm	48
5.5	Main loop execution time of different Multiprocessor configurations	51
5.6	Speedup of Jacobi iterations in the 45-node multiprocessor system	52
A.1	Physical interconnection diagram of Tier 2 network for the 5-FPGA system with 45 processors.	57

B.1 Physical location of the MGT connectors and its naming convention	58
---	----

Glossary

This glossary contains some of the acronyms used in this thesis.

TMD. Originally meant *Toronto Molecular Dynamics* machine, but this definition was rescinded as the platform is not limited to Molecular Dynamics. The name was kept in homage to earlier TM-series projects at the University of Toronto.

MPSoC. Multiprocessor System-on-Chip

SoC. System-on-Chip

MGT. MultiGigabit Transceiver

FSL. Fast Simplex Link

NoC. Network-on-Chip

MPI. Message Passing Interface

OCCC. Off-Chip Communication Controller

NetIf. Network Interface

EOP. End of Packet

SOP. Start of Packet

FIFO. First-In-First-Out

SPMD. Single-Program-Multiple-Data

MPMD. Multiple-Program-Multiple-Data

API. Application Program Interface

ADI. Abstract Device Interface

Chapter 1

Introduction

Recent research has shown that three major challenges in multiprocessor system-on-chip design are the programming model, the design flow and the communications infrastructure. Addressing successfully these issues would result in a wider acceptance of this technology that intends to alleviate more fundamental problems of current computing architectures, such as power consumption, design cost, faster execution times, and shorter development cycles.

This chapter provides more details of the motivation of this thesis in Section 1.1. Section 1.2 outlines the main contributions of this work, and Section 1.3 presents a summary of the content of the remaining chapters of this thesis.

1.1 Motivation

As the level of integration in modern silicon chips increases, the computing throughput requirements of new applications also increases. Currently, the limiting factor of modern processors is not the number of transistors-per-chip available but the wire delay and power consumption issues caused by high frequency operation. An alternative to alleviate this problem is the inclusion of multiple, lower-frequency, specialized computing cores per chip. For example, microprocessor manufacturers, such as Intel, Sun Microsystems, IBM and AMD have announced their multi-core microprocessors [1], although they incorporate only a few general-purpose, homogeneous cores. Another example of low frequency operation is the IBM Blue Gene supercomputer, which has up to 32768 PowerPC processors, all of them operating at only 700 MHz [2]. The Blue/Gene supercomputer takes advantage of specialization by using one of the four processors per node as a communication coprocessor for the node. Also, this supercomputer has five different, specialized networks (Collective Tree network, 3D-Toroidal mesh, Gigabit Ethernet for I/O, Barrier and Interrupt Network, Control Network). With this particular architecture, the Blue-Gene Supercomputer is currently in first place of the TOP500 list [3] of

the fastest computers in the world.

Besides multiple low-frequency cores, heterogeneous cores can provide further power savings by specialization [4]. Application-specific hardware for a particular algorithm can significantly improve the execution speed with lower frequencies. Specialized computing cores can be designed and implemented in FPGAs and their flexibility allows programmers to reconfigure them as needed. Some high-performance computer vendors are including FPGAs in their machines. For example, Cray has the XD1 supercomputer [5], SGI has the RASC technology in Altix Servers [6], and SRC Computers has their MAP processors [7]. AMD's Opteron processors can be connected to FPGAs using the hypertransport protocol [8,9]. FPGAs have evolved from just glue logic to embedded systems to entire Systems-on-Chip and recently to high-performance computing elements. Initiatives such as the FPGA High-Performance Computing Alliance [10] and OpenFPGA [11] are examples of research efforts to incorporate FPGAs into demanding computing applications.

The resource capacity of modern FPGAs is enough to implement multiprocessor systems-on-chip (MPSoC), and interconnecting several large FPGAs provides vast amount of resources to exploit coarse-grain and fine-grain parallelism on the scale of entire applications. Multi-FPGA MPSoCs are an alternative computing architecture for certain types of applications. The TMD project [12] at the University of Toronto is a multi-FPGA research platform intended to accelerate molecular dynamics simulations, which are applications extremely demanding on computing power and a high computation-to-communication ratio. However, fast communications, an efficient programming model, and a flexible hardware-software co-design flow are mandatory to be able to use all the resources available in an efficient and effective manner. It is also important to have tools to measure the performance of the architecture to set a reference point for future improvements on the architecture. Therefore, the motivation of this work comes from the need to provide the TMD platform with these key elements.

1.2 Research Contributions

The novelty and contributions of this thesis are derived from the integration of ideas from different research fields rather than offer the ultimate breakthrough in a specific field. However, by applying known concepts in different ways and adapting existing technology to particular needs, it has been possible to create a novel, working framework for multi-FPGA MPSoC research.

This thesis provides the TMD platform with a programming model based on the existing MPI message-passing model (see Section 2.2). We chose MPI because it is very popular for programming distributed memory machines. MPI has been successful in the software do-

main, and this thesis presents how MPI can be used in an multi-FPGA hardware environment. The MPI implementation presented in this thesis (TMD-MPI) is adapted such that it can be used with embedded processors and hardware engines by building a message-passing engine (TMD-MPE). This thesis also provides a network infrastructure, a design flow, and initial performance measurements of the TMD architecture. With this working environment it has been possible to easily port an entire Linux-based application to the TMD architecture implementing a 45-processor system across five FPGAs. Also, with TMD-MPI and TMD-MPE, it has been possible to build heterogeneous computing systems using embedded processors and dedicated hardware engines to cooperate in solving a parallel application.

The programming model and the network infrastructure provide a homogeneous view of the system, giving the impression to the programmer that the system is built entirely in one single, large FPGA, hiding unnecessary hardware complexities. With this working platform future students at the University of Toronto can focus on optimizing and investigating improvements or alternative approaches for the elements presented in this thesis.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides a brief introduction to ideas and related work important to better understand the contents of this thesis. First, this chapter introduces a classification of computing architectures to position this research within a big picture. Then there is a summary of parallel programming using message-passing with MPI, followed by an overview about the abstraction of communications. Chapter 3 presents the prototype of a multi-FPGA computing system (TMD platform) used to experiment with the programming model, design flow and network-on-chip design. This chapter describes the network infrastructure and the configuration process of this prototype. Then Chapter 4 presents the programming model and the design flow to create applications on top of this hardware infrastructure. Implementation details of TMD-MPI and TMD-MPE are discussed. Chapter 5 describes the tests run on the TMD prototype. First, there is a test for the communication system extracting performance measurements such as link latency, bandwidth, and synchronization cost. The second test is more complex because it involves the test of the design flow and programming model by porting a Linux-based application to the TMD platform. Finally, Chapter 6 provides some conclusions and suggests future directions on this research to improve the performance of the TMD prototype.

In the appendix section there is helpful information to setup the testbed system, including the physical location and interconnection of cables, and a guide to install and configure the TMD-MPI library.

Chapter 2

Background

This chapter presents a brief overview of key concepts to better understand this work, and to establish a context for the ideas presented in this thesis. Each section describes parts of related knowledge and presents some definitions that will be used in the rest of this document. Section 2.1 describes a classification of the target computing architectures for the programming model. Section 2.2 is a brief introduction to parallel programming using the message-passing paradigm, which is the model used by the TMD prototype. Section 2.3 explains the basic principle that enables the communications in the TMD prototype. And finally, Section 2.4 discusses related work and other research efforts in the field of programming models for MPSoC using message-passing.

2.1 Classes of Machines

A categorization for the types of high-performance machines was first introduced in Patel *et al.* [12]. As previously stated, FPGAs are now an active field of research in the high-performance community, and based on the trend to incorporate FPGAs in computation, high-performance machines are classified into three categories, as shown in Figure 2.1.

The first class of machines consists of current supercomputers and clusters of workstations without any involvement of FPGAs in the computation. Typical programming of such machines is coarse-grain parallel programming using message passing or shared memory libraries depending on how the memory is accessed by the CPUs. Examples of these machines are the IBM Blue Gene/L [2] and the Cray T3 Supercomputer [13]

The second class of computers are those that connect FPGAs to CPUs, such machines can execute application kernels in an FPGA and take advantage of the bit-level parallelism and specialized hardware computing engines implemented in the FPGA fabric. Coarse grain parallelism can be achieved by using a cluster of machines, each one with its own FPGA [5–9, 14].

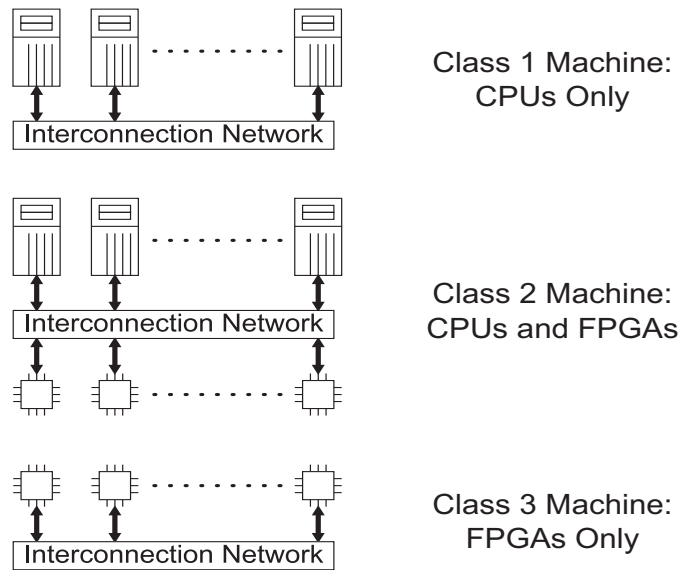


Figure 2.1: Three Classes of Computing Machines

The penalty in performance for Class 2 machines comes from the high latency communication between the CPUs and the FPGA. The programming model for these machines is based on libraries to send data to the FPGA to be processed and to receive the results; the FPGA acts as a co-processor.

The third class of machines comprises purely FPGA-based computers. These machines provide a tight integration between processors and hardware engines because processors can be embedded into the FPGA fabric. Although these embedded processors are not as sophisticated as the state-of-the-art superscalar processors, they can adequately perform control, I/O tasks and some computations. Time critical computations can be performed in special hardware engines that can exceed in performance the fastest processor available with less power consumption. Class 3 machines open the possibility of exploring heterogeneous computer architectures to more efficiently use the resources at hand by letting specialized hardware blocks do what they do best. Although Class 3 machines are application-specific in nature, reconfigurability is implicit in the FPGAs and they can be reconfigured as needed. Examples of Class 3 machines are BEE [15] and the one presented by Cathey et. al [16]. The focus of this work is on implementing the communication scheme and a programming model for Class 3 machines.

2.2 Parallel Programming

Parallel programming is, by definition, the execution of concurrent tasks to solve a particular problem. A task is a reasonable coarse-grain set of operations of an application. For example,

a few boolean logic operations is not advisable to consider as a task, but the computation of the inverse of a matrix could be. Not all the tasks in an application can be executed in parallel, because there are tasks that depend on the output from other tasks. This divides the application code into sequential sections and parallel sections.

The concurrency of parallel sections can be exploited at different levels and with different levels of involvement of the programmer. At one end of the spectrum is the *implicit parallelism*, which requires no intervention from the programmer because the compiler or hardware automatically detect and exploit the available parallelism. On the other end of the spectrum, *explicit parallelism* is exploited by manually coding parallel instructions into the source code, requiring a less sophisticated compiler but more user intervention.

An example of *implicit parallelism* is Instruction Level Parallelism [17] (ILP), which takes advantage of multithreaded, multiple-issue, pipelined processors to execute several instructions at once without user involvement. *Implicit parallelism* takes place at lower levels in a system because it depends on the complexity of the processing unit and the compiler. However, ILP is limited by code dependencies and not much parallelism can be exploited [18]. Conversely, *explicit parallelism* takes place at a high-level view of the problem, and relies on the entire system architecture, which includes the number of processing units, the interconnection network between processing units, and the memory distribution over the entire system. In *explicit parallelism*, the algorithm has to be designed to execute in parallel, which requires more dedication from the programmer. There are advantages and disadvantages in both approaches, and they are complementary rather than exclusive. A complete discussion about this is beyond the scope of this thesis.

However, the present work is focused on the explicit side of parallelism because communications are precisely stated, which is similar to hardware design. Hardware circuits are essentially computing units that execute concurrently. A circuit can be viewed as a task and signals between circuits can be viewed as messages. Within the high-performance computing domain, the two most common paradigms to program explicitly-parallel applications are shared memory and message passing, but the later is a more natural analogy to modular hardware design than the former. This work is about applying a well-known parallel programming paradigm (message-passing) to MPSoC design by representing concurrent tasks that can be executed either in processors or specialized hardware engines.

2.2.1 Message Passing

In the message-passing model, the concurrent tasks are executed typically in computing units that communicate with each other by explicitly interchanging messages between them. These

messages can be data to be processed, or control messages, such as synchronization messages. Message passing assumes that each processing unit has all the data required stored in local memory that is not globally accessible. In explicit parallelism, it is the programmer's responsibility to orchestrate the processing units to efficiently and correctly solve a problem. Despite the manual intervention of the programmer to create a parallel application, message passing is the most common paradigm used in the high-performance community because it is more scalable than the shared memory paradigm and can be used in commodity clusters of workstations providing a low cost/performance ratio.

In SoC design, having shared memory or shared buses between processing units leads to inefficient communications due to bus congestion [19]. A network-on-chip using message-passing can help relieve this bottleneck. Distributed memory machines, such as the grid infrastructure, some supercomputers, and clusters of commodity computers are example computing systems that operate based on the concept of message-passing. This broad spectrum of target architectures makes message passing widely accepted in academia and industry. The present work shows how message-passing can be used as a programming model that enables a flexible software-hardware co-design flow for Class 3 machines.

Typically, in the high-performance domain, the programmer uses an application program interface (API) to explicitly code a parallel application. Examples of message-passing APIs are PVM [20], MPI [21], and P4 [22]. They all provide message-passing capabilities but they differ in the way they do it. However, the greatest advantage for MPI over the other two, is that MPI is a standard interface; therefore, more popular.

Message passing implementations require only two basic primitives: *send* and *receive*. These primitives are always used in pairs. The source task would have to execute a *send* operation to transfer data to a destination task, which, in turn, would have to execute a *receive* operation to receive the data from the source task. Based on these two primitives, more complex communication functions can be implemented. Other parameters can be specified in a send/receive API, such as a message identifier, the type of the data and the amount of data being transferred, but they are API-specific details.

2.2.2 MPI

MPI is a standard, public-domain, platform-independent interface for message-passing programming. MPI was defined in 1992, and currently it is maintained by the MPI Forum [23]. MPI is the result of a joint effort between academia and industry to create a portable message-passing API specification. As a standard, MPI does not rely on a particular research group or a particular company to be defined or evolved, which brings certainty in its use. MPI standardizes

the syntax and semantics of each API function without imposing any particular implementation style or architectural requirement. This has been a key element to its popularity because implementations of the standard can be optimized for a particular architecture and take advantage of specialized communications hardware.

Before MPI, each supercomputer vendor provided their own APIs to program their computers. As a consequence, there was a lack of portability in scientific codes, and a prolonged learning-curve to write parallel applications. After MPI, numerous scientific applications have been developed and ported to different high-performance computing systems allowing cooperation between research groups regardless of the difference in computing hardware infrastructure. Most notably, MPI allows the creation of parallel scientific software libraries, which can be reused and reduce the development times. MPI's latest version is MPI-2, although the first version (MPI-1) is enough to implement most scientific applications. Moreover, the MPI standard is quite vast, but not all the features are required to program a parallel application. Only six functions are essential to code a parallel application. The remaining functions implement more complex communication operations that facilitate the programming of an application; for example, by saving coding lines for the user, and by providing more powerful abstractions of communication, such as reduction operations or optimized communications based on a defined interconnection topology between processes.

MPI communication functions can be classified into point-to-point and collective functions. Point-to-point communication involves only two nodes: the *sender* and the *receiver*. Collective communication involves all the nodes in the application; typically, a *root* node that coordinates the communication and the remaining nodes that just participate in the collective operation. Examples of point-to-point functions are *MPI_Send* and *MPI_Recv* to send and receive data, respectively. Examples of collective operations are *MPI_Barrier* and *MPI_Bcast* to synchronize all the processors, and to broadcast information in the system, respectively. Another way to classify MPI communications is *synchronous* and *asynchronous*. In *synchronous* communication, the *sender* and the *receiver* block the execution of the program until the information transfer is complete. *Asynchronous* communication allows the overlap of communication and computation, but it is the responsibility of the programmer to avoid the corruption of the data in memory, because the data transmission takes place in the background and the transmission buffer can be overwritten by further computation.

MPI uses three parameters to identify a message: the source/destination rank, the message tag, and the communicator. The *rank* of a process is a unique number that identifies that particular process in a parallel program. The *ranks* are numbered consecutively, starting at zero and up to *size-1*, where *size* is the total number of software processes (tasks) in the application. The function *MPI_Rank* returns the rank number of a process. In this way, a process

can make decisions about what data to process or what to do based on the rank number. The *tag* is an integer number that identifies a message. The tag can be used to pass extra information about a message; the meaning of a *tag* is entirely user definable. The *rank* of a process and the message *tags* are scoped to a *communicator*, which is a communication context that allows MPI to selectively communicate to a group of processes within a particular context. Communicators are especially helpful for creating parallel libraries because they isolate the main algorithm's messages from the library's messages. It is important that the three message parameters (source/destination rank, the tag, and the communicator) match in any message-passing operation; otherwise, the application may stall because the receiver will be waiting for the wrong message.

Since MPI is a freely available, standard specification, there are a number of implementations of the standard with different characteristics and performance. The MPI standard just specifies the behavior of the functions, it is the *MPI implementation* that actually performs the data transfer and deals with the hardware or any other lower level software, such as the operating system. Examples of public domain MPI implementations are MPICH [24], LAM [25], and OpenMPI [26]. Other optimized MPI implementations are provided by computer vendors to use more efficiently their platforms. To develop TMD-MPI, MPICH was used as a reference because of its popularity and because of its straightforward layered approach, which is described next.

2.2.3 MPICH

MPICH is a freely available, robust, portable implementation of MPI, and it has been ported to most variants of UNIX. The TMD-MPI implementation presented in this thesis is influenced by MPICH. The layered approach of MPICH is of particular interest because it allows portability, modularity and an incremental approach that may absorb the changes in an essentially changing underlying hardware architecture, such as an FPGA. MPICH can be ported to a new platform by changing only the lower layers of the implementation. Moreover, the MPICH layered approach can be mapped quite naturally to the communication abstraction implemented in our Class 3 machine, which will be explained in Section 2.3.

Figure 2.2 shows a simplified view of the MPICH layered architecture. The first layer (*MPI Layer*) provides all of the MPI API to the programmer. Collective functions are expressed in terms of point-to-point MPI functions from the second layer. The central mechanism for achieving portability and performance is a specification called the abstract device interface (ADI) [27]. All MPI functions are implemented in terms of the macros and functions specified in the ADI, i.e., the point-to-point functions in layer two are expressed in terms of the ADI

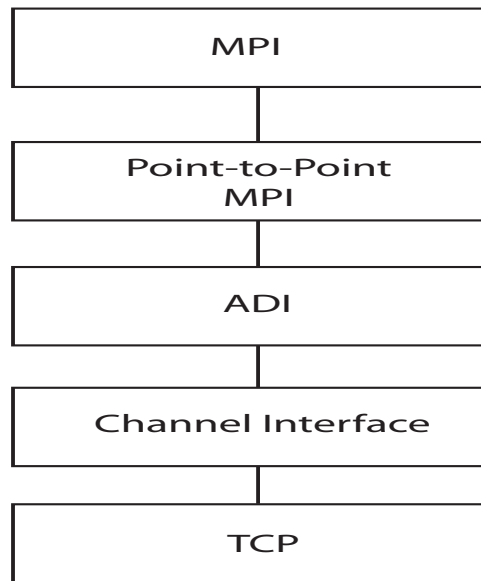


Figure 2.2: Simplified image of the MPICH architecture.

functions. The ADI has the responsibility for handling the message queues, packetizing the messages, attaching headers info, and matching sends and receives. One implementation of the ADI is in terms of a lower-level interface called the *channel interface* [28]. The channel interface can be extremely small (at least five functions), and it can be expanded to gradually include specialized implementations of more of the ADI functionality. The Channel Interface is in charge of buffer management and flow control. Finally, the Channel Interface can be implemented using the computer vendor communication primitives or a common library, such as TCP, which provides the *write* and *read* functions for sockets. Porting MPICH to a new platform (not radically different), would require only the modification of the Channel Interface and the rest of the layers can remain the same. Chapter 4 provides an explanation of how the functionality of MPICH's layers is implemented in the layers of TMD-MPI.

2.3 The Abstraction of Communications

To use a message-passing implementation, an underlying communication infrastructure is required. In Class 1 machines, in the case of supercomputers, the network infrastructure is typically proprietary. In the case of clusters of computers, Infiniband or Gigabit Ethernet switches are typically used. In the case of Class 2 machines, a hybrid scheme is used to interconnect the CPUs and the FPGAs. For Class 1 machines, the interconnection between processors is generic. All the nodes can communicate with all the other nodes, regardless of the communication pattern of the application. In other words, generic architectures have not only generic

processing units, but also generic networks. Current Class 2 machines connect FPGAs as slaves to a processor, so the FPGAs are not usually treated as peers to the processors on a communication network. In Class 3 machines, a special purpose network is defined to suit the needs of a particular application. An application specific topology is defined by the application's communication pattern, and low overhead network components are required. In reconfigurable, application-specific computers, all the overhead of generality in communications can be reduced. The design is simplified and the message latency is decreased.

In a Class 3 machine, the network infrastructure enables the communications between computing elements within an FPGA and between FPGAs. Three network tiers are identified and must be made to provide the view of a single, extensible FPGA fabric to the programmer. The Tier 1 network is the Network-on-Chip inside each FPGA and it is used for intra-FPGA communications between processors and hardware blocks. This network is based on point-to-point unidirectional links implemented as FIFOs. The FIFOs have a master interface and a slave interface. The master interface is typically connected to the source element that contains the data to be transferred. A *write* signal has to be asserted high during one cycle to queue the value into the FIFO. The slave side of the FIFO is typically connected to the destination element that is meant to receive the data. A *read* signal has to be asserted during one cycle to de-queue the data value. This simplifies the design of hardware blocks and provides isolation between them, allowing the use of multiple clock domains by using asynchronous FIFOs. For example, the network can operate at one clock frequency and the computing elements at a different frequency. Moreover, hardware engines can be clocked faster than processors. In general, the FIFO is a powerful abstraction for on-chip communication [29].

Several FPGAs can be placed together on the same printed circuit board to form a cluster. The Tier 2 network would be for inter-FPGA communication on the same board using high-speed serial I/O links, which simplifies the circuit board design and increases the chances to include more FPGAs per board. The Tier 3 network is reserved for inter-cluster communication allowing the use of several clusters of boards of FPGAs providing a massive amount of resources. On top of this network infrastructure, a programming model can be developed to provide a homogeneous view of the entire system to the programmer.

2.4 Related Work

This thesis involves concepts of single-chip multiprocessors and their programming models. Each concept is in itself a current field of research. A complete literature review of all the concepts would be very extensive. Instead, this section presents various pieces of research that were most influential for the conception of the ideas presented in this work.

An outstanding compendium of analysis about MPSoC from different authors can be found in Jerraya *et al.* [30]. The authors claim that despite the similarities between MPSoC and typical multiprocessor computers, the traditional scientific computation model cannot be applied directly to MPSoC because SoC operate under different constraints. Such constraints can be real-time computation, area-efficiency, energy-efficiency, proper I/O connections, and scarce memory resources. For example, a supercomputer can rely on buffering data to accelerate communications, but in an embedded system, memory is not abundant. In addition, some researchers have said that a generic programming model will lead to inefficient solutions given the application-specific nature of MPSoC [31]. According to this, MPI should not be used because it is a generic programming model. On the other hand, in recent research several authors have proposed to use MPI or any other message-passing programming model for System-on-Chip designs [32]. Research in this field is still ongoing and there are many challenges to solve.

Some of the proposals to provide message-passing software and hardware rely on the existence of an underlying operating system [33], which is an overhead for memory and performance to each processor. There are other message-passing implementations, but they are designed for real-time embedded systems [34] adding quality of service, but still relying on an operating system

There is research that is not using MPI as the preferred API. Poletti *et al.* [35] present another message passing implementation for SoC, but it does not use a standard application program interface. Other approaches use a CORBA-like interface and DCOM object model [36] to implement message passing. The authors also develop a message-passing hardware engine for an MPSoC to translate the format of messages between hardware cores. This work is interesting because it provides an alternative generic programming model, and it also investigates the hardware support.

Other approaches provide only simulation environments. Youssef *et al.* [32] use a SystemC model of `MPI_Send` and `MPI_Recv` primitives to refine and debug a C code by performing simulations of an OpenDivX encoder system that also uses MPI as the communication abstraction. Such simulation aids could be helpful to quickly prototype an application and express explicitly the communications. However, this simulation environment is not enough to execute a typical MPI application written in C code without modifications in a multi-FPGA system.

Aggarwal *et al.* [37] present a Class 2 machine that incorporates FPGA hardware for acceleration purposes. They use Handel-C and MPI to exploit parallel hardware architectures within and across multiple FPGAs, but the use of MPI is restricted to inter-host communication, i.e., between CPUs like a typical cluster of computers. In this thesis, MPI is used to communicate between processors and hardware engines within the same FPGA and across several of them.

One way of providing MPI functionality to embedded systems is to port a well-known implementation of MPI, such as MPICH, but that requires resources that may not be available on an *on-chip* embedded system. MPI/PRO [38] is a commercial MPI implementation for high-end embedded systems with large memories.

In the eMPI/eMPICH project [39], the authors try to port MPICH into an embedded systems domain. They first compiled a six-function basic MPICH implementation, and then added functionality to it. They obtained a set of embeddable reduced size libraries with a varying degree of functionality. However, they focused only on the top API layer of MPICH, and assumed the existence of a reduced size lower layer of MPICH (ADI layer), and they also rely on the existence of an operating system.

In the BEE2 project [40], the authors intend to use MPI to ease the task of porting super-computer applications to their architecture. They also suggest that MPI can be used not just as a programming model for the application developer, but also to provide an organizing structure for communication circuits. However, to the best of our knowledge, no results have been presented on using MPI in the BEE2 architecture.

As an example of how some MPI functionality can be implemented in hardware, Keith et. al [41] manage MPI message queues using hardware buffers. This implementation reduced the latency for moderate-length queues while adding only minimal overhead to the management of shorter queues. Although this approach is not framed into the MPSoC domain, it provides some interesting ideas that can be adapted to the on-chip level.

Williams *et al.* [42] present, perhaps the most similar approach to ours. The authors provide MPI for a single-FPGA multiprocessor, but their approach is limited to only eight processors and only one FPGA can be used, whereas our approach can use more than eight processors and multiple FPGAs.

In this thesis, a new implementation of MPI is developed that is targeted at embedded systems but tailored to a particular architecture. However, the implementation is easy to port to other platforms. TMD-MPI implements a subset of the standard API to keep it small in size, but it includes all the layers required to increase its functionality as needed. TMD-MPI does not require an operating system, has a small memory footprint (9KB) and is designed to be used across multiple FPGAs to enable massive parallelism.

Chapter 3

Hardware Testbed

This chapter describes the Class 3 machine testbed used to experiment with the programming model, the design flow to create the multiprocessor designs and the network infrastructure. First, the characteristics of hardware available on the testbed, and the software used are described. Second, an explanation of the type of processing elements inside the FPGAs, followed by a description of the network created to enable the communication between processing elements. After reading this chapter, the reader should have an overview of the target architecture.

3.1 System Overview

The Class 3 machine testbed used in this work is shown in Figure 3.1. It has five Amirix AP1100 PCI development boards [43]. Each board has a Xilinx 2VP100 FPGA [44] with 128 MB of DDR RAM, 4 MB of SRAM, support for high-speed serial communication channels, two RS232 serial ports, and an Ethernet connector. Based on a specific jumper setup, the FPGA can be configured from the on-board configuration flash memory, a compact flash card or from a JTAG interface. We use the on-board configuration flash memory as it can be accessed through the PCI bus.

Each FPGA board is connected to a 16-slot PCI backplane, which provides power to the FPGA boards and it is also used to access the configuration flash on each board. In addition to the FPGA boards, there is a Pentium 4 card plugged into the backplane. However, the Pentium processor is used only for the FPGA configuration and to provide access to I/O peripherals, such as a keyboard, a mouse, a monitor, and a hard disk drive. The Pentium card is running Linux and is also used as a terminal to print out all the standard output from the FPGAs.

The boards are interconnected by a fully-connected topology of Multi-Gigabit Transceiver links (MGTs), which are high-speed serial links [44]. Each MGT channel is configured to achieve a full-duplex bandwidth up to 2.5Gbps. The MGT links form the Tier 2 network for

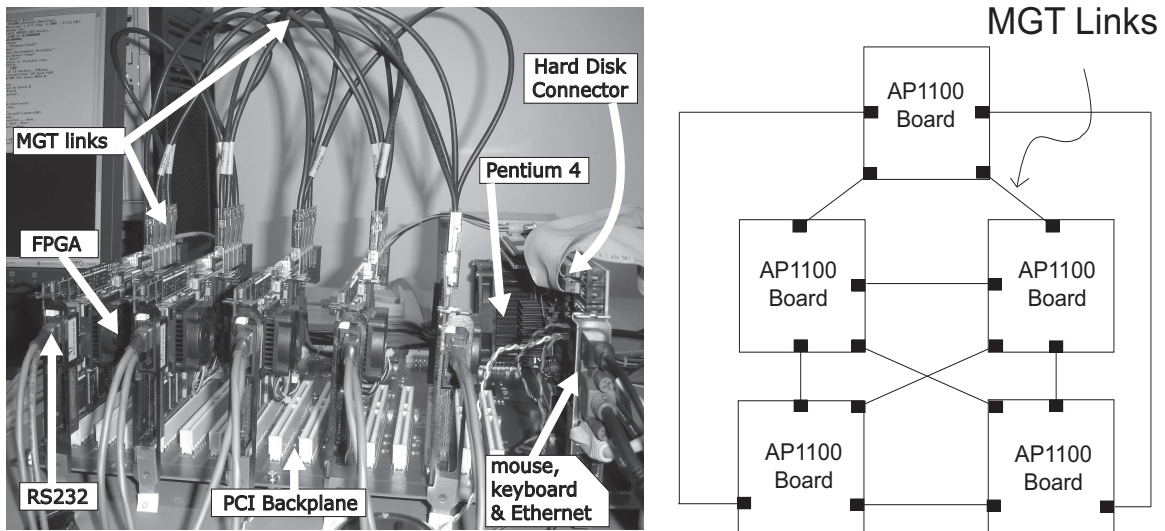


Figure 3.1: Interconnection of AP1100 boards in the Class 3 machine testbed

inter-FPGA communications. A version of Figure 3.1 that includes a detailed description of the Tier 2 interconnection scheme is shown in Appendix A. Although the infrastructure is present for a Tier 3 network to connect two or more backplanes, its use remains future work. Xilinx 2VP100 FPGAs have 20 MGT channels available, but only five boards are used because only four MGT channels have direct connectors on the board. More MGTs are available through an expansion card, which is not used in this work.

The Xilinx EDK/ISE 7.1i suite of tools is used to design and implement the multiprocessor system. For a small number of processing nodes in the system, the actual design entry can be done manually directly coding in the EDK integrated environment. It is especially easy for homogeneous multiprocessors where the description of one processor can be copied and pasted as needed, and then the particular parameters for each processor can be modified. However, for larger systems, the multiprocessor description becomes error prone due to the number of connections in the system. The use of an automated tool is recommended, such as the System Generator [45], which is a CAD tool that, based on a graph description of a multiprocessor system, generates EDK hardware and software specification files, as well as some auxiliary files for EDK. After the system is generated, particular parameters of each core can be modified, if needed.

The design of multiprocessor systems across multiple FPGAs requires considerable computing power to synthesize, place and route the designs. A 40 MHz, 9-processor project takes approximately 1.5 hours to be generated using a 3 GHz Pentium Xeon computer. The design could be generated with higher frequencies but the time it takes multiplied by the number of

boards makes it extremely time consuming for an architectural study, as in this thesis, where changes to the hardware requires frequent regenerations of the entire system. For this reason, we set up a small cluster of workstations to generate the designs and used a slower clock rate. Each FPGA board in the system has its own EDK project, and the EDK projects are distributed among the computers in the cluster. The cluster is used to generate the bitstreams only; the actual configuration of the FPGAs is done by the Pentium 4 card in the PCI backplane.

To configure the FPGA using the on-board configuration flash, a Programmable Read-Only Memory (PROM) binary file is required. We do this by using the Xilinx *promgen* utility, which generates PROM memory files from a bitstream file. The binary files generated by the cluster are transferred by FTP to the Pentium 4 card. Scripts on the Pentium 4 card were developed to automate the configuration process of each FPGA using the *Apcontrol* program [43], which is a command line interface application developed by Amirix to access the on-board configuration Flash through the PCI bus. Examples of how to download PROM files to the FPGAs are shown in Appendix D.

In such a distributed environment, it is convenient to have a centralized repository for the source code of the processors; otherwise, there would be as many source files as projects. Without a centralized scheme, a change in the code of one library would have to be replicated for all the projects increasing the risk of source-code version problems. This scheme is adequate for this particular design example in which we use a Single-Program-Multiple-Data (SPMD) programming paradigm to exploit data parallelism. A single source file is used for all the processors, but every processor has its own binary. However, other paradigms such as Multiple-Program-Multiple-Data (MPMD), in which every processor may have a different source file, can also be implemented and exploit functional and data parallelism.

3.2 The Processing Elements

At this point, the Class 3 machine testbed and how it is programmed has been broadly described. In this section, a deeper view of the computing architecture is presented. The different types of processing elements in the system are first introduced, followed by an explanation of the network components that outlines the communication infrastructure.

Three different processing units are used in this work. First, we use the Xilinx MicroBlaze soft-processor, which is an Intellectual Property (IP) processor core that is implemented using the logic primitives of the FPGA. A key benefit of using soft-processors is that they are configurable based on parameters specified at synthesis time. Also, they offer a fast time-to-market and low obsolescence curve because they are described in a Hardware Description Language (HDL), which can provide IP core updates without changing the underlying chip technology.

For the 2VP100 FPGA, up to 13 MicroBlaze processors can be instantiated in a single chip, if they use 64KB of internal memory and no other logic requires internal memory blocks.

The second processing unit is an IBM PowerPC405, which is a simple RISC processor embedded in the FPGA chip; each 2VP100 FPGA has two PowerPCs. The third type of processing units are hardware computing engines, which are specialized hardware blocks designed to efficiently compute a particular algorithm. The hardware engines are application specific, but can accelerate a particular application kernel by two or three orders of magnitude compared to a state-of-the-art superscalar processor.

In the computation model followed in this thesis, each processing unit has its own local memory. The MicroBlaze has a single dual-port RAM of 64KB for code and data. There is a single memory block, but two different buses are used to allow an independent memory access for data and code. The PowerPC has two independent memory blocks, 32KB for code and 32KB for data. The hardware engines also have available internal RAM memories to store data; the algorithm for the hardware engine is typically implemented as a state machine. The Amirix FPGA boards have external memory and it is also available to the processing elements to store larger blocks of data or a larger execution code for the processors.

The MicroBlaze and the PowerPC405 have essentially different architectures. The PowerPC405 has a 5-stage pipeline, branch prediction logic, and no floating point unit (FPU); and the MicroBlaze V.4 has a 3-stage pipeline, no branch prediction and an optional FPU. This makes the PowerPC405 faster for control sections of the code, for example in array initializations, but it has to emulate in software the FPU operations, which degrades its performance. In contrast, the MicroBlaze is not as efficient as the PowerPC405 for control operations, but if the FPU is enabled, the MicroBlaze is 20x faster in the math section of the code than the PowerPC405 at the same clock frequency. Although the PPC405 supports faster clock frequencies than the MicroBlaze, for simplicity of the design, both are running at the same frequency. Recall that the focus of this study is on the functionality of communications, the programming model and the design flow rather than to achieve peak performance, which will be addressed in future work by optimizing all the algorithms and hardware blocks, and using the maximum frequency possible.

3.3 The Network Components

As explained in Section 2.3, the processing elements communicate by means of a network. The implementation of the network requires the development of network components that enable the on-chip and off-chip communication. This network has to provide a homogeneous view, such that there is no difference, from the processing element perspective, between internal and

external communication. First, an overall description of the network is presented, followed by a detailed explanation of each network component.

Figure 3.2 shows a high-level communication diagram of the Class 3 machine testbed prototype from Section 3.1. Each FPGA in the figure has four gateway points to communicate with other FPGAs, one gateway per FPGA. The *gateway* is the interface between the Tier 1 network and the Tier 2 network. Both networks have different protocols, and as a consequence, a network *bridge* block is included in the gateway to translate network packets on-the-fly as they go off-chip and on-chip again. The off-chip point-to-point communication using the MGT links is performed by a hardware block called the Off-Chip Communication Controller (OCCC) [46]. The OCCC provides a reliable link between FPGAs, it detects transmission errors and requests a packet retransmission, if needed.

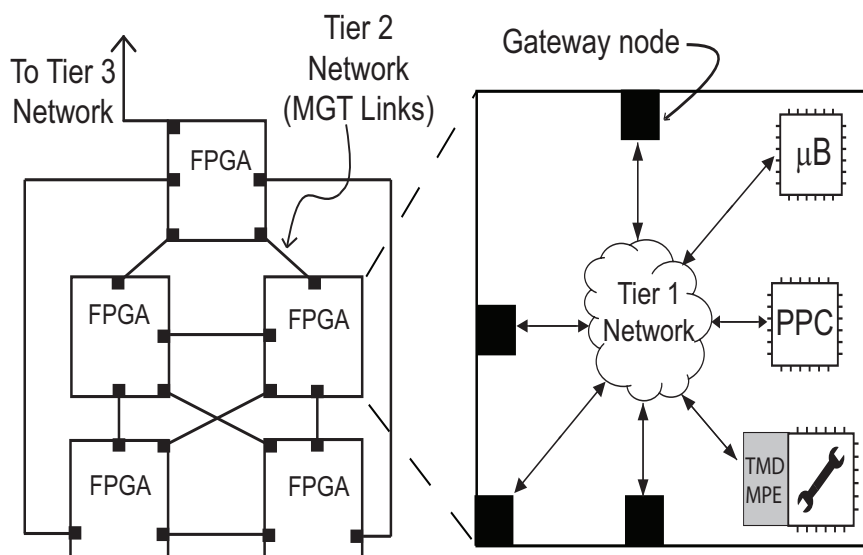


Figure 3.2: Multiprocessor System on Multiple FPGAs

In this heterogeneous multiprocessor system, we define a *node* as an element connected to a Tier 1 network. It can be a MicroBlaze, a PowerPC405, a hardware engine, or a gateway point. All the nodes are connected to a Tier 1 network interface block (NetIf), as can be seen in Figure 3.3. The NetIf has the responsibility to route all the packets in the network based on a unique identification number for each processor and a routing information table. The NetIf in a gateway point can detect when a packet is not local to the FPGA and forward it to a remote FPGA if the packet's destination field corresponds to the range of processor IDs in the remote FPGA.

As explained in Section 2.3, FIFOs are used to connect all the blocks in the Tier 1 network. The FIFO implementations used are the Xilinx FSLs [44]. Each FSL is a 32-bit wide, 16-word

deep FIFO. The depth of the FSL has an impact on performance for the Tier 1 network because deeper FSLs would increase the buffering capacity of the network and decrease the number of FSLs in the *full* condition, which does not allow the input of more data. However, in this work the same depth for all the FSLs is used; a more detailed study of this parameter remains as future work.

Each MicroBlaze soft-processor has eight FSL channels that provide a direct connection to its respective NetIf block using an FSL. In contrast, the PowerPC405 in the 2VP100 FPGA does not have an FSL interface and an additional interface block was developed to allow the PowerPC405 to be connected to the Tier 1 network. This additional block, called *dcr2fsl* [47], is a bridge between the PowerPC405 DCR bus, which is a high-speed local bus, and the FSL interface. The *dcr2fsl* block is shown in Figure 3.3.

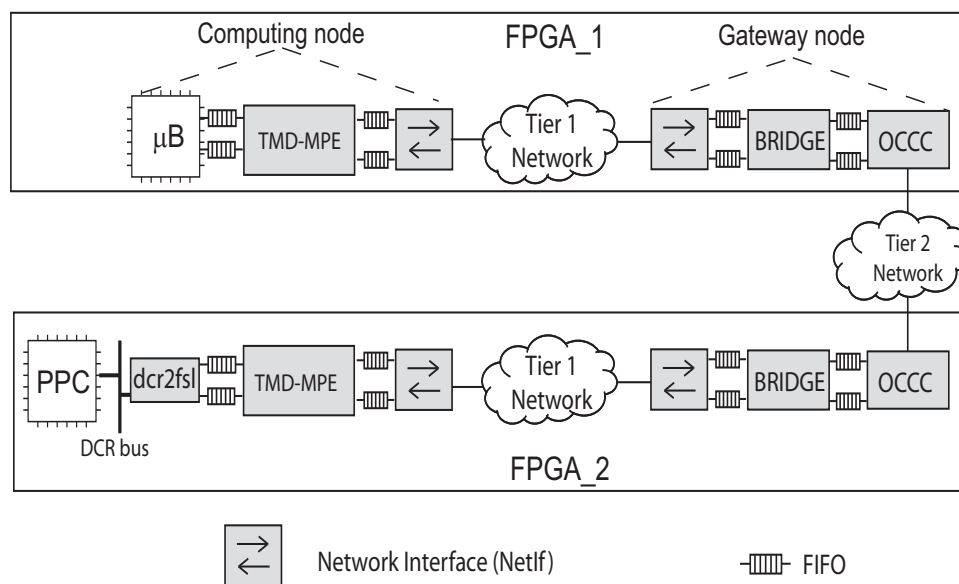


Figure 3.3: Path of a packet from one FPGA to another

Figure 3.3 shows the path that a packet follows in a typical *send* operation from one processor in one FPGA to another processor in another FPGA. First, the MicroBlaze in FPGA_1 sends the entire message to the TMD-MPE, which will split the message into smaller packets of data adding the proper headers. Each packet is sent to the NetIf, which will route the packets to the NetIf of the destination node. In this case, the destination node is the gateway node because the message is not local to the FPGA in which the MicroBlaze is located. The packets are translated into a Tier 2 network packet format by the outgoing bridge and sent to the OCCC. The OCCC will manage the off-chip communication with the receiving OCCC. Once the packets are on the other FPGA, the incoming bridge will translate the Tier 2 network packets to a Tier 1 network packet format again. The NetIf on the destination gateway will

route the packets to the PowerPC's NetIf, which will pass the received packets to the dcr2fsl block, and finally the PowerPC will read the packets through the dcr bus.

The Tier 1 network is more reliable than the Tier 2 network because all data is interchanged within the chip, as opposed to an external network, which is exposed to signal interference and transmission errors. Therefore, the Tier 1 network packet format and Tier 2 network packet format are different, and a bridge block is required to perform packet translation and enable the communication between processors across different FPGAs. The Tier 2 network packet format used by the OCCC includes the Tier 1 network packet as part of the payload. The Tier 1 network packet will be recovered as it passes through the receiving bridge. Both packet formats are shown in Figure 3.4.

For the Tier 1 network packet, the SRC and DEST fields are 8-bit values that identify the sender and the receiver respectively. These fields contain the actual MPI ranks of the tasks. In the current implementation only 256 processing units 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 packet size in words (Number of Data Words); each word is four bytes wide. Consequently the maximum packet size is $2^{16} \times 4 = 256$ KB.

The Tier 2 network packet consists of an 8-bit start-of-packet identifier (SOP), a 14-bit packet size indicator and a 10-bit packet sequence number. The second header consists of 16-bit source and destination addresses of the input channel and output channel respectively. The current OCCC supports up to three input and three output channels to multiplex the serial link and provide virtual channels around it. This is useful since most of the traffic will be concentrated around the serial links. The tail of the packet requires two more words. The *Almost EOP* word is used as an indicator that an end-of-packet sequence is to follow. The final word indicates an end-of-packet, and also acts as a place-holder where a 32-bit CRC will be inserted during transmission. The Tier 2 network packet format is described in more detail by Comis [46].

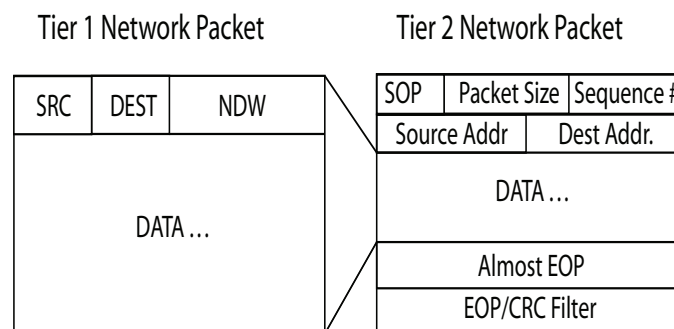


Figure 3.4: Packet Formats

3.3.1 The Network Interface

Much research has been done on developing routers and network interfaces for Networks-on-Chip. However, the existing IP either provide extra functionality that is not required for this project, or they are not in production, or are not well-documented, or are not freely available. The NetIf is a key element to achieve high performance in the system, although an optimal design of a NetIf can be a thesis by itself. At this point, the only requirement is a network device that distributes packets over the network. Eventually, once the architecture is built and an application has been ported to the TMD prototype, experiments with different network interfaces can be performed to optimize the communications. A change of such nature in the network of the TMD machine would not affect the application code. From the user perspective, at a high-level description of the algorithm, the communication is transparent for the processors due to the buffer that the implementation layers of TMD-MPI provide to the application. The layers absorb the change, not the application.

As explained in Chapter 2, FIFOs (Xilinx FSLs) can be conveniently used as communication links. However, the MicroBlaze has only eight FSL channels available, which would limit the number of nodes that can be connected directly to the MicroBlaze. The NetIf solves this problem by using packets that specify the destination node in the packet header. In this way, only one FSL channel from the MicroBlaze is used, leaving the remaining channels for other purposes, such as FPU coprocessors, or any other expansion to the MicroBlaze processor. From a hardware engine perspective, the NetIf alleviates the hardware computing engine from the burden of managing many FSL channels reducing the complexity of the engine's state machine. The engine has to control only one FSL channel and write the proper destination field in the packet header. This allows the engine's designer to focus on the computing part and not on processing packets.

The NetIf acts as a distributor when sending packets and as a concentrator when receiving packets from other NetIfs. We consider two approaches to implement a NetIf based on how the packets are distributed and how the packets are concentrated. One approach is broadcast-based and the other is more selective. In either case, since the architecture of a Class 3 machines is application-specific, the network interface is extremely simple because all the links within an FPGA are point-to-point, extracted from the application's communication pattern. Therefore, there is no need to reroute packets to communicate with nodes within the same FPGA.

Figure 3.5 shows a simplified block diagram of a broadcast-based NetIf. This approach is very simple because on the transmission side there is practically no logic involved, except an OR gate for the *read* signal coming from other NetIfs. This *read* signal will de-queue a value from the processing unit FIFO when a destination NetIf successfully matches a packet to the destination computing node's rank. The output of the processing unit's FIFO is fanned-

out and connected directly to the input of other NetIfs in the network within the same FPGA. Therefore, when a packet is sent to a particular destination node, the packet will be received by all the NetIfs with a direct connection to the source NetIf. However, only the NetIf that belongs to the destination node will let the packet move across the NetIf and reach the processing unit. The other NetIfs will simply discard the packet. The *Incomming Message Logic* block in Figure 3.6 acts as a packet filter by performing a match between the destination header in the packet and the processing unit id number (MPI rank).

The *Channel Celection Logic* block determines which channel to use in case there are more than one packet with the same destination coming from different sources. Currently, the mechanism is a linear priority decoder that grants preference to channel 0 over channel 1, and channel 1 over channel 2, and so forth. Once the channel has been selected, it is registered and held by the *Channel Select Register* until the entire packet has passed-through the NetIf. The *counter* keeps track of the packet size and it gets decreased as the data values are queued into the processing unit's reception FIFO. With the source channel selected, the *read* signal can then be sent back to the source NetIf to de-queue the data values from the transmission FIFO of the source processing unit. This scheme is used for its simplicity, but it may lead to inefficient network traffic management because a short message in a low priority channel would have to wait for a large message in a higher priority channel. Alternative scheduling algorithms can be further investigated for future versions of the NetIf.

For a packet to go off-chip, the NetIf is connected to a gateway node. The gateway's NetIf will have a range of ranks that should match the destination field in the packet header, if the packet is meant to go off-chip. In that way, the packet will automatically be routed on-chip and off-chip without the need of routing tables or complex routing mechanisms. However, the major drawback of this approach is the energy consumption. Since all the links will be active every time there is a message, the switching of logic levels in the wires will increase the power required to transfer data.

The selective approach is more energy-wise because it sends the packets only through the channel that leads to the destination node. The penalty for this is a relatively more complex design that uses more logic and FPGA routing resources per NetIf block. Figure 3.6 shows a simplified block diagram of the selective NetIf. In this approach, the transmission side has an address decoder, which determines what output channel to use based on the *Destination Rank Table*, which is a list of rank-ranges per channel. If the destination rank of a particular packet falls into a certain range, the associated channel will be used to transfer the packet. The range of ranks is specified by a low rank number (LRN) and a high rank number (HRN). For example, if output channel 0 of node *A* is connected to a computing node with rank x , then the range for channel 0 of node *A* will be $LRN_0=x$ and $HRN_0=x$. But if channel 1 of node

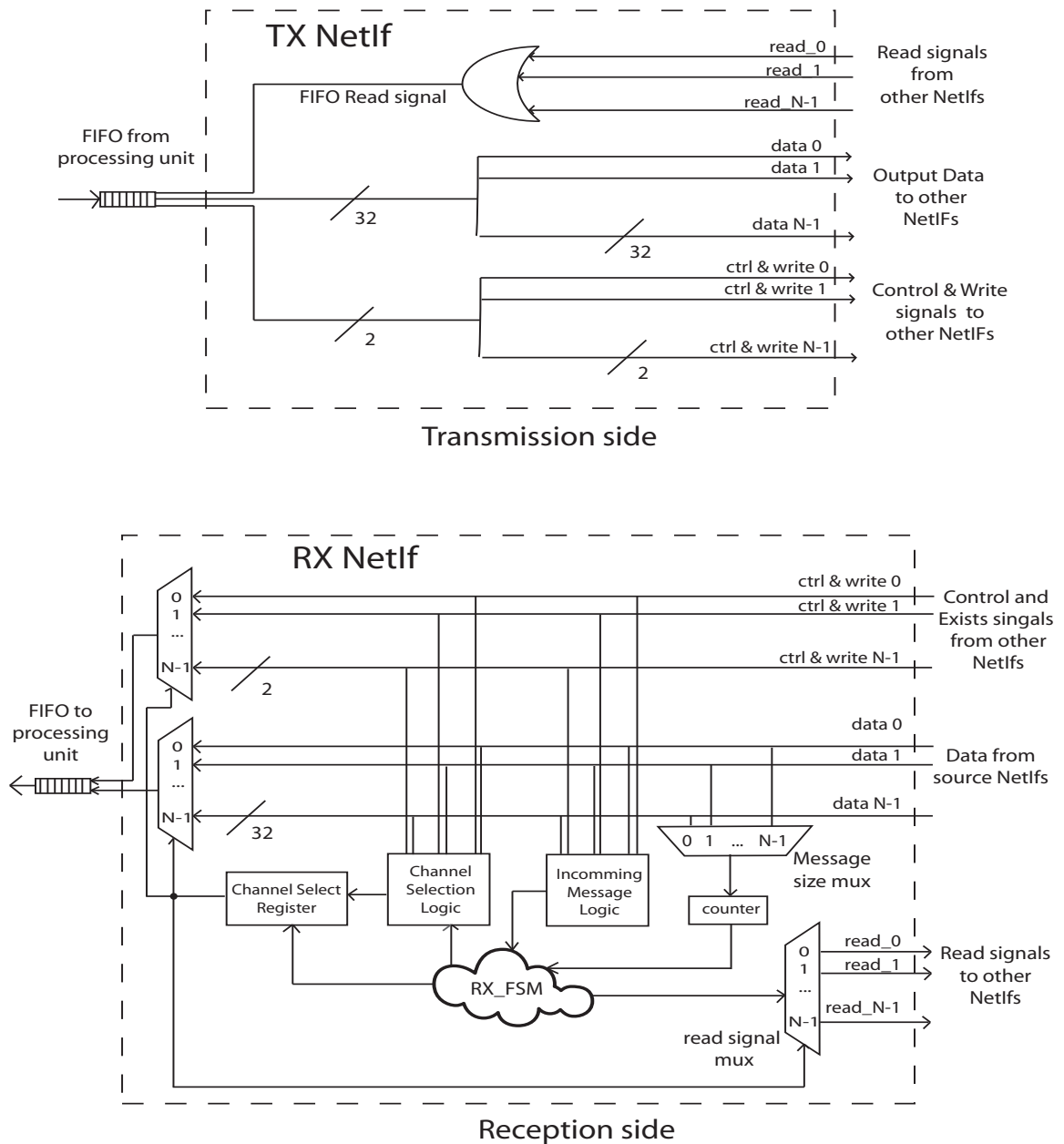


Figure 3.5: Broadcast-based Network Interface

A is connected to a gateway node that leads to another FPGA containing nodes with ranks between 10 and 20, the rank range for channel 1 of node A will be $LRN_1=10$ and $HRN_1=20$. Once the NetIf has resolved what channel to use to transfer a packet, the channel number gets registered in the *Destination Register* and holds until the entire packet has been transferred. The registered channel number is also used to select the control signals for the appropriate output FIFO connected to other NetIfs.

The reception of the selective approach does not perform a match between the packet header

and the node's rank. If a packet is present, it is because it is meant to be there. Nevertheless, there is a *priority decoder* to resolve concurrent packets from different sources. Once the source channel has been determined, the channel number gets registered by the *Source Register* and holds until the entire packet has passed through the NetIf.

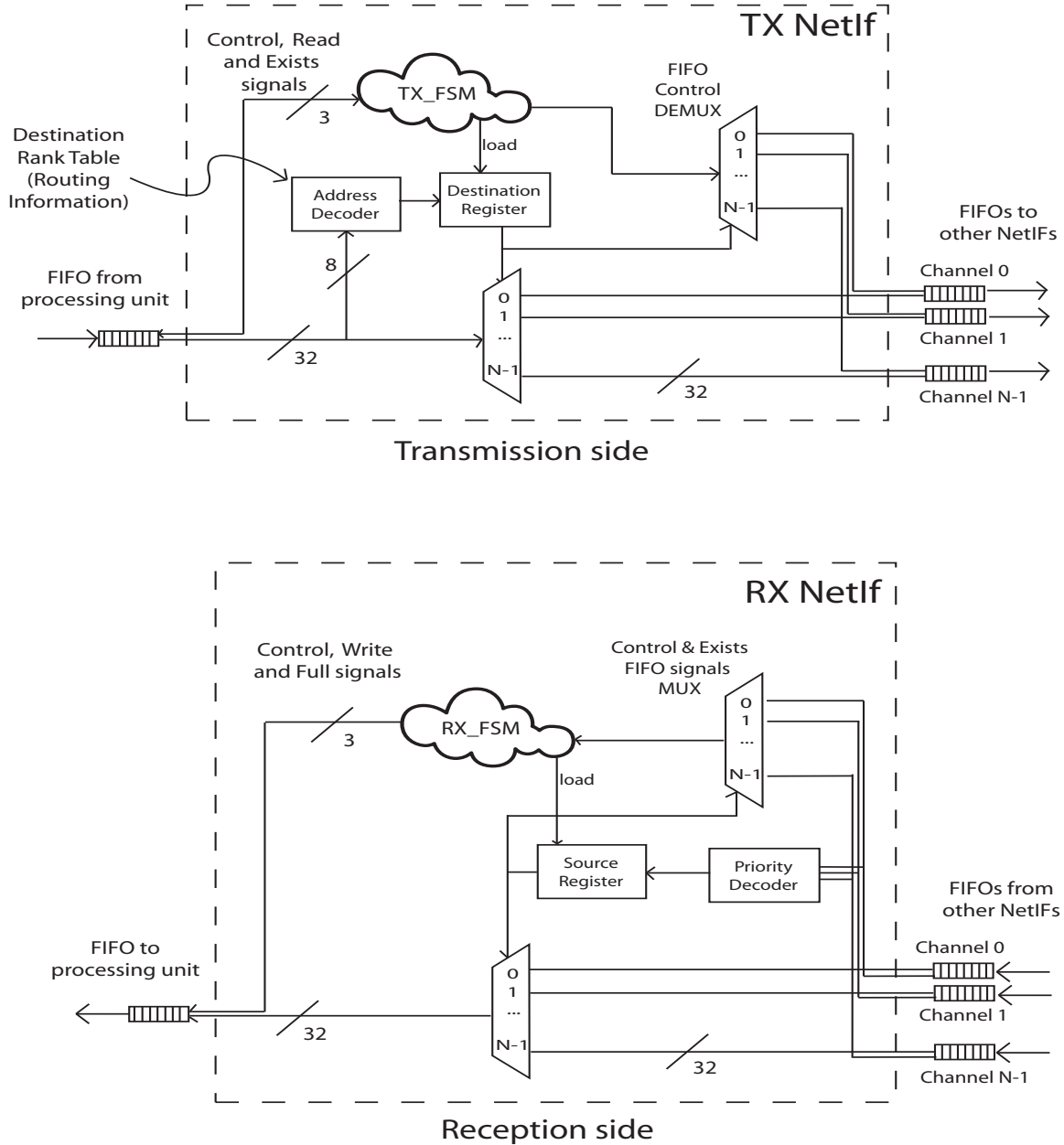


Figure 3.6: Selective Network Interface

In both approaches, the id of the processing unit is the same as the MPI rank in the application. If we compare this networking scheme to a typical, general local area network, this convention simplifies considerably the network stack and network design because there is a

direct mapping between the rank of a task (MPI process) and the physical id number of a node (the equivalent of a MAC address in a typical Ethernet network card). Otherwise, there would be a mapping from a hostname (or MPI Rank) to an IP network address number (for routing purposes on the Tier 2 network), and then from the IP network address number to the MAC address of the network interface (to locate the node within the Tier 1 network). Again, since the architecture is application specific, the generality that the mapping provides can be obviated to reduce the protocol stack overhead. The only extra overhead for sending packets across different network tiers is that the packet must pass through the network *bridge* block, which is explained next.

3.3.2 The Network Bridge

As explained previously, the different packet formats between Tier 1 and Tier 2 networks creates the need for a block that translates the packets as they go from one FPGA to another. This block is implemented in hardware and is called Network Bridge (bridge). Figure 3.7 shows a block diagram of the bridge. The design is extremely simple because the packet formats are also simple. The bridge has two sub-blocks, one for Tier 1-to-Tier 2 network packet translation (internal-to-external), and another from Tier 2-to-Tier 1 network packet translation (external-to-internal). The difference in the blocks is the format expected at the input and the resulting format at the output.

The external-to-internal translation is straightforward because the objective is to remove the extra header and tail control-words required by Tier 2 network. The bridge uses the FIFO's functionality to *read* (de-queue), or to *write* (queue) a value from it. The FSM2 state machine has to de-queue the extra headers from the slave-external FIFO without queuing them into the master-internal FIFO. The Tier 1 network headers will be queued into the master-internal FIFO as soon as they appear on the slave-external FIFO. The entire data payload will also be forwarded from the slave-external FIFO to the master-internal FIFO. Finally, the tails of the Tier 2 network packet will also be discarded when present in the slave-external FIFO.

The internal-to-external FIFO is slightly more complicated than the external-to-internal translation. Instead of removing the headers and tails of the Tier 2 network, they are added to the Tier 1 network packet. When a packet is present in the slave-internal FIFO, the state machine will initialize a counter with the packet size. This counter will indicate when the entire packet has been received by the bridge, and there is no more data words on their way to bridge, still going across Tier 1 network. After initializing the counter, the state machine will add the two headers required for the Tier 2 network, one after the other. Then the data from the slave-internal FIFO will be transferred to the master-external FIFO while the counter

Chapter 4

Message Passing Implementation

The previous chapter described a prototype of a Class 3 machine architecture. The rest of this thesis relies on that architecture to build a message-passing layer that will be used for communication among processing elements in the system. In this chapter, the programming model is presented in Section 4.1. Then Section 4.2 explains how the proposed message-passing model makes possible a hardware-software co-design flow. Section 4.3 presents the details of the software implementation of the message-passing library. Section 4.4 illustrates how the message-passing functionality can be implemented in hardware. Finally, Section 4.5 is a brief discussion on how different computing node configurations can be formed based on the combination of computing elements and the type of message-passing implementation they use.

4.1 The Programming Model

A major challenge in Class 3 machines is the design flow and the programming model. From the user perspective, a programming model should provide an efficient method for implementing applications while abstracting underlying hardware complexities. It is perhaps as important as the design of the architecture itself for the performance of a high-performance computing machine. Programming models are defined by the architecture of a machine, as well as the nature of the applications intended to be executed on the machine (e.g. Internet programming, client/server, graphical user interfaces, parallel programming, and artificial intelligence systems). A very specific programming model would result in less portable code and the user would have to learn new functions for a slightly different application. On the other hand, a very generic programming model would result in an inefficient program. Despite the implicit application-specific architecture of Class 3 machines, the programming model has to remain generic within the nature of the application, but optimized to a particular architecture, to be

able to program other similar applications.

The architecture of the TMD machine does not utilize shared memory or shared bus structures; rather, each computing engine contains its own local memory. The requirement for the programming model is the capability to deal with tasks being executed concurrently assuming each process has its own set of the variables and data structures to work with. The MPI programming model fits this requirement, and has also proven to be a successful paradigm for scientific computing applications running on supercomputers and clusters of workstations. From this, the purpose of this research is to investigate whether the MPI paradigm can be practically applied to high-performance Class 3 machine systems, such as the TMD machine. Using a standardized message-passing interface such as MPI as the basis for the TMD programming model provides a familiar environment for application developers reducing the learning curve. Moreover, using MPI enables portability of parallel applications to different platforms by standardizing the syntax and semantics of the programming interface.

The programming model presented here is based on a network of processing elements implemented on FPGA fabrics exchanging messages via the MPI interface. Applications designed for the TMD are initially described as a set of software processes executing on an embedded processor network. Each process emulates the behavior of a computing engine and uses MPI function calls for exchanging data with other processes. These function calls use the TMD communication infrastructure described in Chapter 3 to transmit and receive data packets. An entire system can be prototyped on the TMD in this manner. The abstraction of soft-processors executing MPI function calls is used as an intermediate step in the overall process of mapping an application to a set of computing engines implemented on the TMD. The overall process is described next in the Design Flow section, along with how the MPI functionality is involved in a hardware-software co-design flow.

4.2 Design Flow

Prior to describing the TMD design flow, it is instructive to differentiate the challenges between mapping a single computational kernel to an FPGA-based hardware module, and mapping an entire parallel application to an architecture containing multiple FPGAs. A number of design tools exist for directly translating single-process C code into synthesizable hardware, such as HardwareC [48], Handel-C [49] or C2Verilog [50]. These tools are best suited for identifying computationally-intensive application kernels and implementing them in hardware modules, usually confined to a single FPGA. However, software applications contain other programming tasks that can pose challenges to automated software-to-HDL design flows, such as I/O functions, complex control structures, and pointer-based memory accesses. Consequently, not

every application can be translated directly into hardware.

Additional complexities arise when developing a method for automatically implementing parallel applications on the TMD. One such issue is the network synthesis from a high-level description based on the communication pattern of an application. Communication networks for interconnecting FPGAs in the TMD are defined, but networks for interconnecting multiple computing engines within individual FPGAs must be implicitly inferred from the software implementation. Intrinsic data and functional parallelism existing within the application should also be automatically detected and taken advantage of. Higher-level parallel application features such as dynamic load balancing and task allocation are also important constructs that may not be realizable in hardware. In Class 2 machines, many of these challenges are mitigated due to the presence of the host processors in the system.

The design flow developed for the TMD does not attempt to address these issues automatically. Rather, a manual flow is used to transform parallel software applications into networks of hardware computing engines. This approach allows us to study and evaluate design decisions for the TMD architecture, identify potential problems in the flow, and debug the initial architecture. An automated design process would follow as a result of these activities. The current flow resembles that presented by Youssef *et. al.* [32], but in this thesis, the model is extended to many soft-processors distributed across multiple FPGAs and the model is applied to heterogeneous processing elements.

Figure 4.1 illustrates the TMD design flow that was first introduced in Patel *et. al.* [12]. 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 generic 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. The granularity of each process is much finer than that of software processes normally used in parallel applications, since each process is meant to emulate the behaviour of a computing engine. Inter-process communication is achieved using a full featured MPI message passing library, such as MPICH, allowing the application to be developed and tested on a workstation. This approach has the advantage of allowing the programmer access to standard tools for developing, profiling, and debugging parallel applications. To maintain compliance with the next step, the programmer must refine the parallel program and use only a subset of the MPI functionality as described in Section 4.3. Validation of the parallel application can be done at this point.

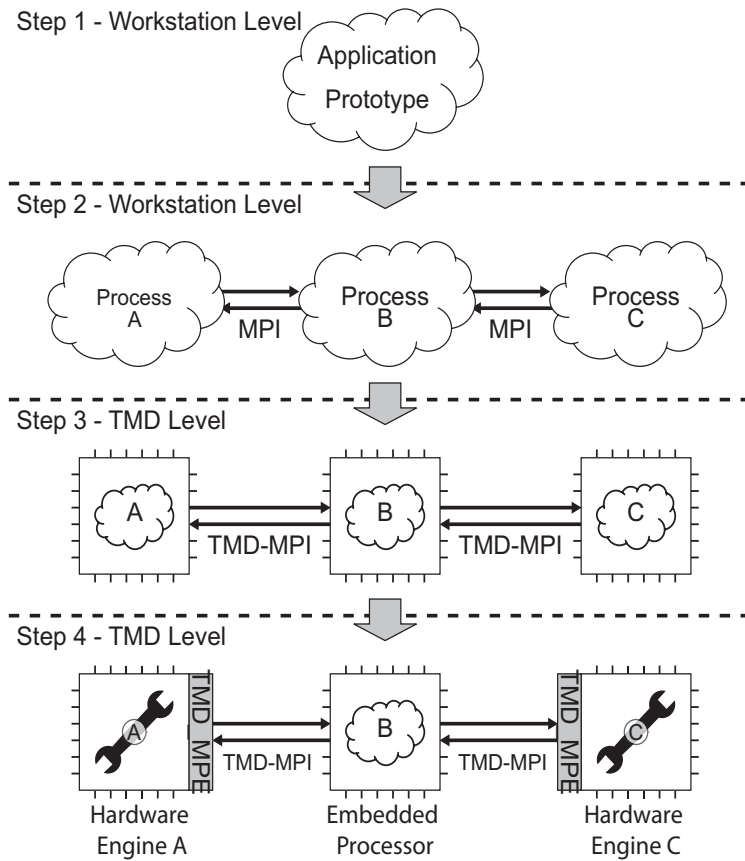


Figure 4.1: Design Flow with MPI

A full featured MPI implementation has a considerable number of lines of code devoted to error checking. These extra lines of code can be obviated in TMD-MPI and reduce its size requirement and overhead at runtime.

Step 3 takes the collection of software processes developed in Step 2 and implements them on the TMD using soft-processors. Each processor contains a library of MPI-compliant message-passing routines (TMD-MPI) designed to transmit messages using the TMD communication infrastructure. The portability of MPI allows the software processes to be recompiled and executed on the soft-processors. At this stage, execution on-chip of the entire application is possible; allowing the interaction between emulated computing engines to be tested and the implementation correctness validated.

The final step of the programming flow substitutes algorithms executing on embedded processors with hardware-based computing engines. This step is only required for performance-critical computing engines and can be omitted for less intensive computing tasks. Additionally, control-intensive tasks that are difficult to implement in hardware can remain as software executing on embedded processors. In some designs, the processors are located in the same chip as the hardware engines. This creates a very tightly coupled design with a very low communication latency. Larger designs may require that the hardware engine be located in a different FPGA, increasing the latency. Translating the computationally-intensive processes into hardware engines is done manually in the current flow. 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. Once a computing engine has been designed, an additional message-passing engine (TMD-MPE, Section 4.4) 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 computing engine. The TMD-MPE can also be used by the processors as a communication coprocessor, which would relieve the processor from handling communications and enable a more efficient data exchange.

4.3 TMD-MPI: Software Implementation

This section describes the software implementation of the simplified, light-weight MPI library called ‘TMD-MPI’ created for the TMD architecture. Although TMD-MPI is currently written for the Xilinx MicroBlaze soft-processor [44] and the IBM PowerPC embedded processor, it can easily be ported to other processors by modifying the lower layers.

TMD-MPI implementation is software only, with the assumption of an underlying network infrastructure that handles the lower levels of networking such as packet routing and flow

control. This infrastructure was explained in Chapter 3.

Tasks that are typically performed by a message-passing implementation are communication protocol processing, handling pending-message queues, packetizing and depacketizing large messages. In the software-only version of TMD-MPI these tasks are performed by the soft-processor itself, which implies an extra overhead for the processor. However, this method allows the message-passing functionalities to be developed and tested easily. Once the details of the TMD-MPI implementation have been finalized, most of the functionality can be provided by a more efficient hardware core. For processors, it means a reduction in the overhead of the TMD-MPI implementation; for hardware computing engines, it means the possibility of interacting with processors. The software-only version can be used where performance is not critical or where adding more hardware is not desired.

As mentioned earlier, TMD-MPI currently implements only a subset of functionality specified by the MPI standard. Although this set of operations is sufficient for many applications, additional features can be added as the need arises. The following list gives a description of the functions implemented to date.

Utility Functions :

MPI_Init Initializes TMD-MPI environment (application initialization)

MPI_Finalize Terminates the TMD-MPI environment (application termination)

MPI_Comm_rank Returns the rank of the calling process within a communicator.

MPI_Comm_size Determines the number of processes associated with a communicator.

In soft-MPI there is only one process per soft-processor; therefore the number of processes is equal to the number of soft-processors in the group.

MPI_Wtime Returns the number of seconds elapsed since the application began execution. This function is included for profiling purposes.

Point-to-Point Functions :

MPI_Send Sends a message to a destination process.

MPI_Recv Receives a message from a source process.

Collective Functions :

MPI_Barrier Blocks the execution of a process until all processes have reached this routine. Used as a synchronization mechanism.

MPI_Bcast Broadcasts a message from the root process to all other processes in the group.

MPI_Reduce Reduces values on all processes to a single value in the root process.

MPI_Gather Gathers values from a group of processes together.

There are several outstanding issues in TMD-MPI that are currently being investigated. In a MPI-based application executing on a Class 1 machine, the *mpirun* command is invoked by a user to launch the application. This command uses the underlying operating system to spawn multiple processes on different hosts, and is also responsible for assigning unique ranks to each process. Since there is no underlying operating system in the current implementation, we statically assigned each process to a soft-processor and determine MPI process ranks at compile-time. A boot-loading mechanism is being developed to provide this functionality. Other minor issues include: lack of support for data types larger than 32 bits, support for only synchronous (blocking) implementations of `MPI_Send` and `MPI_Recv` primitives, support for only maximum and addition reduction operations in `MPI_Reduce`, and reduced error-checking of function parameters to reduce the overhead of the library, although error-checking is not crucial as the application should have been validated in a full-featured MPI implementation in a workstation.

It has been said before that other efforts to use MPI for embedded systems require an operating system. The TMD-MPI approach does not require one, but that does not preclude processors from having one. There is no interference between TMD-MPI and an OS because of the way the Microblaze accesses the FSLs is similar to the way it accesses a register from the register file. The only requirement would be that the operating system does not use the same FSL as the MPI implementation is using to connect to the network; otherwise, there will be conflicts with the device driver of the operating system.

4.3.1 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 4.2 shows the layered scheme implemented in TMD-MPI.

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

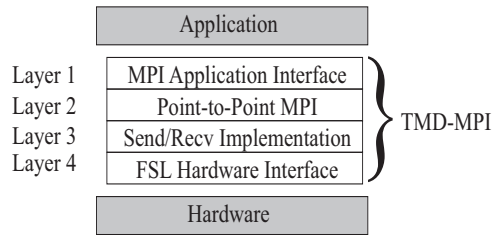


Figure 4.2: Implementation Layers

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.

4.3.2 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 [21] in MPI, respectively. Figure 4.3 shows a typical example of the difference between the protocols. 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 Kilobytes or even Megabytes 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 packet to the producer. Once the producer receives the clear-to-send packet, 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.

Based on the Tier 1 network packet format described in Section 3.3, TMD-MPI defines three types of packets as shown in Figure 4.4. Each one of these packets correspond to the arrows in Figure 4.3. All the packets contain a packet header that is used to route the packet over the network; the difference between them is the payload content. The *envelope packet*

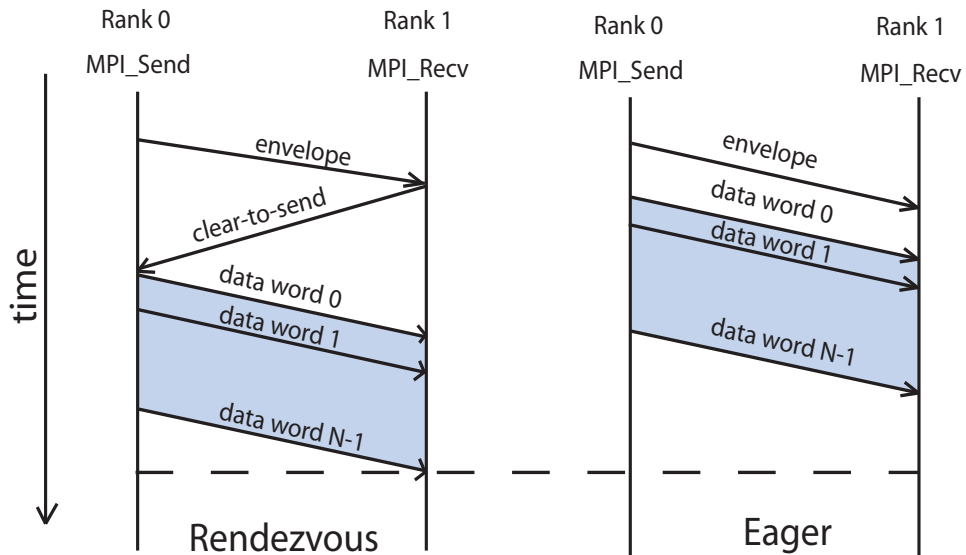


Figure 4.3: Difference between Rendezvous and Eager protocols.

contains the message *tag* as the only data value in the payload. Recall that the *tag* is a user definable message identification number. The tag, in addition to the source and the destination ranks in the header, provide enough information to describe a message and allow the receiving side to match the message with the expected message parameters. However, the *envelope packet* does not contain actual data of the message. The *clear-to-send packet* is used in the rendezvous protocol to confirm to the sender that the receiver is ready to receive the message. The *clear-to-send packet* contains only one data word, which has the value of `0xFFFFFFFF`. It is a special *tag* value; therefore, it should not be used as a normal value in a TMD-MPI application. The third type of packet is the *data packet* that contains the actual data of the message.

In large multiprocessor systems, messages out of order or unexpected are common. This becomes a problem because unexpected messages would have to be buffered, and if there is a large number of unexpected small messages, or a small number of large messages, a buffer overflows can occur. In a system with scarce memory, buffering messages would be inadequate because it would limit the scalability of the system, and it may also lead to wasted memory in the case when unexpected messages are rare. We decided to use the rendezvous protocol in TMD-MPI because of the smaller memory requirements. However, a mix of both protocols would be desirable because an eager protocol would be better for short messages and reserving the rendezvous protocol for large messages. This is an improvement for future versions of TMD-MPI.

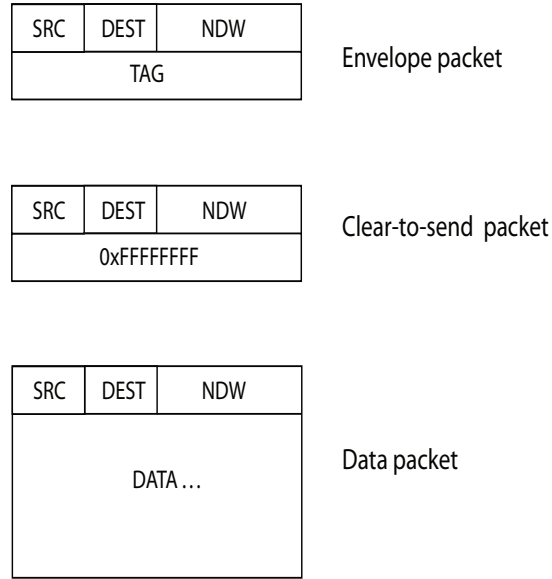


Figure 4.4: Type of packets used by TMD-MPI

4.3.3 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.

Figure 4.5 shows the memory that is used for the message queue. The low address word of the memory stores the source and destination ranks, and number of data words, which is the packet size (NDW). The high address word stores the message tag, which is a message identifier defined by the user. Each word is 32 bits wide, and the number of words in the memory is $2 \times \text{maximum_number_of_unexpected_messages}$. Since the rendezvous protocol is used for the TMD-MPI, the worst case scenario will be when there is one pending message per node because any node will never send two messages without an acknowledge. Therefore, the *maximum_number_of_unexpected_messages* should be equal to the number of nodes in the sys-

tem. A larger memory will only be wasted. This shows that the rendezvous protocol sacrifices communication performance for memory efficiency.

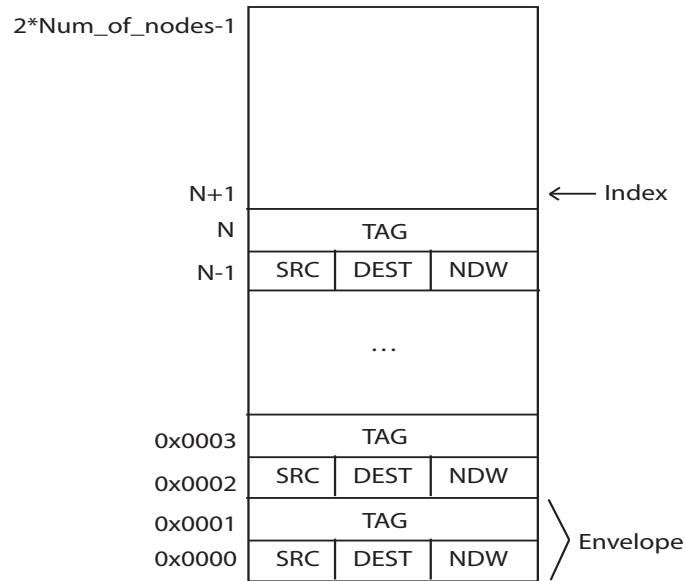


Figure 4.5: Queue of envelopes from unexpected messages.

4.3.4 Packetizing and Depacketizing

The OCCC supports a variable packet size between eight 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 Tier 1 network was chosen to be the same as the packet size for the Tier 2 network. TMD-MPI performs a *packetizing* process that divides large messages into packets no shorter than two words and no longer than 512 words. Similarly, the inverse process of combining 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 sequence number in the *depacketizing* process.

Figure 4.6 shows the *packetizing* process of a large message into packets of the maximum size allowable. It is highly probable that the last packet is smaller than the rest of the packets because the last packet contains the remainder. This requires the network components to handle variable size packets, which they do. Each packet contains a header with the source and destination ranks of the tasks involved in the communication for routing purposes. Only the

first packet (*envelop packet*) contains the message tag, the rest of the packets does not include it.

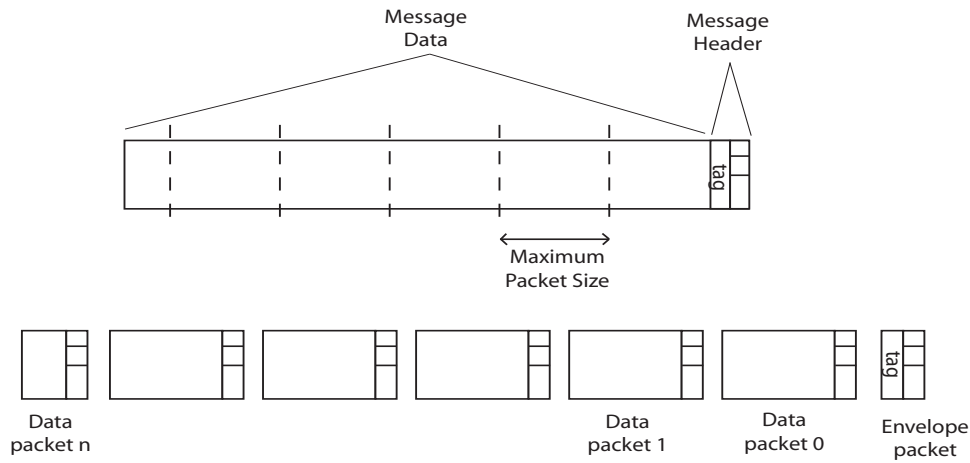


Figure 4.6: Packetizing process of a large message into smaller packets.

4.4 TMD-MPE: Hardware Implementation

The goal of transferring the TMD-MPI functionality into a hardware block (TMD-MPE) is to improve the overall performance of the communication system by reducing the latency, increasing the message throughput, and relieving the processor from handling the message-passing protocol. TMD-MPE provides support for handling unexpected messages, handles the communication protocol and divides large messages into smaller size packets. Currently, TMD-MPE only provides the equivalent to the `MPI_Send` and `MPI_Recv` MPI functions to a hardware engine. The collective functions are not yet implemented.

As shown in Figure 3.3, the TMD-MPE is connected between the computing element (processor or hardware engine) and the network interface (NetIf). The TMD-MPE, will receive the message parameters and data from the computing element. These parameters are 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 operation). From this point, the TMD-MPE will handle the communications through the network with the destination node.

The Message-passing engine is meant to be used for hardware engines, but can also be connected to processors. However, three factors limit the advantages of using the TMD-MPE

with processors. The first factor is the memory access time when the processor is reading or writing data to be sent or received. This delay depends on the memory hierarchy; the memory being accessed could be in the processor's local bus, or a common on-chip bus, or external memory. Each memory has different access times that impact the data transmission throughput. In a hardware engine, memory can be accessed directly without the overhead of arbitration in a bus. Nevertheless, there will be a delay if the engine accesses off-chip memory. The second factor is the FSL interface access time. In the case of the MicroBlaze, the FSL access time is three cycles, for constant values, i.e. no memory access. These three cycles account for the copy of the value from the FSL interface to a register when reading, and from a register to the FSL interface when writing. For the PowerPC405 there is an extra delay when accessing the FSL interface because the information has to go to the DCR bus first, and from there pass through the dcr2fsl-bridge. The third factor that limits the throughput is the implicit sequential execution of instructions in a normal processor. A hardware engine can execute operations in parallel considerably faster. Therefore, the message-passing engine will be most beneficial to hardware engines, whereas a processor will limit the message-passing engine.

Figure 4.7 shows a simplified block diagram of the message-passing engine. In this version of the engine, the hardware resources, such as registers, multiplexers, memories, and logic are controlled by a single state machine, and shared for the send and receive processes. As a consequence, the engine is half-duplex, which allows sending or receiving, but not both at the same time. This is clearly a limitation, since the network architecture described in Chapter 3 allows full-duplex communication. A future version of the engine has to be full-duplex to take advantage of the hardware engine's capacity to send, receive, and compute at the same time. However, the MicroBlaze and the PowerPC405 cannot take advantage of a full-duplex message-passing engine, because of the single stream of execution.

The TMD-MPE has an internal memory to store the envelopes of unexpected messages. The message envelopes are stored in memory in the same manner as TMD-MPI does (see Figure 4.5).

The TMD-MPE also has a *Message Match Logic* that determines whether a message is expected or not. If the message is not expected then the message envelope will be stored into the message queue, and no clear-to-send packet is sent back to the source. If the message is expected, then a clear-to-send packet is sent to the source node; when the message arrives it will be redirected to the host. The *Message Match Logic* also determines if there are envelopes from a previous unexpected message in the envelope memory queue.

The *Header Building Logic* creates the packet headers that will be sent to the network based on the packet size, the source, and the destination. This information is obtained from the host computing element or from a previously received envelope from another node. The *Reception*

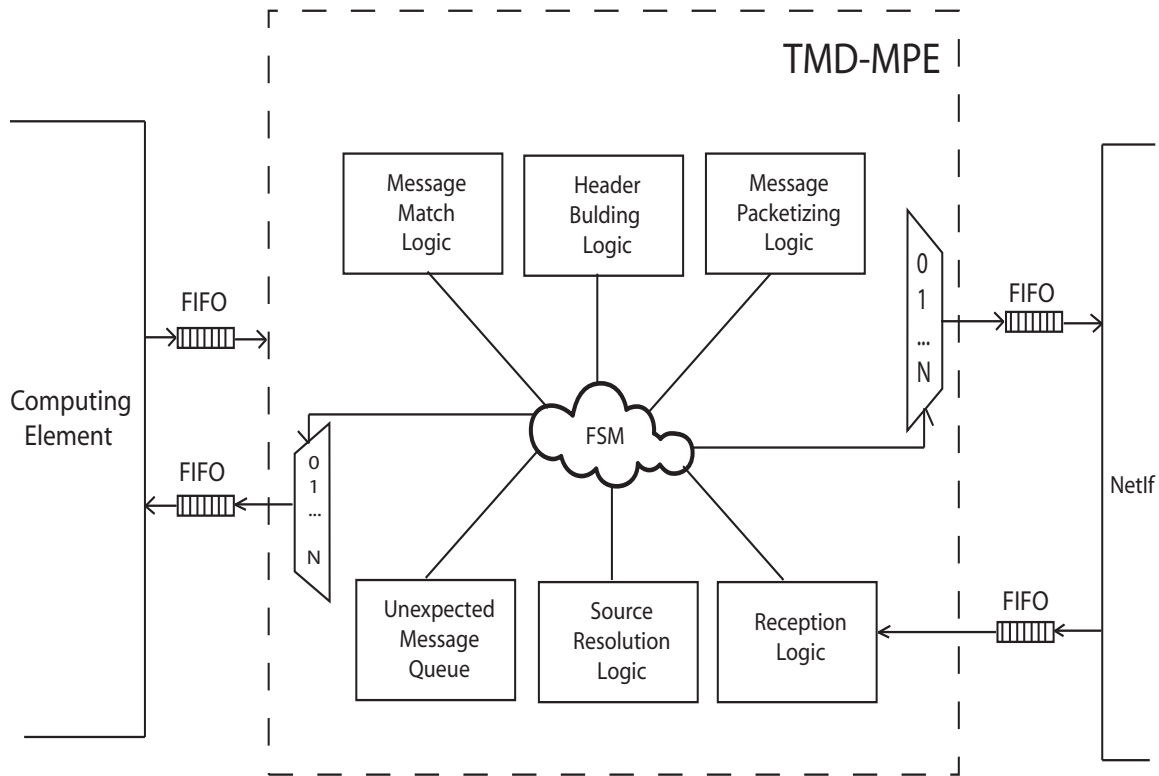


Figure 4.7: TMD-MPE block diagram

Logic registers the source, destination, packet size, and message tag from incoming packets. The reception logic also provides information to other logic in the TMD-MPE based on the received values. The *Message Packetizing Logic* determines if a message is larger than the maximum packet size. If that is the case, the packet size field in the packet header is modified accordingly. The large message is divided into equal-sized packets as long as there is data. The last packet has the remainder of the data, which might be less than the maximum packet size. This is also sent to the *Header Building Logic* to correctly create the packet header.

The *Source Resolution Logic* determines to which node a particular packet has to be sent. It can be seen as the process to determine the sender of a letter, so the message can be replied to. A message source can be determined by a direct specification from the host computing element, from an envelope of a previous unexpected message stored in the memory queue, or from a newly received envelope from the network.

Figure 4.8 shows the communication protocol between the computing element (hardware engine or processor) and the TMD-MPE. The first value sent from the computing element to the TMD-MPE is the message-passing operation code *TMD-MPE Opcode*. A *send* operation (MPE_Send) has an opcode of 0x00000001, and a *receive* operation has an opcode of 0x00000002. In this version of the TMD-MPE, only these two functions are implemented.

The second value to be sent is the message size, i.e., the number of 32-bit words, not bytes. The *Local Rank* field is the rank number of the local computing element. The *Remote Rank* field is the rank of the other node involved in the communication. It can be either a sender or a receiver. If the local computing unit is receiving, the *Remote Rank* is the senders rank. If the local computing unit is sending, the *Remote Rank* is the receivers rank. The *message TAG* field contains the MPI message tag that identifies the message.

After this, the TMD-MPE has all the information to deal with the communications protocol. If the local computing unit is receiving, the TMD-MPE will send back to the computing unit the received message in a special *data packet*, which is as large as the message itself, i.e., this large packet does not have the restriction of a typical *data packet* as explained in Section 4.3.4.

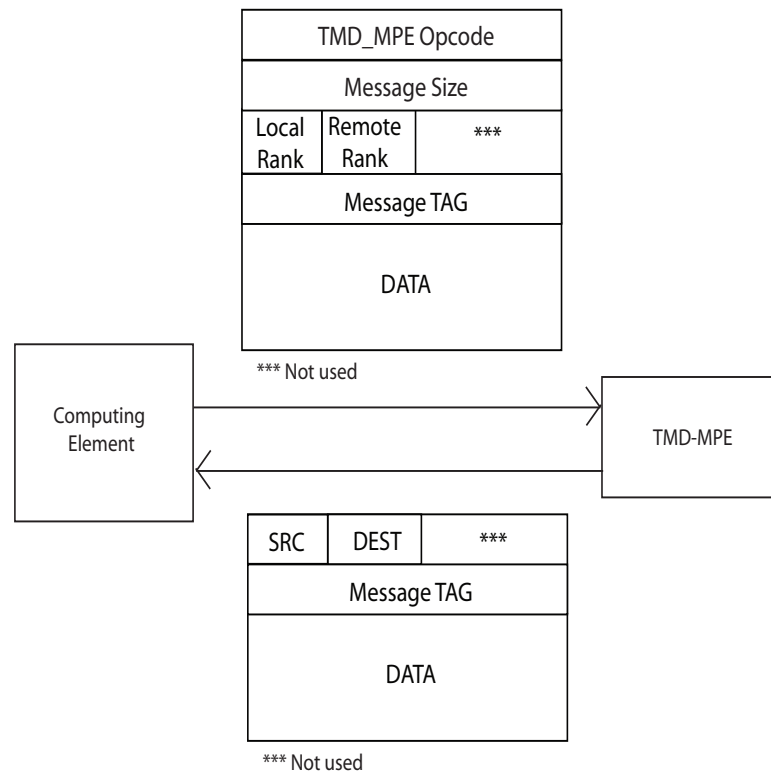


Figure 4.8: TMD-MPE communication protocol with the associated computing unit.

4.5 Computing Node Configurations

Based on the the type of computing element and whether the TMD-MPE is used, different configurations can be formed. Figure 4.9 shows the possible combinations for a computing node. As previously explained, the PowerPC405 requires the dcr2fsl adapter to use FSLs. The MicroBlaze and PowerPCs that have the TMD-MPE, require a lightweight version of

TMD-MPI because much of the functionality is now in hardware; the processors that do not have the TMD-MPE, require a full TMD-MPI version. The hardware engines will always require the TMD-MPE to be able to connect to the network. The version of TMD-MPI used by the processors is defined at compile time by declaring a constant in the command line of the compiler.

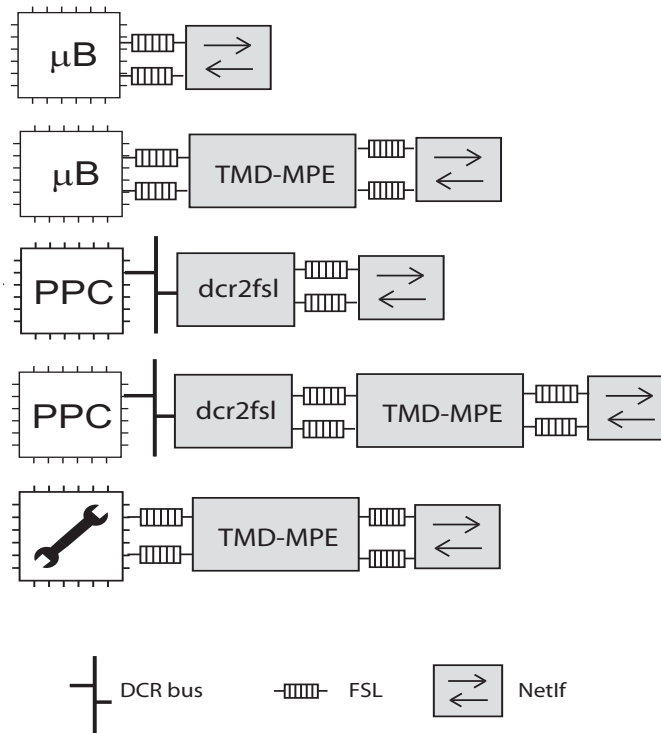


Figure 4.9: Different node configurations based on the use, or not, of TMD-MPE

Chapter 5

Testing the System

In this chapter, two different sets of tests are performed to evaluate the design flow, the programming model, the network infrastructure, and the TMD-MPI and TMD-MPE implementations. The first set of tests is meant to measure basic communication characteristics, such as latency, bandwidth, and synchronization performance. The objective is to ensure the correct functionality of the lower levels of the system: the network, and the TMD-MPI library. It is also important to have a benchmark to provide results for further improvements.

The second set of tests is focused on the higher level of abstraction by mapping a scientific application to hardware. The goal is to illustrate the use of the design flow and programming model. The aim is to demonstrate the functionality and viability of the work presented in this thesis and not to achieve maximum performance. The parallel scientific algorithm used was chosen for its simplicity, but it is not the most efficient one. Finding the most efficient algorithm and building an optimal hardware engine for a particular scientific application is beyond the scope of this thesis. Recall that the objective of this thesis is to build a functional platform that can be used for further research on MPSoC across multiple FPGAs.

For the first set of tests the broadcast version of the network interface is used, whereas for the second set of tests the selective version of the network interface is used. Since they are two independent sets of tests and the focus is on testing functionality, there is no conflict for using different network interfaces.

5.1 Testing TMD-MPI Communication Characteristics

There are existing benchmarks [51–54] to measure the performance of an MPI implementation. However, they are too complex for embedded systems, they use functions not yet implemented in TMD-MPI, or they require a file system I/O, or a command line interface, or batch execution of processes. This assumes the existence of an operating system, which is not present in

the systems presented in this work. ParkBench [52] 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, a set of C-language benchmarks is developed for embedded processors called *TMD-MPIbench*. This same testbench was executed on the testbed system shown in Figure 5.1, on a network of Pentium-III Linux workstations (P3-NOW) running at 1 GHz using a 100 Mbit/s Ethernet network, and also on a 3GHz Pentium 4 Linux Cluster using a Gigabit Ethernet Network (P4-Cluster). TMD-MPIBench proves the portability that MPI provides to parallel C programs by allowing the execution of the same code in a Class 1 machine and a Class 3 machine. The P3-NOW and P4-Cluster are using MPICH versions 1.2, and the testbed system in Figure 5.1 is using TMD-MPI. These tests are meant to demonstrate TMD-MPI functionality and to characterize the performance of the initial implementation of our platform.

It must be mentioned that the tests on the P3-NOW and the P4-Cluster were executed in a non-exclusive, not fully-controlled environment. Therefore, the sudden activity of other users, the operating system overhead, cache effects, among other factors can influence the results of the tests for the P3-NOW and P4-Cluster. However, the objective of the tests is to obtain an initial reference point to compare the performance of our multi-FPGA platform against a more typical computing system, and to show the results that TMD-MPIBench produce in those systems.

In TMD-MPIBench, the functions *MPI_Send* and *MPI_Recv* are used to send and receive messages. These functions use what is referred in the MPI standard as *standard communication mode*. The *standard communication mode*, can use a buffered communication mode or a synchronous communication mode; the decision depends on the implementation. The P3-NOW and the P4-Cluster use MPICH, which uses an asynchronous communication mode for small messages and a synchronous communication mode for large messages. In TMD-MPI, only the synchronous communication mode is implemented. This means, that the comparison may not be fair for TMD-MPI since MPICH could be using an asynchronous protocol, which may perform better than TMD-MPI. But recall that the tests described in this chapter are meant to prove functionality of the platform, verification of TMD-MPI itself, the portability of the application code, and to make sure we have a reasonable performance compared to other systems.

5.1.1 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. This is done by sending round trip messages between two processors. The variables are the message size, the type of memory in which data is stored

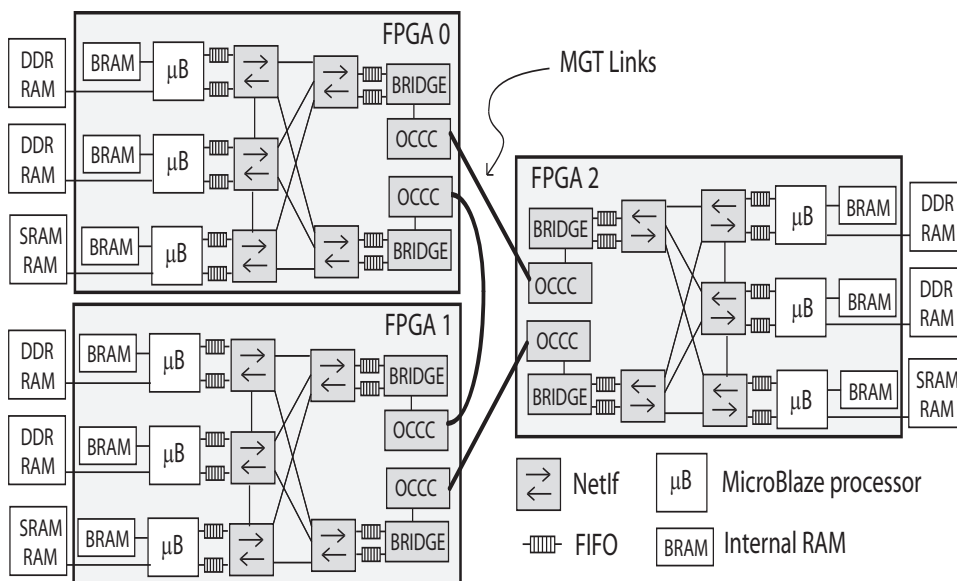


Figure 5.1: Experimental testbed hardware system

(internal BRAM or external DDR memory) and the scope of the communication (on-chip or off-chip). The results are shown in Figure 5.2.

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. Note that, in this case, latency affects the message throughput because round trip times are being measured, 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 5.2, not all the results from the P4-Cluster are shown as they would compress the other plots.

Similarly, from Figure 5.2, it can be seen 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, the highest link-bandwidth is achieved, but still less than the Ethernet P3-NOW and the P4-Cluster. By doubling the frequency of the multiple MicroBlaze system, a higher link-bandwidth than the P3-NOW is expected. 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 $17\mu\text{s}$, $22\mu\text{s}$, $75\mu\text{s}$ and $92\mu\text{s}$, respectively. These measurements are taken using the `MPI_Wtime` function and subtracting its timing overhead, which is $96\mu\text{s}$ for the MicroBlaze, $3\mu\text{s}$ for the P3-NOW and $2\mu\text{s}$ for the P4-Cluster.

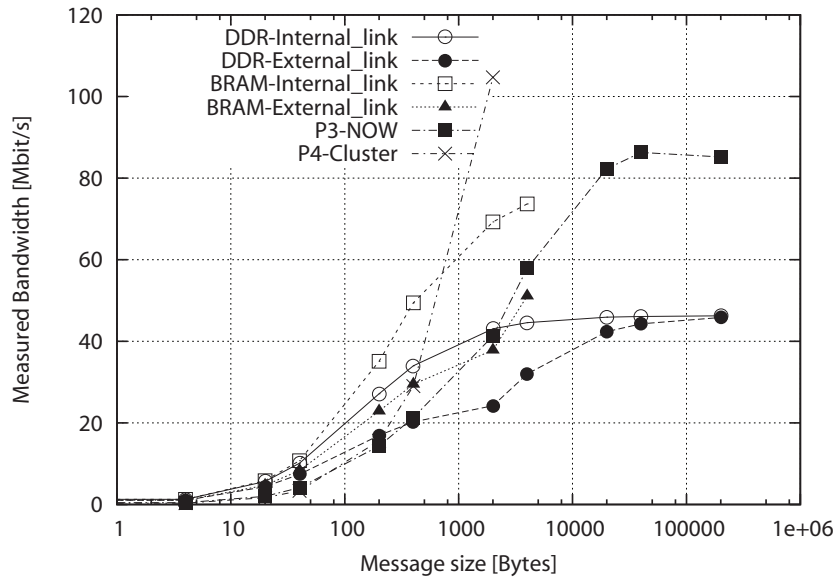


Figure 5.2: Measured link bandwidth under no-traffic conditions

5.1.2 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. In the P3-NOW and P4-Cluster the worst case link-bandwidth remained almost the same. In contrast, in our network the worst case link-bandwidth dropped by almost half of the bandwidth previously reported under no-load conditions. 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-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 are using a buffered communication mode, which takes advantage of the absence of synchronization, and the Ethernet switch has a more advanced scheduling algorithm.

5.1.3 Synchronization Performance

In parallel programs, barrier synchronization is a common operation and it is important to guarantee correctness. For example, the BlueGene Supercomputer [2] 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 5.3. 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, as shown in Figure 5.3, 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 and the operating system may cause sudden changes.

5.2 The Heat Equation Application

To test the functionality of the system, an example application is presented in this section. This application is the heat equation, which is a partial differential equation that describes the temperature change over time, given a specific region, initial temperature distribution and boundary conditions. The thermal distribution is determined by the Laplace equation $\nabla(x, y) = 0$, and

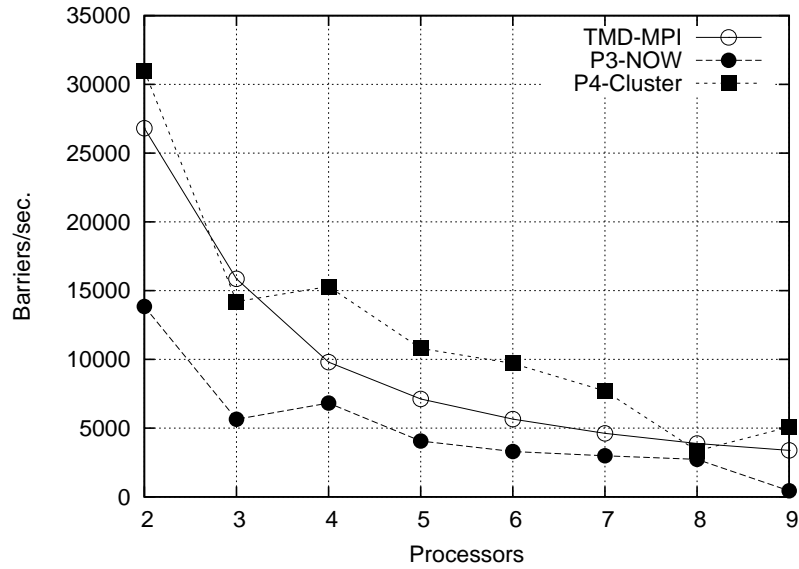


Figure 5.3: Barrier Synchronization Overhead

can be solved by the Jacobi iterations method [55], which is a numerical method to solve a system of linear equations. It is not the goal of this thesis to develop the best possible algorithm to solve the heat equation. This method was chosen for its simplicity and because it requires communication among the processors to perform the computation. This example application is meant to prove that the multiprocessor system in our Class 3 machine works by doing a meaningful computation.

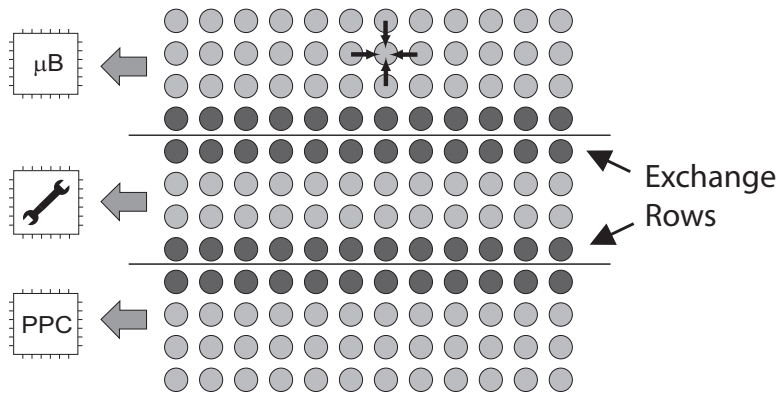


Figure 5.4: Data decomposition and point-to-point communication in Jacobi algorithm

The basic Jacobi algorithm is to solve iteratively Equation 5.1, where u is the matrix with the temperatures in a given iteration step and v is the matrix with the temperatures for the next iteration. In every iteration, the values of u have to be updated with the values of v . The convergence condition is computed as the square root of the sum of the square of differences between the old values and the new values as shown in Equation 5.2.

$$v_{i,j} = \frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}}{4} \quad (5.1)$$

$$\sqrt{\sum_{i,j} (u_{i,j} - v_{i,j})^2} < \epsilon \quad (5.2)$$

The basic parallel algorithm for Jacobi is to split the matrix into smaller, equal-size matrices as shown in Figure 5.4. Each processor would have to exchange the border rows with its immediate neighbors to compute the limiting rows in each section. The steps in the algorithm are:

Step 1. Send to every node the section of data to process

Step 2. Exchange rows with neighbors

Step 3. Perform computation

Step 4. Compute the convergence data

Step 5. Reduce the convergence data and send it to the master

Step 6. Receive from master convergence signal, go to step 2 if data has not converged

To program the Jacobi iterations method, the programming flow described in Section 4.2 is followed. The changes in the parallel C code from the workstation to our testbed are minimal. Based on the type of processor used, conditional compilation macros are coded to include the proper header files for the MicroBlaze and the PowerPC405 processors generated by the EDK; the actual MPI code remains untouched. The solution of the heat equation is validated by comparing the final results of the testbed system with the results of the sequential version of the code running on a Pentium processor.

5.2.1 Performance Results

In this section, two experiments are conducted to test the design flow, the programming model and the network infrastructure running the Jacobi iterations method in different heterogeneous multiprocessor systems. Again, these tests provide an initial reference to compare the performance with further improvements to TMD-MPI, TMD-MPE and the network components.

5.2.2 Performance Test

In this first test, the problem size is fixed to a square region of 60x60 elements, and the impact on performance of heterogeneous cores is measured. Following the design flow and TMD-MPI, a number of multiprocessor configurations can be created and programmed easily. For

this objective, only one FPGA with nine processing units is used. There are a considerable number of possible combinations for this experiment based on the configuration of the computing nodes, as explained in Section 4.5. Only a subset of the possible combinations is used, which encompasses the most representative configurations. Table 5.1 summarizes the different configurations for this experiment. The $\#\mu B$, $\#PPC$ and $\#Jacobi\ Engines$ columns are the number of MicroBlaze processors, the number of PowerPC405 processors, and the number of Jacobi hardware engines, respectively. The $FPU\ for\ \mu B$ column indicates whether the floating point unit is enabled in the MicroBlaze. Finally, the $Experiment\ ID$ column is a label to identify the experiment.

Table 5.1: Experiment configurations

$\#\mu B$	$\#PPC$	$\#Jacobi\ Engines$	FPU for μB	Experiment ID
9	0	0	no	<i>9uB</i>
9	0	0	yes	<i>9uB_FPU</i>
7	2	0	yes	<i>7uB_FPU_2PPC</i>
1	0	8	yes	<i>1uB_FPU_8HW</i>
4	2	3	yes	<i>4uB_FPU_2PPC_3HW</i>

Figure 5.5 shows the main-loop execution time of each configuration in Table 5.1. The initialization of the square region is not considered because the time spent in that part of the code is not significant. The y-axis has a log scale because the difference in execution times vary considerably between configurations. In this experiment, a MicroBlaze with FPU unit enabled, achieves 60x faster execution times than a MicroBlaze without FPU, and 20x faster execution times than a PowerPC405 because of the overhead of software emulation of the floating point operations in the PowerPC405. The impact of this can be seen in configuration *9uB*, which is slow compared to the other FPU-enabled configurations, as expected. For configuration *4uB_FPU_2PPC_3HW*, three MicroBlazes are substituted by Jacobi hardware engines. The Jacobi hardware engines use the TMD-MPE to connect them with the MicroBlazes and PowerPCs. The execution time decreases gradually for the first seven nodes. However, only the first four nodes are MicroBlaze processors, the next three are the Jacobi hardware engines, but there is no improvement in performance because the MicroBlazes are considerably slower than the Jacobi hardware engines forcing them into an idle state while waiting for the stop message coming from the first MicroBlaze. The performance degrades even further when nodes eight and nine are used. These nodes are PowerPC processors without FPU support, which causes a drastic increase in the execution time. A similar situation happens with configuration *7uB_FPU_2PPC* because the PowerPC is slower than the MicroBlaze with FPU. For clarity, the plot for configuration *7uB_FPU_2PPC* is not included in Figure 5.5 since it is identical

to the plot of configuration *4uB_FPU_2PPC_3HW*. Finally, in configuration *1uB_FPU_8HW*, eight MicroBlazes are substituted by hardware engines, but the improvement in performance is marginal compared to the *9uB_FPU* configuration, because the first Microblaze is still part of the computation, which slows down the Jacobi hardware engines. In this case, a master-slave approach, in which the MicroBlaze is not part of the computation and only collects the convergence data and sends the stop message would produce faster execution times for configuration *1uB_FPU_8HW*.

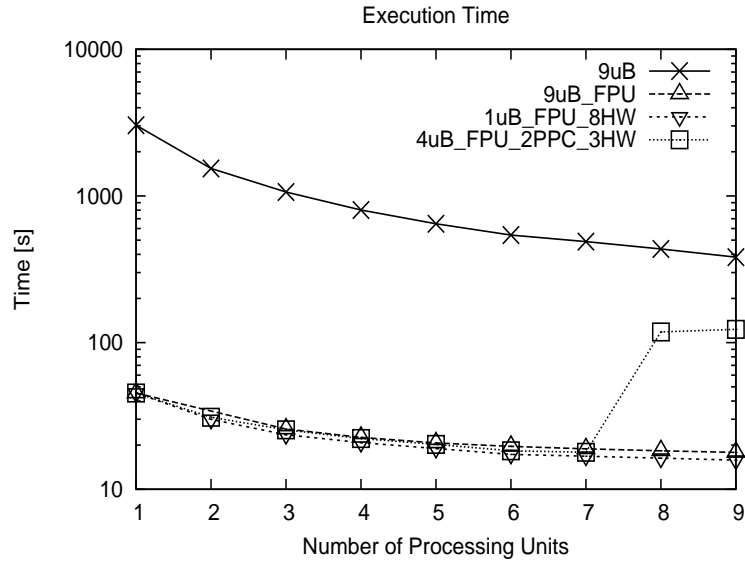


Figure 5.5: Main loop execution time of different Multiprocessor configurations

5.2.3 Scalability Experiment

In this experiment, the scalability of the system is tested by using the five boards available. Although each FPGA board can have a different MPSoC configuration, the same configuration is used in all the FPGAs for simplicity. The changes from one FPGA to another are the network routing table and a constant defined at compile time for each processor to define the MPI rank. Seven MicroBlazes and two PowerPC405 are implemented in each board. None of the processors are using TMD-MPE and the MicroBlaze FPU unit is disabled. There is a total of 45 processors, limited by the amount of internal RAM in the FPGA. In a fixed-size problem, the region to be computed has to be the same size regardless of the number of processors. Therefore, a region with enough rows for each one of the 45 processors and still big enough to fit in the memory of a single processor is required. A narrow region of 240 rows by 16 columns meets that requirement; a larger region will not fit into the memory of a single processor. For one processor, there will be a computation of 240 rows and there will not be communication.

For 45 processors, each node will compute only five rows on average and exchange two rows of 16 data values per iteration.

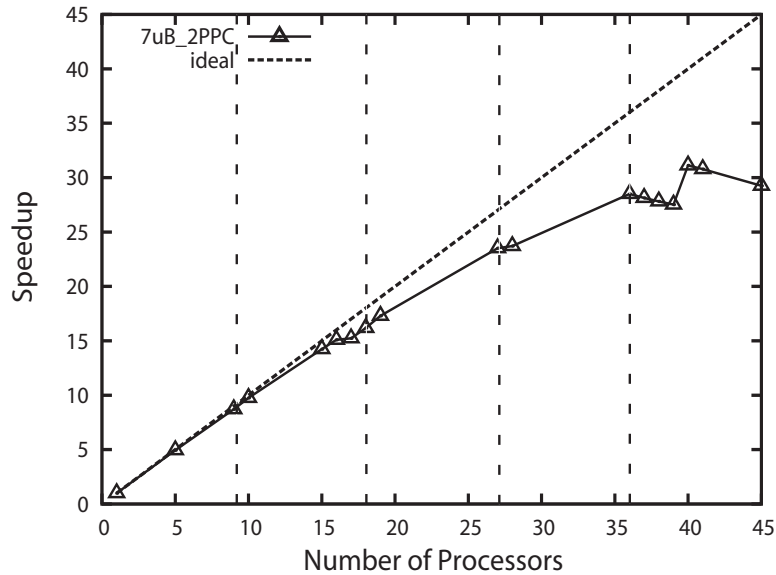


Figure 5.6: Speedup of Jacobi iterations in the 45-node multiprocessor system

Figure 5.6 shows the speedup of the execution of the Jacobi algorithm for 45 processors. The irregularities of the plot are due to the nonlinearities of the network interface's scheduling algorithm and the heterogeneity of the computing cores and communication links. The effect of using multiple boards connected by the MGT links produces a slight change in the slope of the plot. The vertical dashed lines show when there is an addition of a new board. For this particular architecture-algorithm pair, there is a sustained speedup up to 35 processors. But there is an inflection point around 40 processors representing a peak performance-point after which adding more processors will only slow down the execution of the application.

Chapter 6

Conclusions

This thesis describes the architecture of a multi-FPGA-based computing platform, its programming model and its communication infrastructure. This computing platform is called TMD and is intended to be used for high-performance computing applications that can take advantage of application-specific hardware and that exhibit a high computation-to-communication ratio.

To this end, a low-overhead, low-latency network infrastructure was developed to allow computing tasks to interchange data and control messages efficiently. The network infrastructure has a modular design based on components that can be instantiated as needed to expand the number of nodes in the system providing a scalable architecture.

On top of this network infrastructure, a programming model was developed to provide the user with tools to create applications. This programming model is a lightweight subset MPI standard implementation called TMD-MPI for executing parallel C/C++ programs on a multi-processor System-on-Chip across multiple FPGAs. TMD-MPI does not require an operating system and is approximately 10KB in size, which is sufficiently small to fit in internal RAM in an FPGA. TMD-MPI facilitated the porting of an application from a Linux cluster to the multi-FPGA system by just including the proper header files for the MicroBlaze and the PowerPC405 processors; the rest of the code remained the same. This portability and reusability allow faster development cycles and shortens the learning-curve to program a reconfigurable computer. TMD-MPI was also used as a model to develop a message-passing engine that provides basic TMD-MPI functionality in hardware. This allowed the communication between computing hardware engines and embedded processors in the FPGA.

The network infrastructure and the programming model make possible a flexible hardware-software co-design flow, which permitted the prototyping of an entire application in the TMD prototype using embedded processors. By following the design flow explained in this thesis, the substitution of some processors by computing hardware engines was transparent to the rest of the processors in the system. This level of abstraction and isolation from the actual imple-

mentation provided great flexibility to prototype the hardware engines and test them in a real multiprocessor architecture on-chip and not in simulation. With the design flow methodology used, the most time consuming task of adding processors to the system was not the design entry but the synthesis, place and route of the system.

To test the network infrastructure, a set of benchmarks was executed and compared to Linux clusters. Our experiments showed 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 were achieved with a clock frequency of only 40MHz.

To analyze the design flow and further test the programming model, an entire application was developed and implemented on the TMD prototype. This application is the Heat equation solved using the Jacobi iterations method in a 45 embedded processor system across five FPGAs. A sustained speedup of the application up to 35 processors shows that the system is scalable and limited by the amount of on-chip memory and on-chip resources, but not by the communications.

Finally, we can conclude that the network infrastructure, the programming model, the design flow and their integration with the implementation tools have made it easy to design, program and implement multi-FPGA systems. Additionally, there are various concrete opportunities for further research to improve the current status of the ideas presented in this thesis.

6.1 Future Work

With this working Class 3 machine testbed the focus should be on optimization and improvement of the architecture and the design flow. Some future work and suggestions are presented in this section.

The TMD platform should be used to optimize the TMD-MPI algorithms. For example, more efficient collective operation algorithms are required, such as tree-based reduction instead of the basic linear algorithm that is currently implemented. Also, TMD-MPI uses only the rendezvous protocol, which saves memory and avoids buffer overflow problems for large messages, but is inefficient for short messages. To alleviate this limitation, a hybrid scheme should be implemented including the eager protocol or buffered scheme for short messages, and the rendezvous protocol for large messages.

Currently, the TMD-MPE only performs the basic *send* and *receive* operations, and collective operations are still coordinated by the computing element that the TMD-MPE is connected

to. In future versions of the TMD-MPE, it should be the message-passing engine the coordinator of the collective operation and not the computing element.

Also, a DMA version of the TMD-MPE would eliminate the overhead for the processor of copying the data to the FSL. In the DMA version, an address where data is located should be provided to the TMD-MPE by the computing element. The TMD-MPE then is responsible for accessing the data. The Xilinx dual-port RAM can facilitate the implementation of the DMA version of the TMD-MPE.

The current network protocol only has 8-bit address fields to specify the source and destination nodes. Therefore, there are only 256 possible addressable nodes, which is clearly a limitation to the scalability of the system. The first improvement suggested is to at least double the width of the address to support 65536 nodes. The address width can be increased even further, but the wider the address field is the more latency will be experienced. It is a tradeoff between generality and application-specific needs.

Improvements of the hardware blocks that form the Tier 1 network are also scheduled for future work, such as, better scheduling and routing algorithms in the NetIf block. One caveat of our system is that to include a new node in the multiprocessor system, all the boards in the entire system have to be synthesized, placed and routed again, which is time-consuming, because the NetIf's routing information changes and some links are added to the system. To solve this problem, it is recommended to investigate incremental synthesis, partial reconfiguration or software configuration at runtime of the routing table using an embedded processor, while minimizing the overhead in the protocol or additional hardware.

Further investigation can be done to evaluate the impact on resource usage and performance benefits of having the Tier 1 network packet format and the Tier 2 network packet format merged into a unique network packet format. This would eliminate the need of translating packets.

In terms of the design flow, we are working on CAD tools to develop multiprocessor systems for multi-FPGA platforms to help with the layout of the architecture, the mapping of code into processors, and to automatically generate the design files for the FPGA design tools. Finally, we will concentrate our efforts on the implementation of molecular dynamics simulation on this prototype as it is a demanding application with a low communication-to-computation ratio, in which a Class 3 machine, such as the one presented in this thesis can provide a significant speedup.

Appendices

Appendix A

Tier 2 Network for the 45-Processor System

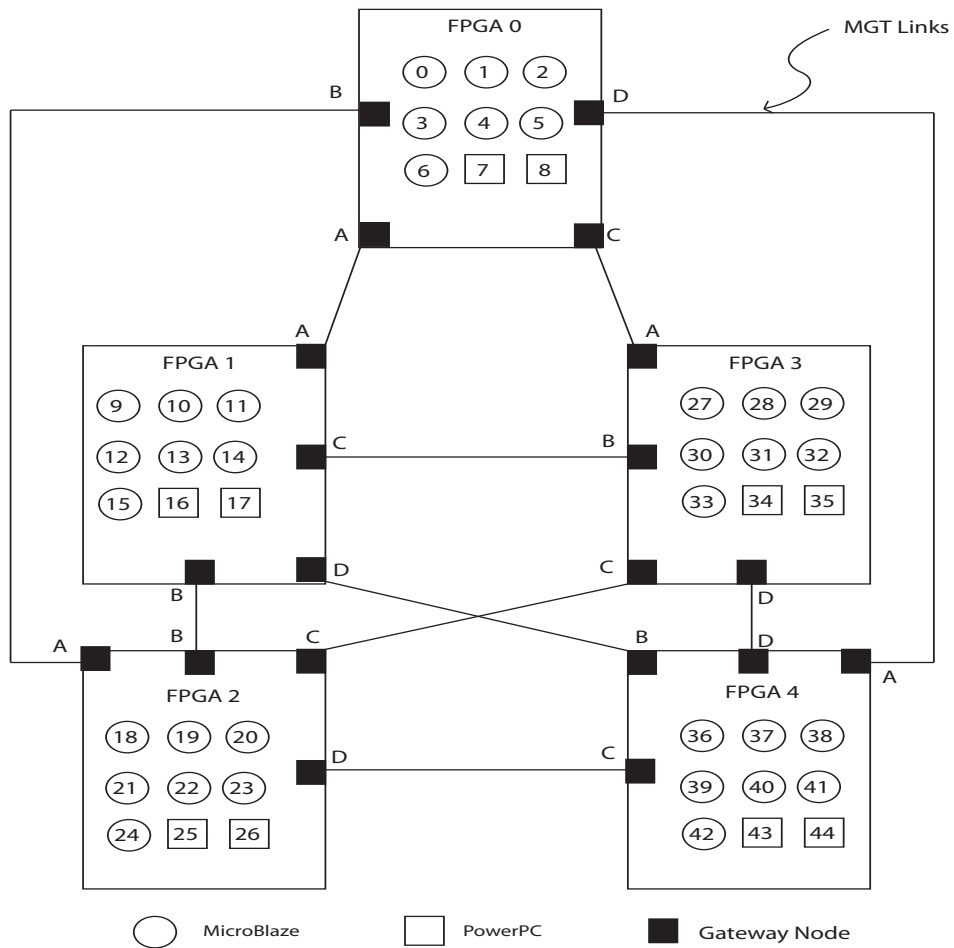


Figure A.1: Physical interconnection diagram of Tier 2 network for the 5-FPGA system with 45 processors.

Appendix B

Physical location of the MGT connectors on the AP1100 boards

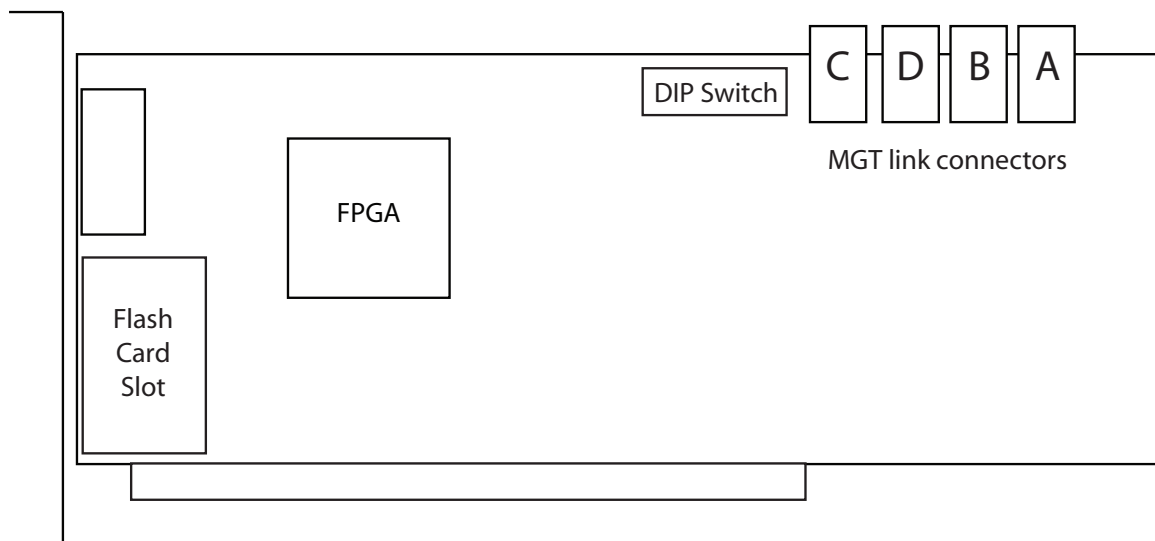


Figure B.1: Physical location of the MGT connectors and its naming convention

Appendix C

TMD-MPI Manual

This appendix contains some details not covered in the body of this thesis, and describes two test projects. Previous knowledge about Xilinx EDK tools is assumed as the details about how to manage a Xilinx project are not described. This appendix is intended for use as a brief manual to configure and execute TMD-MPI programs on the TMD prototype.

INDEX

1. Description of the test projects.
2. IP cores used
3. TMD-MPI files and usage
 - 3.1 How to compile a program
 - 3.2 How to execute a program that uses external links
 - 3.3 Hardware-dependent code
 - 3.4 Important parameters in the mpi.h file
4. Network node ID numbers and routing
5. Debugging the TMD-MPI code

1. Description of the test projects

The TMD-MPI distribution comes with two test projects. The first project (*test_project_simple*) is a single FPGA system with three MicroBlazes. This project provides

a minimal system to understand how TMD-MPI works. There is no off-chip communication, and no external memory to simplify the adaptation to a different board. The second project is a 3-FPGA system, with MGT links for off-chip communication, and external memory. In this project, there is one FPGA per board, and each board has a project (board1, board2 and board3, under the *test_projects_with_mgt* directory). Each FPGA contains three MicroBlazes and they communicate between them through an internal network. Inter-FPGA communication is achieved through the MGT links using a fully-connected topology. The 3-FPGA system is described in detail in Chapter 5 of this thesis. Both test projects execute essentially the same code, which is a benchmark (*TMD-MPI-bench*) that measures basic communication characteristics of the system, such as link bandwidth and latency for internal and external links under no-load condition, link bandwidth and latency with network traffic and synchronization performance.

The *system.ucf* file in every project is written for the Amirix AP1100 board. A proper setup of such file is required to match the FPGA external pins (RS232, MGTs, clocks, external memory) with the hardware board at hand.

In *TMD-MPI-bench*, processor 0 (board1 in the 3-FPGA system) is the root node and it is responsible for reporting the test results to the terminal, and a one-to-one serial cable extension connected to a computer is required to view the output.

2. IP cores used

After unpacking the compressed file that contains the project, the cores required to build the system are located in the `./pcores` directory.

- `clk_rst_startup_2ddr_v1_00_a`: Digital clock management and reset control
- `bridge_v1_00_a`: This is the bridge that performs packet translation
- `fsl_aurora_v1_00_a`: This is the Off-Chip Communications Controller (OCCC)
- `net_if_v4_00_a`: This is the switching network interface that routes packets
- `opb_emc_cypress_v1_10_b`: This is the external SRAM memory controller.

For the single FPGA project, the external memory, the *fsl_aurora*(OCCC) and bridge cores are not present.

3. TMD-MPI files and usage

This section describes how to compile any generic program for the test projects provided using TMD-MPI.

3.1 How to compile a program

TMD-MPI has two files (`mpi.c` and `mpi.h`). In the single FPGA project, the files are located in the

```
<TARGET_DIR>\TMD_MPI_v1\test_project_simple\code
```

directory where `<TARGET_DIR>` is the directory in which the files were decompressed.

In the case of the 3-FPGA system, the TMD-MPI code is located under the

```
<TARGET_DIR>\TMD_MPI_v1\test_project_with_mgt\TMD_MPI
```

and those files are used for the three projects (`board1`, `board2`, and `board3`). Since the MPI code is not under each project's directory, the files have to be referenced with an absolute path instead of a path relative to the project. Adjust the new location of the TMD-MPI code and TMD-MPI-Bench according to the `<TARGET_DIR>` directory.

In some systems, normally with a vendor specific MPI distribution, MPI is used as a library at compile time (*lmpi*) and some system use scripts, such as MPICH, with the *mpicc* script. In TMD-MPI, the `mpi.c` code has to be compiled as another source file together with the main application file. For example:

```
mb-gcc -O2 code/mpi.c code/main.c -o mpi_mb_1/executable.elf
-I./microblaze_1/include/ -Icode/ -L./microblaze_1/lib/ -DRANK0
....
```

where `mb-gcc` is the C compiler for the MicroBlaze, `main.c` is the application code, and `mpi.c` is the TMD-MPI code.

The `-DRANK0` defines the constant `RANK0` to let TMD-MPI know what rank MicroBlaze_1 will have. For MicroBlaze_2, the constant would be defined as `-DRANK1`, and for MicroBlaze_3 the constant would be defined as `RANK2`, and so on. In that way, the mapping from logic processes (ranks in MPI) to physical processors is performed at design time.

Note that the `MicroBlaze_x`, where “x” is the MicroBlaze instance number (physical), has no relation with the rank (logical) assigned to it with the `-D` option at compile time. The “x” is used to assign different names to the various MicroBlaze IP blocks.

The SIMD or MIMD programming models can be used in TMD-MPI. In the SIMD model, there is only one source file, but there are as many executables (.elf files) as processors are on the system. Each executable is compiled with a different RANK constant, as explained above. For the MIMD paradigm, many source files can be assigned to as many processors there are in the system. But again, there will be as many executable files as processors, each one with their own RANK constant specified. A Master-Slave model can be easily implemented, and it is even advisable, because not all the processors have a stdout device and the Master node can be the only node to handle I/O whereas slaves can perform computation tasks only.

The `tmd-mpi-bench.c` program can be compiled for a UNIX environment with a well known MPI distribution, such as MPICH or LAM. The only requirement is that the constant `-DUNIX` has to be defined to prevent the inclusion of MicroBlaze header files that are not present in a UNIX environment. For example, the compilation should be like this

```
mpicc -DUNIX -o tmd-mpi-bench tmd-mpi-bench.c
```

And to execute the program it can be executed as any other MPI program

```
mpirun -np 3 tmd-mpi-bench
```

Either in a UNIX environment or in the FPGA, the output should be similar to this

```
#R0: MPI_Wtime overhead=96 [us]; loop overhead=0 [us]
#R0: STEPS=8, SAMPLES=100
#R0: List test done!
#R0: Sanity check...Performing ring test with 3 processors!(3=3)
#Ring Test Done
#R0: Latency and Bandwidth test (INTERNAL link).
#size_of_message      time [us]      BW[B/s]

      0              15              0
      1              29             137042
      5              31             635185
     10              33            1185071
     50              49            4035451
    100              68            5821357
     500             234            8536475
    1000             437            9148903
#Latency and Bandwidth test...DONE!
#-----
```

```
#BARRIER test;  NUM_BARRIERS=100
#Synch. Overhead: 16474 [barriers/sec]
#BARRIER test DONE!
#-----  \ \
```

3.2 How to execute a program that uses external links

When using external links in the 3-FPGA system, it is recommended to synchronize all the processors by calling `MPI_Barrier` function before exchanging any data between processors. This will wait until all the FPGAs have been programmed and ready to send and receive data.

Also, by calling `MPI_Barrier` at the beginning, makes the order in which the FPGAs have to be program irrelevant, as long as the last FPGA to be programmed is the one that has node 0 (normally the root node); otherwise, some packets may be lost if some of the boards had a previous configuration with the MGTs enabled, as packets would be transfer to other FPGAs that might be reprogrammed later, losing the transferred packets. `MPI_Barrier` ensures that all the nodes have to send a small message to node 0, and by keeping the FPGA (to which node 0 belongs) in a reset state (my `_reset` signal is connected to a dip switch in the Amirix board) will prevent packets to jump over the MGT links, because the system is in reset mode. The bitstream for the FPGA that contains node 0 can be downloaded with the reset switch active. Once the FPGA is programmed, then the switch can be turned-off and then all the packets from processors in other FPGAs, waiting to reach node 0, will pass through the MGTs and the entire multiprocessor system will be synchronized at the beginning.

An alternative way to have all the processors synchronized is to keep all the boards in a reset state, and turn-off the reset switch one by one, leaving the root board at last.

3.3 Hardware dependent code

One of the objectives of TMD-MPI is to increase the portability of application codes to the embedded systems domain. However, there are some restrictions in embedded systems that are not present in a workstation, or hardware details that are hidden to the programmer by an operating system, which TMD-MPI does not use. For example, the following snippet of code from the `tmd_mpi_bench.c` file shows not portable code

```
#ifdef EXT_RAM
    int *buf;

#ifdef RANK0
    buf = (int *)XPAR\_DDR_CONTROLLER_1_BASEADDR;
#endif
```

```

#ifdef RANK1
    buf = (int *)XPAR\_DDR\_CONTROLLER\_2\_BASEADDR;
#endif

#ifdef RANK2
    buf = (int *)XPAR\_OPB\_EMC\_CYPRESS\_1\_MEM0\_BASEADDR;
#endif

#ifdef RANK3
    buf = (int *)XPAR\_DDR\_CONTROLLER\_1\_BASEADDR;
#endif

#ifdef RANK4
    buf = (int *)XPAR\_DDR\_CONTROLLER\_2\_BASEADDR;
#endif

#ifdef RANK5
    buf = (int *)XPAR\_OPB\_EMC\_CYPRESS\_1\_MEM0\_BASEADDR;
#endif

#ifdef RANK6
    buf = (int *)XPAR\_DDR\_CONTROLLER\_1\_BASEADDR;
#endif

#ifdef RANK7
    buf = (int *)XPAR\_DDR\_CONTROLLER\_2\_BASEADDR;
#endif

#ifdef RANK8
    buf = (int *)XPAR\_OPB\_EMC\_CYPRESS\_1\_MEM0\_BASEADDR;
#endif

#else // Not external memory, then use internal memory.
    int buf[5000];
#endif

main() { ....} \

```

This code declares and initializes the pointer `buf` to the base address of the external memory for each processor. Each processor is attached to a different type of memory. Depending on the number of the memory controllers and on the type of memory (SRAM, DDR, etc), a different constant name is used, and has to be specified properly for every processor. Also, the conditional compilation instructions prevent compilation errors in a UNIX environment since the concept of external memory controllers does not apply in a typical multiprocessor computer in same manner as in the Amirix boards. This code has been removed from the single FPGA code because there is no external memory in such system.

A similar situation occurs with the `stdout/stdin` devices. A UART constant has to be defined to prevent errors at compile time for those processors that does not have a UART device. In

Amirix boards, there are only two serial ports, but there are three processors. One processor does not have a stdin/stdout device; therefore *printf* functions can not be used for that particular processor. The compilation errors are avoided by using conditional compilation around the *printf* functions.

Some other devices, such as interrupt controllers and timers present the same problem, but those devices are easily homogenized by TMD-MPI in the `mpi.h` file.

3.4 Some important TMD-MPI parameters in the `mpi.h` file

In the `mpi.h` file there are constants defined that are useful to describe the system. The following list is just a summary of the most important ones; the rest can be found in comments in the code.

MPI_DEBUG_ENABLED Enables the `printf` statements for debugging mode

MPI_SIZE Total number of processes in the system. It is different from the number of physical processors. There may be nine processors (including all the boards) but only use the first four, in which case the `MPI_SIZE` should have the value of 4.

MAX_PENDING_SENDS Defines the maximum number of pending messages (unexpected message queue size).

CLOCK_HZ Specifies the operating frequency of the processors. It is used to compute the `MPI_Wtime` function.

RESET_VALUE Specifies the reset value for the timer. Currently is reset the timer to 0x0000 resulting in a 107.37 sec. per interruption (maximum count) running at 40MHz. If `RESET_VALUE` value is increased, the processor will be interrupted more often.

MGT_MAX_PACKET_SIZE Defines the maximum number of words (each word is 32 bits) per MGT packet.

The most common constant that is likely to change is the `MPI_SIZE` as it defines how many processors are involved in the program execution, is the equivalent to the `np` option in the `mpirun` command.

4. Network and node ID numbers

The OS is in charge of two types of mapping: one is application process to physical processor mapping, and the other is physical processor to network node mapping. Without an OS, in TMD-MPI, the mapping of application processes to physical processors is performed by using the RANK constant at compile time, as explained in section 3.1. The mapping of physical processors to the network nodes is achieved by keeping the network node id number the same as the process number id (MPI RANK). By doing this, the network interfaces will know which processor has which process and the network packet will include directly the TMD-MPI source and destination fields used by the network interface to route the packets accordingly. This is a tight integration between application logic and physical implementation.

Before generating the system, the routing parameters for the network interfaces have to be specified. Once the routing information is set, the system can be synthesized, mapped, placed and routed with the EDK tools. The routing table is static and it is specified in the MHS file. For example, the next snippet from the MHS file in the 3-FPGA system is

```
BEGIN net_if
PARAMETER INSTANCE = net_if_2
PARAMETER HW_VER = 4.00.a
PARAMETER C_CH0_L = 3
PARAMETER C_CH0_H = 5
PARAMETER C_CH1_L = 6
PARAMETER C_CH1_H = 8
PARAMETER C_CH2_L = 2
PARAMETER C_CH2_H = 2
PARAMETER C_CH3_L = 0
PARAMETER C_CH3_H = 0
BUS_INTERFACE to_host = fsl_s_mb_2
BUS_INTERFACE from_host= fsl_mb_s_2
BUS_INTERFACE ch0_in = fsl_a_2
BUS_INTERFACE ch0_out = fsl_2_a
BUS_INTERFACE ch1_in = fsl_b_2
BUS_INTERFACE ch1_out = fsl_2_b
BUS_INTERFACE ch2_in = fsl_3_2
BUS_INTERFACE ch2_out = fsl_2_3
BUS_INTERFACE ch3_in = fsl_1_2
BUS_INTERFACE ch3_out = fsl_2_1
PORT rst_n = sys_rst_s
PORT clk = sys_clk_s
END
```

The routing table for process 1 can be explained as follows:

```
CH0_L = 3
CH0_H = 5
```

means that channel 0 of the network interface is attached to a bridge-OCCC pair that is the gateway to processes 3, 4 and 5. All the packets whose destination is between 3 and 5 are routed through channel 0.

```
CH1_L = 6
```

```
CH1_H = 8
```

similar to channel 0, but for channel 1, and for processes 6 to 8.

```
CH2_L = 2
```

```
CH2_H = 2
```

means that the packets whose destination is 2 will be routed through channel 2.

```
CH3_L = 0
```

```
CH3_H = 0
```

means that the packets whose destination is 0 will be routed through channel 3.

5. Debugging the TMD-MPI code

TMD-MPI has *printf* statements in the code for debugging purposes. They are enabled by the constant `MPI_DEBUG_ENABLED` in the `mpi.h` file. The *printf* statements will print to the terminal when data is received, if there were errors during the transmission of message, if the timer or the interrupt controller could not be initialized, etc. It will make the communications slower and the code larger but useful for debugging.

Appendix D

TMD Configuration using the PCI Bus

This appendix presents detailed information on how to configure the TMD prototype. Section D.1 describe the extension required to the EDK flow to send the bitstream to the Pentium 4 card, and Section D.2 explains how to download the configuration file to the FPGA over the PCI bus using automated scripts.

D.1 Extension to the EDK makefile

To download the bitstreams to the different FPGAs in the TMD prototype an additional feature was added to the Xilinx EDK, which provides two user definable buttons in the IDE. One of the user buttons is used to call a makefile that converts the bitstream to a binary file, which is sent to the Pentium 4 card using a non-interactive FTP session.

```
#filename: tmd.make

include system_incl.make

BOARD = 0
TMD_USER = tmd
TMD_PASSWD = tmd
TMD_IP_ADDRESS = 192.168.0.254
FTP_SCRIPT = ftpscript
TMD_DOW_PATH = tmd_dow

binary:
rm -f $(DOWNLOAD_BIT)
rm -f $(ALL_USER_ELF_FILES)
rm -f $(DOWNLOAD_BIT).$(BOARD).bin
/usr/bin/make -f system.make init_bram;
promgen -s 16384 -c -u 0 $(DOWNLOAD_BIT) -p bin -w -o $(DOWNLOAD_BIT).$(BOARD).bin -b
ls -l $(DOWNLOAD_BIT).$(BOARD).bin
echo $(TMD_USER) > $(FTP_SCRIPT)
echo $(TMD_PASSWD) >> $(FTP_SCRIPT)
```

```

echo "BINARY" >> $(FTP_SCRIPT)
echo "send $(DOWNLOAD_BIT).$(BOARD).bin $(TMD_DOW_PATH)/board_$(BOARD).bin" >> $(FTP_SCRIPT)
echo bye >> $(FTP_SCRIPT)
ftp -s:$(FTP_SCRIPT) $(TMD_IP_ADDRESS)

```

The `BOARD` variable is a unique number assigned to each AP1100 board. This number should start at 0 and incremented by one. The `BOARD` number is used to generate the binary filename on the Pentium 4 card. The `TMD_USER` and `TMD_PASSWD` must be a valid user and password in the Linux system on the Pentium card. The `TMD_IP_ADDRESS` is the IP address of the Pentium 4 card. The `TMD_DOW_PATH` is the path in the Pentium 4 card where the files are be transferred to. In this case, the files are transferred to the `tmd_dow` directory under the home directory for the `tmd` user. The `FTP_SCRIPT` is the filename that is created on-the-fly by the makefile script to use the non-interactive FTP session.

D.2 Commands to use the PCI interface

After all the binary configuration files for all the FPGAs are located in the Pentium 4 card, the download process is performed using the PCI bus controlled by Python scripts on the Pentium 4 card.

The Amirix AP1100 boards have a Flash memory to configure the FPGA on after powering up the boards, or after resetting the boards. The configuration memory has three different banks to store bitstream configurations. Bank 0 has a default configuration from Amirix, banks 1 and 2 are available to the user. The Python scripts use the first user definable configuration bank (bank 1). Note that to have access to the PCI bus in Linux, the user must be login as root. There are three scripts; each one is presented next with an example.

The `tmd_dow.py` script downloads all the bitstreams to all the FPGAs; one bitstream per FPGA. The bitstreams need to follow a naming convention. The bitstream for FPGA 0 should be called *board_0.bit*, for FPGA 1, the bitstream should be called *board_1.bit*, and so forth.

For example, the following command will download the bitstream for the first three FPGAs. After configuring the FPGAs, the script will issue a software reset through the PCI bus to each Amirix FPGA board in reverse order, i.e., from Board 2 down to Board 0.

```
root# tmd_dow 3
```

The following command will download the bitstream to all the FPGAs in the same backplane. Again, after configuring the FPGAs, the script will issue a reset to each FPGA

board.

```
root# tmd_dow all
```

The *dow.py* command downloads a single bitstream to a single FPGA. The syntax is as follows

```
dow.py <bitstream_file> board_number
```

For example, the following command will download the bitstream called *test.bit* to FPGA in board 2. The board will be reseted after configuration.

```
dow.py test.bit 2
```

The *tmd_reset.py* will issue a software-reset in reverse order to all the FPGA boards in the same PCI backplane. This command is useful to restart the execution of a particular configuration.

```
tmd_reset.py
```

Bibliography

- [1] D. Geer. Chip Makers Turn to Multicore Processors. *Computer*, 38(5):11–13, 2005.
- [2] IBM. BlueGene. <http://www.research.ibm.com/bluegene/>.
- [3] Hans Meuer. Top500 supercomputer sites. <http://www.top500.org>. Curr. Jan. 2006.
- [4] R. Kumar, D.M. Tullsen, N.P. Jouppi, and P. Ranganathan. Heterogeneous Chip Multiprocessors. *Computer*, 38(11):32–38, 2005.
- [5] Cray XD1 Supercomputer for Reconfigurable Computing. Technical report, Cray, Inc., 2005. <http://www.cray.com/downloads/FPGADatasheet.pdf>.
- [6] Extraordinary Acceleration of Workflows with Reconfigurable Application-specific Computing from SGI. Technical report, Silicon Graphics, Inc., Nov. 2004. <http://www.sgi.com/pdfs/3721.pdf>.
- [7] General Purpose Reconfigurable Computing Systems. Technical report, SRC Computers, Inc., 2005. <http://www.srccomp.com/>.
- [8] DRC computer. <http://www.drccomputer.com/>.
- [9] Xtreme Data Inc. <http://www.xtremedatainc.com/>.
- [10] FPGA High Performance Computing Alliance. <http://www.fhpca.org/>. Curr. Jan. 2006.
- [11] OpenFPGA - Defining Reconfigurable Supercomputing. <http://www.openfpga.org/>. Curr. Jan. 2006.
- [12] 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.

- [13] Cray XT3 Supercomputer. Technical report, Cray, Inc., 2005. <http://www.cray.com/products/xt3/index.html>.
- [14] Tsuyoshi Hamada, Toshiyuki Fukushige, Atsushi Kawai, and Joshiyuki Makino. PROGRAPE-1: A Programmable Special-Purpose Computer for Many-Body Simulations. In *FCCM*, pages 256–257, 1998.
- [15] Chen Chang, John Wawrzynek, and Robert W. Brodersen. BEE2: A High-End Reconfigurable Computing System. *IEEE Des. Test '05*, 22(2):114–125, 2005.
- [16] Charles L. Cathey, Jason D. Bakos, and Duncan A. Buell. A Reconfigurable Distributed Computing Fabric Exploiting Multilevel Paralellism. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*. IEEE Computer Society Press, 2006.
- [17] J. L. Hennessy and D. A. Patterson. *Computer Architecture: a quantitative approach*. Morgan Kaufman, 2004.
- [18] David W. Wall. Limits of Instruction-Level Parallelism. In *ASPLOS-IV: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, New York, NY, USA, 1991. ACM Press.
- [19] Luca Benini and Giovanni De Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1):70–78, 2002.
- [20] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.
- [21] MPI. <http://www-unix.mcs.anl.gov/mpi/>.
- [22] R. Butler and E. Lusk. Monitors, Messages, and Clusters: The p4 Parallel Programming System. *Parallel Computing*, 20(1), 1994.
- [23] The MPI Forum. MPI: a message passing interface. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 878–883, New York, NY, USA, 1993. ACM Press.
- [24] 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.

- [25] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.
- [26] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open MPI: A Flexible High Performance MPI. In *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*, Poznan, Poland, September 2005.
- [27] William Gropp and Ewing Lusk. An Abstract Device Definition to Support the Implementation of High-Level Point-to-Point Message Passing Interface. Technical report, MCS-P392-1193, Argone National Laboratory, 1994.
- [28] William Gropp and Ewing Lusk. MPICH Working Note: Creating a New MPICH Device Using the Channel Interface. Technical report, ANL/MCS-TM-213, Argonne National Laboratory, 1995.
- [29] J. A. Williams and X. Xie N. W. Bergmann. FIFO Communication Models in Operating Systems for Reconfigurable Computing. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 277–278. IEEE Computer Society Press, 2005.
- [30] A. Jerraya and W. Wolf, editors. *Multiprocessor Systems-on-Chip*. Morgan Kaufman, 2004.
- [31] A D&T Roundtable: Are Single-Chip Multiprocessors in Reach? *IEEE Design and Test of Computers*, 18(1):82–89, 2001.
- [32] Mohamed-Wassim Youssef, Sungjoo Yoo, Arif Sasongko, Yanick Paviot, and Ahmed A. Jerraya. Debugging HW/SW Interface for MPSoC: 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.
- [33] John A. Williams, Neil W. Bergmann, and Robert F. Hodson. A Linux-based Software Platform for the Reconfigurable Scalable Computing Project. In *MAPLD International Conference*, Washington, D.C., USA, 2005.
- [34] MPI/RT Forum. <http://www.mpirt.org/>.
- [35] Poletti Francesco, Poggiali Antonio, and Paul 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.

- [36] Pierre 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.
- [37] V. Aggarwal, I. A. Troxel, and A. D. George. Design and Analysis of Parallel N-Queens on Reconfigurable Hardware with Handel-C and MPI. In *2004 MAPLD International Conference*, Washington, DC, USA, 2004.
- [38] Verari Systems, Inc. <http://www.mpi-softtech.com/>.
- [39] 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.
- [40] Chen Chang, John Wawrzynek, and Robert W. Brodersen. BEE2: A High-End Reconfigurable Computing System. *IEEE Design and Test of Computers*, 22(2):114–125, 2005.
- [41] Keith D. Underwood, K. Scott Hemmert, Arun Rodrigues, Richard Murphy, and Ron Brightwell. A Hardware Acceleration Unit for MPI Queue Processing. In *Proceedings of IPDPS '05*, page 96.2, Washington, DC, USA, 2005. IEEE Computer Society.
- [42] 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.
- [43] Amirix Systems, Inc. <http://www.amirix.com/>.
- [44] Xilinx, Inc. <http://www.xilinx.com>.
- [45] Lesley Shannon and Paul Chow. Maximizing System Performance: Using Reconfigurability to Monitor System Communications. In *International Conference on Field-Programmable Technology (FPT)*, pages 231–238, Brisbane, Australia, December 2004.
- [46] C. Comis. A High-Speed Inter-Process Communication Architecture for FPGA-based Hardware Acceleration of Molecular Dynamics. Master's thesis, University of Toronto, 2005.

- [47] Manuel Saldaña, Daniel Nunes, Emanuel Ramalho, and Paul Chow. Configuration and Programming of Heterogeneous Multiprocessors on a Multi-FPGA System Using TMD-MPI. In *In the Proceedings of the 3rd International Conference on Reconfigurable Computing and FPGAs.*, San Luis Pototsi, Mexico, 2006., *To appear*.
- [48] David Ku and Giovanni DeMicheli. HardwareC - A Language for Hardware Design. Technical Report CSL-TR-90-419, Stanford University, 1988.
- [49] Handel-C Documentation. <http://www.celoxica.com>. Curr. Jan. 2006.
- [50] D. Soderman and Y. Panchul. Implementing C Algorithms in Reconfigurable Hardware Using c2verilog. In *Proceedings of FCCM '98*, page 339, Washington, DC, USA, 1998. IEEE Computer Society.
- [51] William Gropp and Ewing 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.
- [52] ParkBench Project. <http://www.netlib.org/parkbench/>.
- [53] Intel, Inc. MPI Benchmark 2.3. <http://www.intel.com/>.
- [54] SKaMPI Project. <http://iinwww.ira.uka.de/skampi/>.
- [55] Jianping Zhu, editor. *Solving Partial Differential Equations on Parallel Computers*. World Scientific, 1994.