View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by The University of North Carolina at Greensboro

HARDWARE DESIGN OF MESSAGE PASSING ARCHITECTURE ON HETEROGENEOUS SYSTEM

by

Shanyuan Gao

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Electrical Engineering

Charlotte

2013

Approved by:

____
Dr. Ronald R. Sass

____
Dr. James M. Conrad

____
Dr. Jiang Xie

____
Dr. Stanislav Molchanov

© 2013
Shanyuan Gao
ALL RIGHTS RESERVED

ABSTRACT

SHANYUAN GAO. Hardware design of message passing architecture on heterogeneous system.
(Under the direction of DR. RONALD R. SASS)

Heterogeneous multi/many-core chips are commonly used in today's top tier supercomputers. Similar heterogeneous processing elements — or, computation accelerators — are commonly found in FPGA systems. Within both multi/many-core chips and FPGA systems, the on-chip network plays a critical role by connecting these processing elements together. However, The common use of the on-chip network is for point-to-point communication between on-chip components and the memory interface. As the system scales up with more nodes, traditional programming methods, such as MPI, cannot effectively use the on-chip network and the off-chip network, therefore could make communication the performance bottleneck.

This research proposes a MPI-like Message Passing Engine (MPE) as part of the on-chip network, providing point-to-point and collective communication primitives in hardware. On one hand, the MPE improves the communication performance by offloading the communication workload from the general processing elements. On the other hand, the MPE provides direct interface to the heterogeneous processing elements which can eliminate the data path going around the OS and libraries. Detailed experimental results have shown that the MPE can significantly reduce the communication time and improve the overall performance, especially for heterogeneous computing systems because of the tight coupling with the network. Additionally, a hybrid "MPI+X" computing system is tested and it shows MPE can effectively offload the communications and let the processing elements play their strengths on the computation.

ACKNOWLEDGMENTS

No words can express my appreciation to my advisor, Dr. Ron Sass. Your wisdom and character have persistently nurtured and guided me through all the joys and difficulties during this journey, and continually so.

I am grateful to my committee, Professor James Conrad, Professor Jiang Xie, and Professor Stanislav Molchanov. Your advices have challenged me in every aspect and perfected this work.

I would like to thank the RCS lab (Andy, Will, Robin, Bin, Scott, Yamuna, Rahul, Ashwin, Shweta, and countless others). Our untiring discussions have inspired me to explore new directions. Your help has contributed to many parts of this work.

I would like to thank my mother and my father for their full-hearted support.

Rong, thank you for your love. Without you, nothing of this matters.

TABLE OF CONTENTS

LIST OF TABLES viii

LIST OF FIGURES ix

LIST OF ABBREVIATIONS xi

CHAPTER 1: INTRODUCTION 1

1.1 High-Performance Computing 1

1.2 Interconnect and Communication 2

1.3 Motivation 4

1.4 Thesis Question 7

CHAPTER 2: BACKGROUND 11

2.1 Field Programmable Gate Array 11

2.2 Computational Science 12

2.3 Top500 Supercomputers 12

2.4 Message-Passing 13

2.5 Benchmarks 15

2.6 Amdahl's Law 17

2.7 Communication Model 17

CHAPTER 3: RELATED WORK 19

3.1 MPI Related Research 19

3.1.1 Point-to-point Communication 19

3.1.2 Collective Communication 19

3.1.3 Hardware Optimization 21

3.2 On-chip Message-Passing 23

3.2.1 Raw 23

3.2.2 Intel Terascale Computing 24

3.2.3 RAMP 24

3.2.4 Reconfigurable Computing Cluster 25

CHAPTER 4: DESIGN 26

4.1 Design Infrastructure 26

4.1.1 Off-chip Network 26

4.1.2 On-chip Network 27

4.1.3 Network Interface 28

4.1.4 Base System 28

4.1.5 Miscellaneous IP Cores 30

4.2 Stage 1: Hardware Message-Passing Engine 30

4.2.1 Point-to-point Communication 30

4.2.2 Collective Communication 31

4.3 Stage 2: Heterogeneous System 35

4.3.1 Parallel FFT Operation 38

4.3.2 Parallel Matrix-Vector Multiplication 41

CHAPTER 5: EVALUATION AND ANALYSIS 45

5.1 Evaluation Infrastructure 45

5.2 Testing Methodology 45

5.3 Stage 1 Experiment 46

5.3.1 Barrier Performance Result 46

5.3.2 Broadcast Performance Result 48

5.3.3 Reduce Performance Result 52

5.3.4 Allreduce Performance Result 54

5.3.5 Summary 55

5.4 Communication Model 57

5.4.1 Linear Fitting for Barrier 57

5.4.2 Latency Model 58

5.5 Stage 2 Experiment 63

5.5.1 Parallel Fast Fourier Transformation 65
5.5.2 Parallel Matrix-Vector Multiplication 68
5.5.3 Hybrid Computing System 71
5.6 Validation 72
CHAPTER 6: CONCLUSION 75
REFERENCES 77

LIST OF TABLES

TABLE 1.1: Top500 HPC trend 3

TABLE 5.1: `Broadcast` measurement vs. simulation 59

TABLE 5.2: `Reduce` measurement vs. simulation 63

TABLE 5.3: Bandwidth measurement vs. simulation 64

TABLE 5.4: Resource utilization of hardware MPE 73

TABLE 5.5: Performance improvement of hardware MPE 73

TABLE 5.6: Performance improvement of MPE on heterogeneous systems 74

LIST OF FIGURES

FIGURE 1.1: Configuration of heterogeneous HPC system 4
FIGURE 1.2: Communication of future heterogeneous HPC system 5
FIGURE 1.3: Proportion of collective operation in synthetic benchmark 6
FIGURE 1.4: Traditional operation flow 8
FIGURE 1.5: Operation flow of hardware message-passing 8
FIGURE 2.1: Diagram of `barrier` operation 14
FIGURE 2.2: Diagram of `broadcast` operation 15
FIGURE 2.3: Diagram of `reduce` operation 15
FIGURE 2.4: Amdahl's Law, performance gain of parallelism 18
FIGURE 4.1: Direct connected off-chip network with the router 27
FIGURE 4.2: 4-ary 3-cube torus network 27
FIGURE 4.3: On-chip components connected around the crossbar switch 28
FIGURE 4.4: Signal interface of LocalLink 28
FIGURE 4.5: Hardware base system with the on-chip router 29
FIGURE 4.6: FSM of `barrier` operation 32
FIGURE 4.7: FSM of `broadcast` operation 33
FIGURE 4.8: FSM of `reduce` operation 34
FIGURE 4.9: Topologies of hardware collective communication 34
FIGURE 4.10: Block diagram of MPE in heterogeneous system 36
FIGURE 4.11: Typical programming method for parallel heterogeneous system 37
FIGURE 4.12: FFT Decimation-In-Time 39
FIGURE 4.13: FFT Decimation-In-Frequency 39
FIGURE 4.14: Block diagram of FFT Core 41
FIGURE 4.15: Block diagram of FFT IO 42
FIGURE 4.16: Block diagram of vector-vector multiplication 43

FIGURE 4.17: Hybrid of hardware thread and software thread 44

FIGURE 5.1: MPE `barrier` using different topologies 47

FIGURE 5.2: Software `barrier` 47

FIGURE 5.3: Bandwidth of different `broadcast` topologies 49

FIGURE 5.4: Bandwidth of software `broadcast` 49

FIGURE 5.5: Latency of different `broadcast` topologies 51

FIGURE 5.6: Latency of software `broadcast` 51

FIGURE 5.7: Bandwidth of different `reduce` topologies 53

FIGURE 5.8: Bandwidth of software `reduce` 53

FIGURE 5.9: Latency of different `reduce` topologies 54

FIGURE 5.10: Latency of software `reduce` 55

FIGURE 5.11: Execution time of different `allreduce` topologies 56

FIGURE 5.12: Latency of software `allreduce` 56

FIGURE 5.13: Mathematic fitting for MPE `barrier` 58

FIGURE 5.14: Time chart of `broadcast` operation 59

FIGURE 5.15: Latency simulation of `broadcast` 60

FIGURE 5.16: Time chart of `reduce` operation 61

FIGURE 5.17: Latency simulation of `reduce` 62

FIGURE 5.18: Max bandwidth simulation of `broadcast` and `reduce` 64

FIGURE 5.19: Communication impact on software FFT 65

FIGURE 5.20: Comparison of communication with software FFT 66

FIGURE 5.21: Communication impact on hardware FFT 67

FIGURE 5.22: Comparison of communication with hardware FFT 67

FIGURE 5.23: Communication impact on software MACC computation 68

FIGURE 5.24: Communication impact on accelerated MACC computation 69

FIGURE 5.25: Hardware MACC and MPE 70

FIGURE 5.26: Communication impact on hybrid computing system 72

LIST OF ABBREVIATIONS

CPU	Central Processing Unit
COTS	Commodity Off The Shelf
IC	Integrated Circuit
PCB	Printed Circuit Board
IP	Intellectual Property
FPU	Floating Point Unit
FPGA	Field Programmable Gate Array
Flops	Floating point operations per second
FIFO	First In, First Out
PetaFlops	10^{15} Flops
TeraFlops	10^{12} Flops
GigaFlops	10^{9} Flops
MPI	Message-Passing Interface
PE	Processing Element
SOC	System on Chip
PLB	Processor Local Bus
MPMC	Multiport Memory Controllor
PC	Personal Computer
UART	Universal Asynchronous Receiver/Transmitter
P2P	Point-to-point

CHAPTER 1: INTRODUCTION

Frequency scaling has played a major role in pushing the computer industry forward. Nevertheless, due to the memory wall, the instruction-level parallelism (ILP) wall, and the power wall [1, 2], conventional frequency scaling has shown diminishing returns in performance in past few years. As a result, academia and industry research has shifted the focus towards the multi/many-core era. New single-chip architectures have been designed and manufactured to explore and exploit parallelism rather than single-thread performance. To make use of the massive amount of cores — or, processing elements (PEs) — the on-chip and off-chip interconnect becomes the critical component of these new architectures. Presently, these multi/many-core processors follow traditional multi-chip symmetric multiprocessor (SMP) designs, which are integrated onto a single chip. Consequently, interconnect designs mainly provide point-to-point communication and are mostly used for the general shared-memory processor model. This is sufficient for desktop personal computers; however, in large scale systems with many heterogeneous PEs, this could lead to seriously inefficient communication and make the general-purpose processor the bottleneck of the systems.

1.1 High-Performance Computing

High-Performance Computing (HPC) focuses on computing methodologies that solve complex computational problems in the shortest possible time. High-performance computers, often called supercomputers, are machines built to fulfill these computing needs. Frequency scaling, while pushing the PC industry forward, also benefitted the HPC world in the form of commodity off-the-shelf (COTS) clusters, also known as Beowulf style clusters [3]. These clusters achieved great success by integrating low cost commodity components, and therefore became the mainstream of HPC ma-

chines in the commercial market. Traditionally, HPCs were built in a homogeneous fashion, in which uniformly distributed general-purpose processors were used. In recent years, researchers built heterogeneous HPC system that incorporated not only general-purpose PEs, such as the Dual-core and Quad-core processors from Intel and AMD [4, 5], but also some modern multi/many-core computing accelerators, such as General Purpose computation on Graphics Processing Units (i.e. GPGPUs) from Nvidia and Cell Broadband Engine (Cell B.E.) from a Sony Toshiba IBM partnership [6, 7]. The newly built machines achieved PetaFlops computing milestone in June 2008 [8]. Up until now, more HPC systems are using heterogeneous components, and the trend is heating up. However, as more heterogeneous components are used in these ever-larger HPC systems, the communication hierarchy between these PEs becomes more complex, which could possibly slow down the progress towards the next HPC target — Exascale Computing [9].

1.2 Interconnect and Communication

The term *interconnect* can be used in different ways. From the PC point of view, the interconnect connects the discrete chip-sets together, for example, the processor ICs, memory, video card, and other peripherals. A *bus* is the common term for an interconnect that shares physical connections. The peripherals connected to the bus are often categorized as masters and slaves. Because sharing mechanism could cause contention, in some systems, multiple buses can be used.

With the emergence of multicore and System on-Chip, multiple components can be pushed into one single silicon device. Traditional system interconnects, such as buses, are apparently inadequate because the growing number of on-chip components would compete for the sharing resources. On-chip networks, proposed for modern multicore architecture, can take advantage of the hardware that has very short signaling and is tightly coupled with the on-chip components, thereby providing efficient communication between the on-chip components. The common use of the on-chip net-

Table 1.1: Top500 HPC trend

TOP500 list	Sys 2011	Sys 2012	Perf. 2011	Perf. 2012
Infiniband:	41.8%	44.8%	38.7%	32.5%
Gigabit Ethernet:	44.8%	37.8%	19.3%	12.6%
Custom Interconnect:	N/A	N/A	24.1%	36.8%

work is point-to-point communication, such as Intel QPI [10, 11] and HyperTransport [12, 13].

In the HPC world, the interconnect has another definition. These interconnects either directly or indirectly connect the distributed systems together. Each distributed system has its own OS and libraries, which handle the communication between each system. The Beowulf style cluster [3, 14] uses many cost-effective COTS components, has made Fast Ethernet and Gigabit Ethernet popular in HPC systems. There are some less popular interconnects as well: Some obsolete HPC systems use proprietary interconnect, such as Connection Machine [15], iWarp [16], and IBM SP-2 [17]. Nowadays, most commercial interconnects are standardized, such as Quadrics [18], Myrinet [19] and InfiniBand [20, 21].

In recent Top500 lists there is an interesting observation about the interconnect family. In Table 1.1, it shows that from 2011 to 2012, Gigabit Ethernet lost system shares while Infiniband increased its system shares. From performance point of view, both Gigabit Ethernet and InfiniBand lost shares against custom network. Another interesting observation is that the top machines on Top500 list all possess custom or proprietary interconnects. Although the performance gap can be due to many reasons — such as processor types, number of processors, or operating systems — one obvious reason is the use of different interconnect.

As the HPC world is shifting to use modern multicore processors, the interconnect hierarchy in the heterogeneous HPC system is becoming very complex. As illustrated in Figure 1.1, one PCB board could host multiple sockets of general pro-

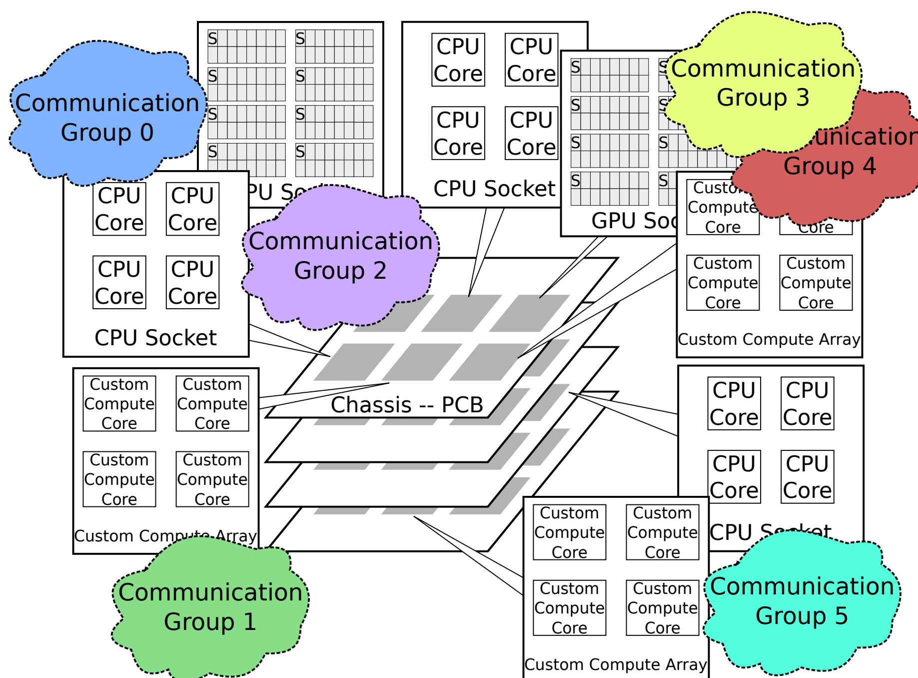


Figure 1.1: Configuration of heterogeneous HPC system

cessors (CPUs), several heterogeneous processors (i.e. GPGPUs), and some custom computing accelerator chips. (To be general, processing elements (PEs) is used to represent CPU cores, GPU cores, or any other hardware accelerator cores in the following text unless otherwise mentioned.) Within each chip, PEs are connected via a certain type of on-chip network. Off-chip networks are used to connect these packaged ICs and PCB boards together. When communications occur, a single chip can possibly participate in multiple communication groups. Figure 1.2 shows the rack view of PEs involved in communications; the number denotes which communication group the core is involved in. One communication could use PEs across the entire HPC system, such as the PEs on different silicon devices or on different PCB boards. Because of different physical locations, the communication time between PEs could be non-deterministic.

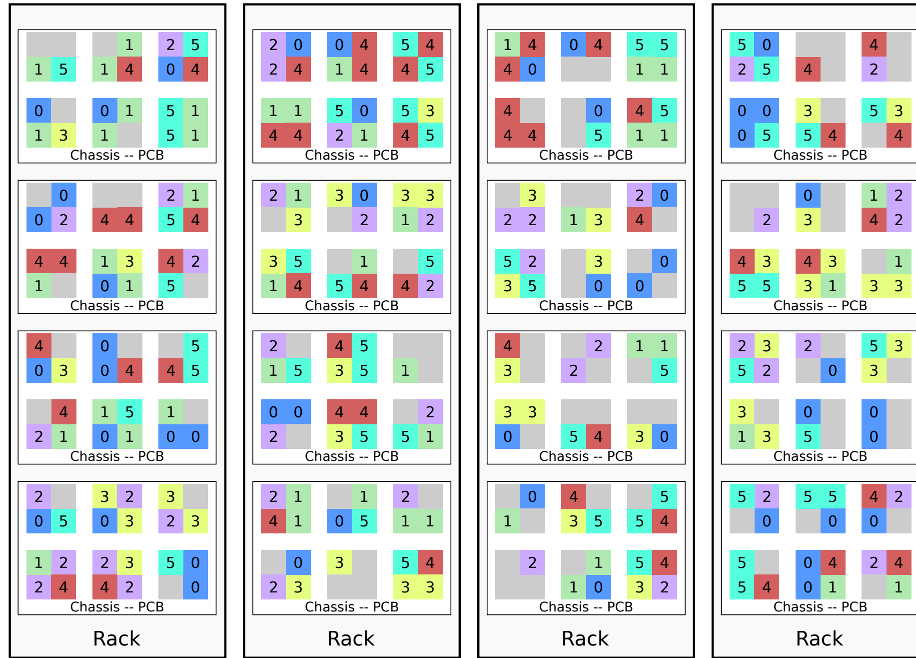
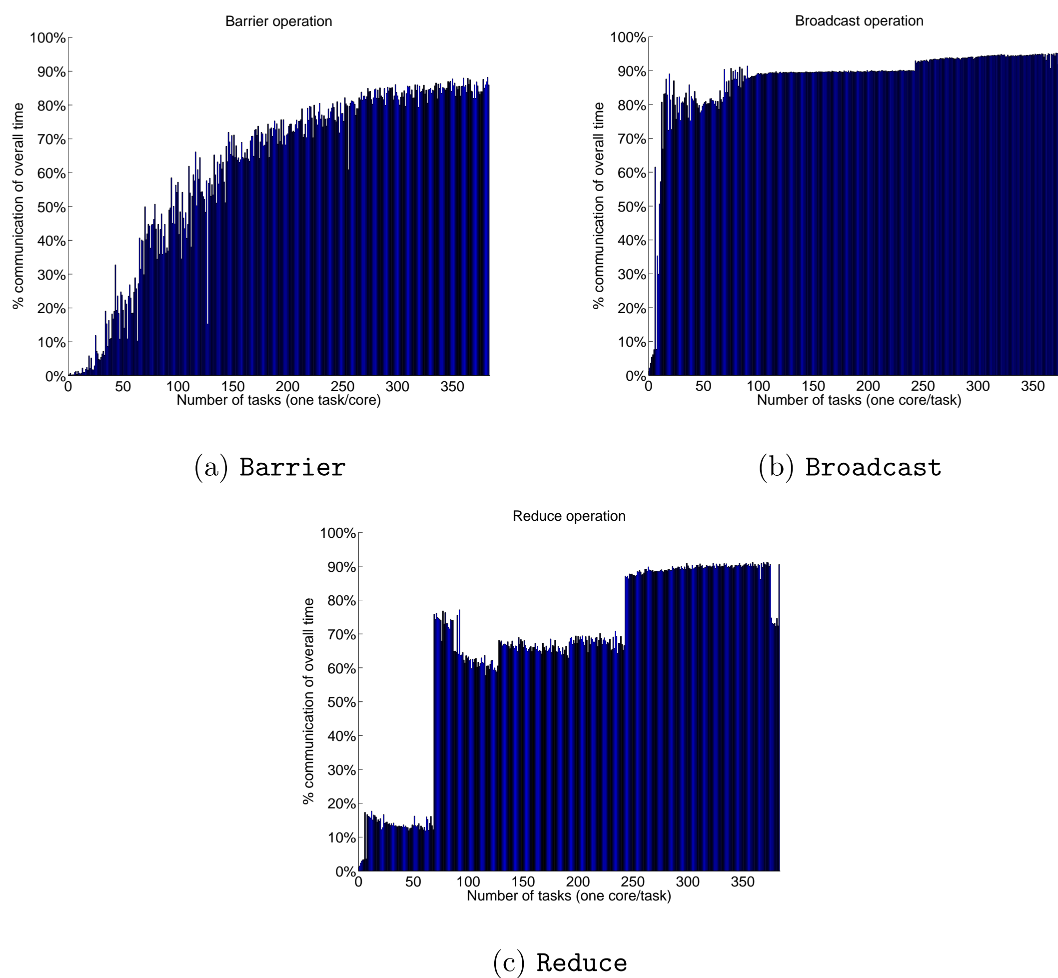


Figure 1.2: Communication of future heterogeneous HPC system

1.3 Motivation

With the massive amount of PEs working in parallel, it will generate large volume of communication for coordinating and exchanging data. Depending on the hierarchical position, the communication time between the PEs will vary in a dispersed range. To find out how the large volume of communication is affecting the overall performance on the real HPC system, a preliminary test has been performed on a commercial multicore HPC cluster. The Python cluster, located at UNC Charlotte, consists of 384 computing cores, with both Gigabit Ethernet and QDR InfiniBand interconnect [22]. Using standard Message-Passing Interface (MPI) OpenMPI 1.4.3 [23] with VampirTrace [24], a synthetic benchmark was written to test collective communications against a simple calculation. By keeping the total problem size constant and varying the number of computation units (tasks), we are able to profile the time each subroutine occupies the whole benchmark. The ratio of communication and computation is reported in Figure 1.3.



(a) `Barrier` (b) `Broadcast`

(c) `Reduce`

Figure 1.3: Proportion of collective operation in synthetic benchmark

It can be observed in Figure 1.3 that as the number of tasks grows, the proportion of communication time increases while the proportion of computation time decreases. This occurs for two reasons:

- First, increasing the number of tasks, n, requires more communication (i.e. a large n means more tasks have to coordinate).
- Second, the computation time decreases because an increasing n divides a fixed-size problem into smaller tasks (W/n).

(Note that the synthetic benchmark may not describe the behavior of all real applications. Often, the user will increase the problem size as the system scales up. However, the synthetic test does highlight the trend that increasing number of tasks would make communication the bottleneck in the whole system.)

To solve this communication bottleneck, many research efforts have been put into optimizing MPI in traditional homogeneous systems. As will be seen in Chapter 3, the software nature of MPI has limited the performance improvement. The standard MPI communication needs to pass multiple software protocol stacks, which introduces overhead, as shown in Figure 1.4 (a). In the heterogeneous HPC system, the communication bottleneck becomes worse because not all the PEs are able to host a full-fledged OS. The communication between these PEs is still handled by the host CPU. Although the communication load can be small, when the number of PEs is large, the volume of communication will increase, therefore saturate the CPU and create the bottleneck, as shown in Figure 1.4 (b).

With the abundant transistor resources, we conjecture that moving some MPI operations into hardware can avoid the traditional protocol stacks, which leaves a small amount of interactions with the OS and the host CPU, as shown in Figure 1.5 (a). Moving message-passing function can also apply to heterogeneous systems. With the direct messaging function in hardware, these heterogeneous PEs can talk to the network, without overloading the general processor, as shown in Figure 1.5 (b).

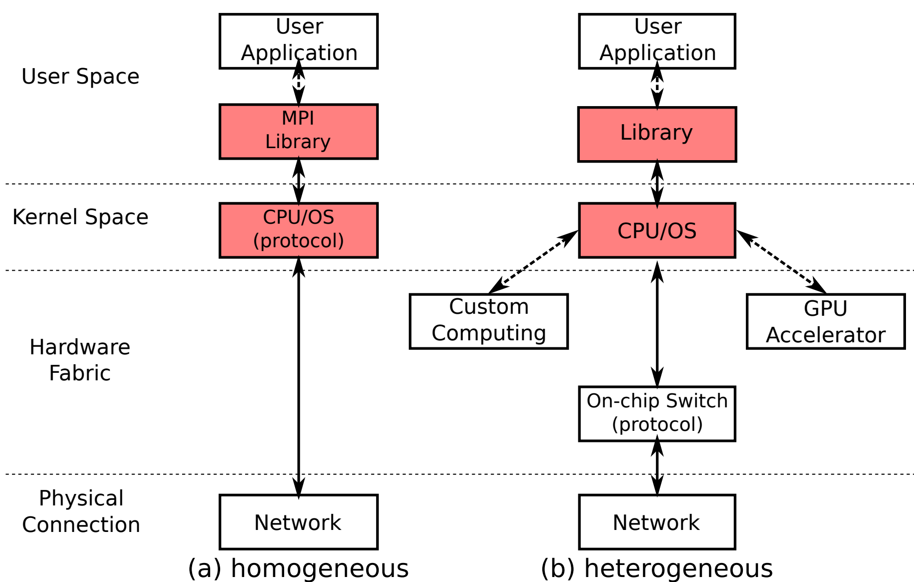


Figure 1.4: Traditional operation flow

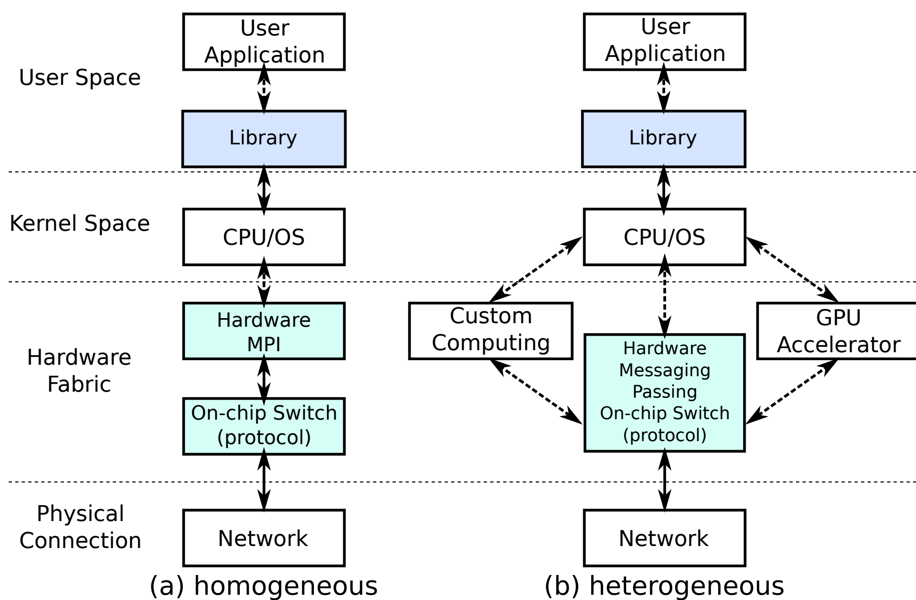


Figure 1.5: Operation flow of hardware message-passing

1.4 Thesis Question

Future VLSI technology will have millions of heterogeneous PEs assembled into one single HPC system. When working together, these PEs could produce a large volume of communication for coordinating and exchanging data. In order to provide communication for these PEs, a complex hierarchical interconnect should be used, which would exhibit diverse communication time between PEs in different hierarchical levels. Traditionally, communications between the PEs are handled in software, such as OS or libraries. As the volume of communication increases, the software can become the bottleneck because of the software overhead.

While the VLSI technology enables multi/many-core heterogeneous PEs on chip, it also provides silicon resources to build the on-chip network, which can be used to handle communications directly in hardware. The streaming and parallel nature of hardware on-chip network would provide short signaling and tight coupling between the PEs. Nowadays, on-chip networks are commonly used for point-to-point communications between 4 or 8 cores; however, it is not practical for hundreds of cores because point-to-point communication requests would sequentially line up, thereby become the bottleneck even in hardware. Facing the future massive amount of on-chip and off-chip PEs and the complex interconnect hierarchy, the question arises: *Can hardware be used to provide a unified view of the heterogeneous system and provide message-passing function to the chip as well as to the cluster?*

To answer this question, a custom hardware communication engine will be designed and tested against the traditional software communication method. With different sets of experiments, the thesis question can be answered in following aspects:

- Is the hardware communication engine practical and feasible? If the communication function can be implemented in hardware and function as the traditional software communication, it is a practical design. If the hardware communica-

tion engine consumes reasonable amount of hardware resources — on par with a general processor — the hardware communication engine is feasible.

- Can the hardware communication engine improve the overall performance? By comparing different experimental configurations, we can quantitatively study the advantages and the limitations of the hardware communication engine.
- Is the hardware communication engine scalable when the system grows? Measuring the detailed communication cost and the software overhead, a model can be established to predict the performance beyond the test infrastructure. Comparing with the software communication, if the hardware communication engine shows similar or slower growth trend of the scalability, it is recognized as scalable.
- Can the hardware communication engine be used in the heterogeneous system? A heterogeneous computing environment will be designed. Hardware computing accelerators would interact with the hardware communication engine directly without involving the central processor. The implementation would prove the applicability.
- In the heterogeneous system, can hardware communication engine bring performance gain? Tests will be designed to measure the communication time of the hardware communication engine or the software communication.

CHAPTER 2: BACKGROUND

This chapter provides background knowledge for the proposed research — covering FPGA, computational science, Top500 supercomputers, Message-Passing Interface, benchmarks, and communication model. Technical and research related work can be found in in Chapter 3.

2.1 Field Programmable Gate Array

Field Programmable Gate Array (FPGA) is a reconfigurable IC on which the hardware fabric can be modified and programmed after manufactured. To program an FPGA, Hardware Description Language (HDL) is commonly used to describe the hardware wiring in register-transfer level. Using tools from the vendor or from the third party, an HDL design is translated and implemented as a device specific bitstream, which can be used to program the device. Intellectual Properties (IPs) are design blocks which are modularized and can be inserted into the design with small or no configurations. There are two types of IP: One is called soft IP, which normally contains logic design only. This type of IP is portable and requires FPGA tools to translate into fabrics. The other type of IP is called diffused IP, sometimes known as hard IP. This type of IP is a set of device specific circuits which are already implemented in the device, such as on-chip memory blocks, high-speed transceivers, and processor cores. To make use of the diffused IP, the designer needs to instantiate the IP and connect IO signals in the design, and the tools would wire the signals and activate the IP.

Because of the reconfigurability, FPGA is often used to quickly prototype and validate the hardware design. The hardware characteristics of FPGA has also made it a nature fit to execute some complicated operations in hardware. Furthermore, FPGA

is capable of processing operations in parallel. Therefore, FPGA is often used as co-processor in some applications, which sometimes can achieve orders of magnitude performance improvement compared to traditional general processor system [25, 26]. Further details of how FPGA works and how it is implemented can be referred to [27].

2.2 Computational Science

The advancement of science and technology has made the scale of research, design, and decision systems large and complex. The traditional theoretic and experimental approaches to solve these problems become less efficient and sometimes can barely meet the requirement. Computational science, an emerging approach based on the development of modern computing technology, opened the door to some new scientific and engineering areas, such as bioinformatics, computational fluid dynamics, financial modeling, etc.

Computational science problem, built upon certain mathematical models and numerical algorithms, normally requires huge amount of computation. Generally speaking, the computation is not possible to be accomplished by a single workstation in the required time. The straightforward answer to solve the computational science problem is to use supercomputers, which generally consist of many computation units computing in parallel. By breaking the big problem into smaller pieces and distributing the pieces to the computation units, the users can exploit the parallelism of computation and solve the computational problem efficiently.

2.3 Top500 Supercomputers

Started from 1993, the Top500 has been ranking the world's fastest computers biannually. Back in the June 1993, because building a supercomputer required strong financial support, only big companies such as CRAY, Thinking Machines, Fujitsu, and HP were the major players in the industry. At that time, each company has its own processor design and proprietary architecture. The number of the processors

were small: 20% of the machines on the list had single processor, and nearly half of the machines on list had less or equal than four processors.

As technology advanced, powerful processors were designed and the architecture of these supercomputers were changed. In June 2000, more than 80% of the machines on the list had 33 to 256 processors. Scalar processors ruled the market. Cluster architecture started to dominate the market. Specifically, Beowulf Cluster — featuring off-the-shelf hardware components, open source software, and Ethernet connection — were very cost-effective to provide HPC power to the budget-limiting users. From June 2000 to June 2003, the ratio of cluster architecture in Top500 list grew from 6.4% to 29.8%

Cluster architecture continued to dominate the market in the past few years ($>$ 80%). In the latest Top500 list, June 2011, 4k to 16k processors were the typical number of processors in a machine. Roadrunner [8], the first machine reached PetaScale in June 2008, was outperformed by Jaguar [28] with 1.7 PetaFlops in November 2009. Tianhe-1A [29], surpassed Jaguar with 2.566 PetaFlops after 12 months. However, K [30], challenged Tianhe-1A with 8.162 PetaFlops in June 2011, after another 6 months it reached 10.51 PetaFlops [31]. 4-core, 6-core general processors were used in most of the machines. Modern processors, such as GPGPUs and Cell B.E. became popular in the list.

2.4 Message-Passing

Along with the development of the hardware, as mentioned in section 2.3, there are several programming model for the HPC systems. However, Message-Passing has been practically the *de facto* standard for programming HPC machines. The Message-Passing standard specifies the programming model for moving data explicitly between the tasks. Message-Passing Interface (MPI), is the library specification for implementing the standard. There are several MPI implementations maintained by different research groups, such as MPICH and OpenMPI [32, 23].

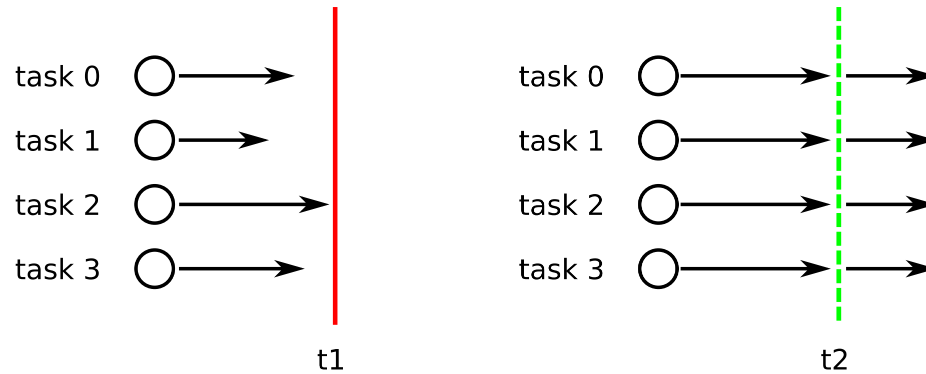


Figure 2.1: Diagram of `barrier` operation

Two types of communication primitives are widely used in MPI applications. One is point-to-point operation, which involves communication between exact two tasks. The other type is called collective communication, which involves communication among a group of tasks. Though there are some variants of point-to-point operations, the functions are essentially the same — passing data from one task to the other. Collective communications, on the contrary, perform various duties — including synchronization, exchanging data, or performing computations.

Among all the collective communication primitives, `barrier` operation is a relatively simple but important operation; it is widely used in MPI as well as other programming models. The function of `barrier` is to synchronize multiple parallel tasks, and it is critical to maintain correct ordering of parallel operations in some algorithms. The semantics of `barrier` is to block all tasks when they enter the operation and wait until every task has reached the barrier. At that point, all tasks are allowed to proceed. Shown in Figure 2.1, at $t1$, task 1 reaches barrier, but it needs to wait for tasks. At $t2$, all tasks reach the barrier and are released thereafter.

`Broadcast` is another frequently used collective primitive. Literally, `broadcast` distributes the data from the source task to all the other tasks in the communication. As shown in Figure 2.2, before the `broadcast`, tasks own different data. After the operation, all the recipient tasks own the same data as the source task.

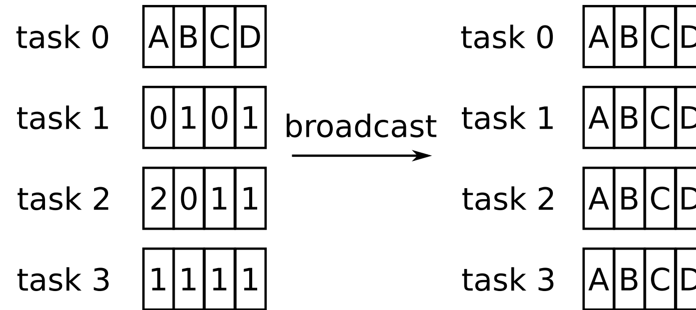


Figure 2.2: Diagram of `broadcast` operation

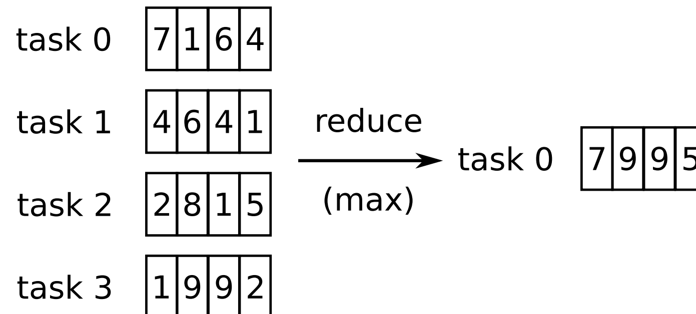


Figure 2.3: Diagram of `reduce` operation

Another collective communication is `reduce`, which performs commutative computation (such as ADD or MAX) on data passed to the operation. `Reduce` provides functions to reduce the dimension of input data by 1. For example, if a set of parallel tasks compute max value of each row in a 2-D matrix, then `reduce` operation will fulfill the job and return the result as a 1-D vector in the root task. In Figure 2.3, before the `reduce`, tasks each has a column of the input data. After the operation, task 0 (the root) has the result, max value of each row.

2.5 Benchmarks

Benchmarks are used to measure the performance of the HPC system. Some benchmarks are specifically designed for certain aspect of the system, such as the floating point operation rate, the IO bandwidth, or the communication latency. Some benchmarks are extracted from scientific applications, emulating the real operations in the HPC system, and measuring the overall performance.

High Performance Linpack Benchmark (HPL) is the standard benchmark used for measuring the performance and ranking the TOP500 HPC systems. The algorithm embedded in HPL is a double precision (64-bit, IEEE-754) dense linear system LU solver. MPI is utilized by HPL to distribute the data, synchronize the tasks and collect the result. By properly setting up the system parameters (problem size, row partition, column partition, etc.), HPL can test the accuracy of the result and measure the performance of the system in FLOPS (floating point operations per second).

The NAS Parallel Benchmark (NPB) is a set of benchmarks developed by NASA. Since NPB is derived from computational fluid dynamics (CFD) applications, it is widely recognized and used to help evaluate the performance of HPCs. The NPB consists of five kernels and three pseudo-applications, each one has several "classes", targeting at different problem size on different HPCs [33, 34].

- EP: An "embarrassingly parallel" kernel. It provides an estimate of the upper achievable limits for floating point performance, i.e., the performance without significant interprocessor communication.
- MG: A simplified multigrid kernel. It requires highly structured long distance communication and tests both short and long distance data communication.
- CG: A conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large sparse symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication employing unstructured matrix vector multiplication.
- FT: A 3-D partial differential equation solution using FFTs. This kernel performs the essence of many spectral codes. It is a rigorous test of long distance communication performance.
- IS: A large integer sort. This kernel performs a sorting operation that is important in "particle method" codes. It tests both integer computation speed and communication performance.

- BT: Solution of multiple, independent systems of nondiagonally-dominant, block tridiagonal equations with a (5 x 5) block size.
- SP: Solution of multiple, independent systems of nondiagonally-dominant, scalar pentadiagonal equations.
- LU: Regular-sparse, block (5 x 5) lower and upper triangular system solution.

2.6 Amdahl's Law

Beyond the benchmarks, characterizing and modeling the parallelism and communication is another active research area. In the HPC world, one of the most well-known and classic theory is called Amdahl's Law [35], which characterized the speedup of parallelizing an application program:

$$Speedup = \frac{1}{(1-F)+\frac{F}{N}}$$

Here F $(0 < F < 1)$ denotes the fraction of the program which can be parallelized, N represents the number of computing unit. Assuming the parallel computing units can achieve N times speedup on the parallel portion, the formula suggests that the maximum performance is limited by the sequential (non-parallel) part of the application. The speedup of using different F is illustrated in Figure 2.4.

2.7 Communication Model

David Culler et al. proposed LogP model [36], which models the communication of modern and future massive parallel processor system. The model is based on four parameters listed below:

L: an upper bound on the latency, or delay, incurred in communicating a message containing a word (or small number of words) from its source module to its target module.

o: the overhead, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor cannot perform other operations.

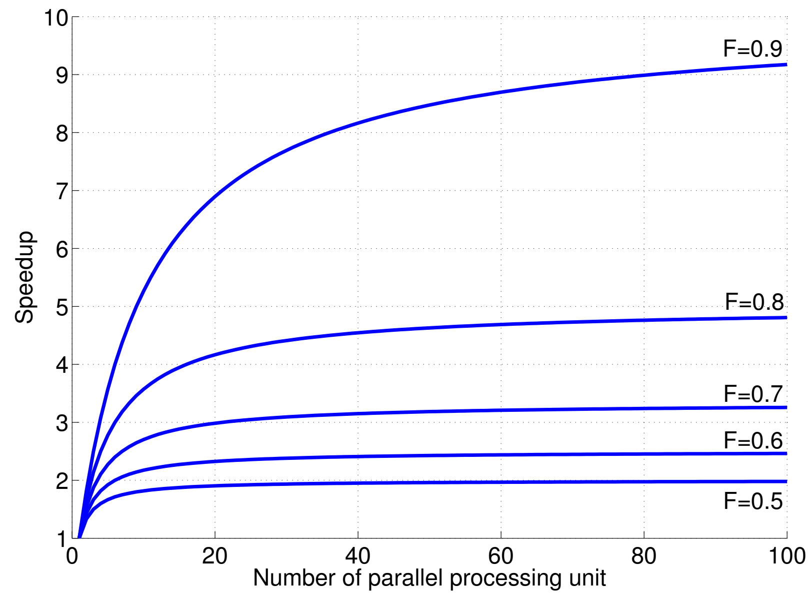


Figure 2.4: Amdahl's Law, performance gain of parallelism

g: the gap, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g corresponds to the available per-processor communication bandwidth.

P: the number of processor/memory modules. We assume unit time for local operations and call it a cycle.

CHAPTER 3: RELATED WORK

3.1 MPI Related Research

Because MPI is the standard interface to program supercomputers, there are countless HPC related research of MPI. Following the specification, users can write applications in MPI and run the applications using different parallel machines. When running the program, the users are free to specify what algorithm to choose and which hardware interconnect to use. These flexible features let the users focus on the functionality of their applications, while let researchers focus on optimizing MPI.

3.1.1 Point-to-point Communication

Point-to-point communications, such as `send` and `receive`, are the fundamental operations in MPI. The MPI specification defined several varieties of `send` and `receive` primitives, each one with different handshaking protocols and different buffering options. The varieties have different performance, meanwhile provide the programmers the freedom to choose the best point-to-point operation based on their needs. Some research has been done to optimize the point-to-point operations. In [37, 38], the researchers leveraged RDMA to develop a set of customized protocols to maximize the performance of point-to-point communication. The TMD-MPI [39] implemented `MPI_Send` and `MPI_Recv` in FPGA.

3.1.2 Collective Communication

Compared to point-to-point operation, collective communication, which involves multiple tasks, has received a lot more research attention, ever since the advent of parallel computing. An interesting profiling research done in [40], studied the behavior of real MPI applications running on state-of-the-art clusters. The statistical results showed that more than 40% of the execution time of all MPI calls are spent on `MPI_`

`Allreduce` and `MPI_Reduce`. To alleviate the heavy load on these two primitives, the author proposed several reduce algorithms optimized for different vector size and number of processes in [41]. By experimenting different parameters on the target machine, a $3 - -100\times$ speedup of `reduce` operation could be achieved.

The work in [42] presented several `barrier` algorithms. With respect to algorithms, one conventional approach is to create a head (or root) node which receives all the barrier messages and distributes the clear messages. Specifically, Central Counter [43] is one algorithm where a counter is kept on one node to track the number of nodes that have reached their barrier. When the counter equals the size of the network, the clear barrier message is issued. The basic implementation of `MPI_Barrier` used within OpenMPI utilizes point-to-point communications to pass barrier messages to and from each node and the head node, which is called Sequential Tree in [44]. Other Tree based barriers such as Combining tree [45, 46] can differ based on the internal tree structure and the decision making process to achieve parallelism in message transmission. Alternatively, Butterfly barrier [47] and Recursive Doubling [48] utilize pairwise message exchange to implement the barrier instead of using a head node to issue the clear barrier decision.

In [49], the authors comprehensively summarized the design and implementation of collective communication on several distributed-memory architectures, which covered the research in the past 30 years. This paper not only summarized the algorithms used to implement the collective communication instances, but also analyzed these algorithms using mathematic models. Based on the commonly used algorithms, Minimum-spanning tree algorithms (MST), Bidirectional exchange algorithms (BDE), and Bucket algorithm, the authors proposed several hybrid algorithms focusing on different message size and architectures. The test results on a Myrinet connected Xeon cluster showed that the hybrid algorithms achieved performance improvement in most situations compared to common implementation of MPI such as MPICH.

The work in [44] summarized the general algorithms for collective communication. By experimenting the algorithms with different parameters (message size, communicator size, user application, etc.), a static tuned collective communication library was obtained. The results were reported to improve the performance by 35% to 650% when compared to native MPI implementation. However, in most cases, the static optimization is tuned for a particular architecture or a specific application. The static optimization requires an extensive test of all the combinations of the parameters, which is not possible when the system scale is large. So mathematical models were used to predict the performance of the algorithms. In [50], the author used Hockney, LogP/LogGP, and PlogP models to analyze the performance of the collective algorithms. Compared to the static tuned library, the prediction of the mathematical models can achieve a near-optimal solution. In [51], quadtree encoding method was used to build run-time decision tree, based on statistical learning. This research showed feasible approach to optimize collective communication in run-time.

3.1.3 Hardware Optimization

Although there is a large body of work related to changing the software optimization (switching the algorithm depending on the size and number tasks participating in the operation), some of the optimization can be applied in hardware as well.

Several algorithms were proposed in [52] for "global combination", which is now `MPI_Allreduce` on a 2-D mesh interconnect with wormhole routing. This paper proved that it is possible and efficient to execute global operations on 2-D mesh interconnect. However, to achieve best performance over the full range of data size, different algorithms should be adopted for different scenarios.

In [53, 54], IBM implemented dedicated networks for Blue Gene/L and Blue Gene/P. The nodes in the system are interconnected via three networks: 3D torus, collective tree network, and global interrupt. The torus network is the main network for point-to-point communication. The collective tree network is capable of providing

low latency and high bandwidth for fan-in and fan-out operations (`broadcast` and `reduce`). The global interrupt provides configurable OR wires to perform hardware-based synchronization. Beyond BG/L, BG/P features DMA to offload messaging work from processors and achieve better communication and computation overlap.

In [55], the researchers explored hardware feature on the Infiniband adapter, ConnectX-2 from Mellanox Technologies. The hardware offloading feature, called CORE-Direct, can offload a series of send, receive and reduction tasks to the adapter. The researchers generalized the collective communication into several primitives and designed these primitives using the hardware feature. The test result showed the designed `MPI_Barrier` (from the primitives) achieved almost perfect overlap of computation and communication and some performance improvement of `Recv-Replicate` primitive.

The PERCS high-speed interconnect developed by IBM [56] features a Hub chip that integrated into the compute node. The Hub chip is used to connect local Power7 chips and interconnect with other compute node. In the Hub chip, there is a Collective Acceleration Unit (CAU) designed to speed up the collective communication, specifically the `barrier`, `multicast`, and `reduction`. The large-scale PERCS installation, Blue Waters is being constructed at NCSA, and it is expected to deliver sustained Petascale performance over a wide range of applications.

Cray Inc. designed Seastar Interconnect [57] and Gemini Interconnect [58] to support high-performance distributed system. The Portals network interface [59, 60] designed by Sandia National Lab can leverage the hardware DMA on the NIC to bypass the OS and offload the `send` and `receive` operations.

As part of the Adaptable Computing Cluster project, [61] implemented `MPI_Reduce` in the FPGA fabric of a Network Interface Card. This has the advantage of using a commodity off-the-shelf interconnect (Gigabit Ethernet, in this case) in a commodity cluster. These ideas were further explored in [62]. Voltaire has recently

announced support for collective communications inside of their InfiniBand switch; however, no peer-reviewed report is available yet to characterize the advantages.

The OSU group studied several collective communication primitives [63, 64, 65, 66, 67] on Myrinet. The research has shown that NIC-based collective operations is able to reduce the host processor involvement, avoid bus traffic and increase the tolerance to process skew and OS effects.

In the work of [68], the authors described an implementation of collective communication with a combination of shared and remote memory access (RMA) protocols. The proposed approaches were tested on IBM SP with LAPI support for RMA, achieved performance improvements in all test configuration.

3.2 On-chip Message-Passing

While the previous section describes many research focusing on the standard MPI implementation and optimization, the message-passing concept is not limited to the software. Some research and developments of on-chip architecture are implementing similar message-passing mechanism.

3.2.1 Raw

The Raw Architecture Workstation (Raw) is a tiled multicore architecture that explores the fine-grain parallelism between many replicated processing elements [69]. The key feature of Raw is that the hardware architecture is exposed to the programmers, so the compilers or application designers are required to choose the correct tile and program the routing between the tiles.

The prototype of Raw processor is a 4×4 tile structure. The processor core in each tile is a 8 stage MIPS processor along with local memory and the cache. The on-chip network between the tiles consists of a static network and a dynamic network. The static network is used for passing operands and data streams within or between the tiles. While the dynamic network provides DMA or message passing. Relatively speaking, The dynamic network has lower performance than the static network. Some

researchers utilized the Raw processor in their application and achieved considerable performance gain [70].

3.2.2 Intel Terascale Computing

As the leading manufacturer in the industry, Intel has several research and experimental projects shooting at the future generation processor and computer systems. Based on current trend of multicore, Intel has envisioned the future processor to have 100s cores on a single chip. More importantly, the visioned architecture would rely heavily on the on-chip network, advanced power management technologies and support for “message-passing”.

One prototype project implemented 80 simple cores on a single chip [71]. Each core has a message passing router that is connected as a 2D mesh network that allow message-passing communication. Another 48-core architecture “Single-chip Cloud Computer” is built as an experimental processor that resembles a cluster of computers [72]. Besides the components that are common in x86 system, the designers build SRAMs with each computation tile, called message passing buffer (MPB), which is able to provide fast communication between cores via messages.

3.2.3 RAMP

RAMP is the acronym for “Research Accelerator for Multiple Processors”, which is a group of research projects originated from UC Berkeley [73]. The goal of RAMP project is to utilize the FPGA as a hardware instrument to prototype, simulate future computer system, programming languages and other tools. Several prototype machines were built for different research purposes.

The RAMP-Red is a multiprocessor system with hardware support for transactional memory. On the development board, multiple processors are connected to a shared memory via a switch. Custom cache is designed to support transactional memory. This design is 100 times faster than the software simulation [74].

The RAMP-Blue is a manycore message-passing architecture. 1008 Microblaze

cores are connected with a custom network. The designers choose uClinux as the OS. With UPC framework and GASNet to support message-passing, the system is able to run NAS Parral Benchmark [75].

3.2.4 Reconfigurable Computing Cluster

The Reconfigurable Computing Cluster (RCC) project is investigating the feasibility of cost-effective Petascale clusters of FPGAs [76]. A prototype machine is built with 64 Xilinx ML-410 development board.

The network design is the critical component within the RCC project. The initial design includes a custom high-speed network card, which utilizes the RocketIO and Aurora cores from Xilinx [77]. The custom network, AIREN (Architecture Independent REconfigurable Network), aggregate both the single FPGA network-on-chip and multiple-FPGA networks. The bit rate of this high-speed network is measured 3.2 Gb/s per channel. There are 8 channels on each network card, so different topologies can be built around the hardware. For the researchers' test, the network can be arranged in Torus structure or a Ring network. DMA engine can be built into the network to fulfill the point-to-point communication. For each transfer, the latency between the neighbor nodes is 0.8μs. With this custom high-speed network, the researchers can have multiple hardware accelerators executing in parallel, obtaining linear speedup, from $5.0\times$ to $20.92\times$ [78].

CHAPTER 4: DESIGN

Given the fact that communication frequency and data size will rise with the increasing number of PEs and growing size of the problem, the processor hosting the OS can be overloaded by the heavy communication, and therefore become the bottleneck of the system. In order to solve this bottleneck, the proposed solution is to design a dedicated Message-Passing Engine (MPE) in hardware to handle the communications, especially collective communications. The design is split into two stages. Stage 1 will focus on using hardware to implement the message-passing function and offload some software MPI operations in a homogeneous system. In stage 2, the design integrates the MPE in the heterogeneous system, in which the hardware MPE will provide communication for different types of processing elements in the system.

4.1 Design Infrastructure

Spirit Cluster, described in Section 3.2.4, is used as the design infrastructure to implement and evaluate the proposed work. *Spirit* is a cluster of 64 ML-410 FPGA development boards. Each development board has a Xilinx Virtex 4 FX60 FPGA. A high-speed network card has been designed to route 8 high-speed transceiver ports off the board.

4.1.1 Off-chip Network

With the designed custom high-speed network card [77], the development boards can be arranged as a directly connected network, in which each node (FPGA) has a local router with a unique network ID. To route the packets between indirectly connected nodes, it requires the router on the intermediate node to route the packets through. The example shown in Figure 4.1 is a connection of 8 independent systems

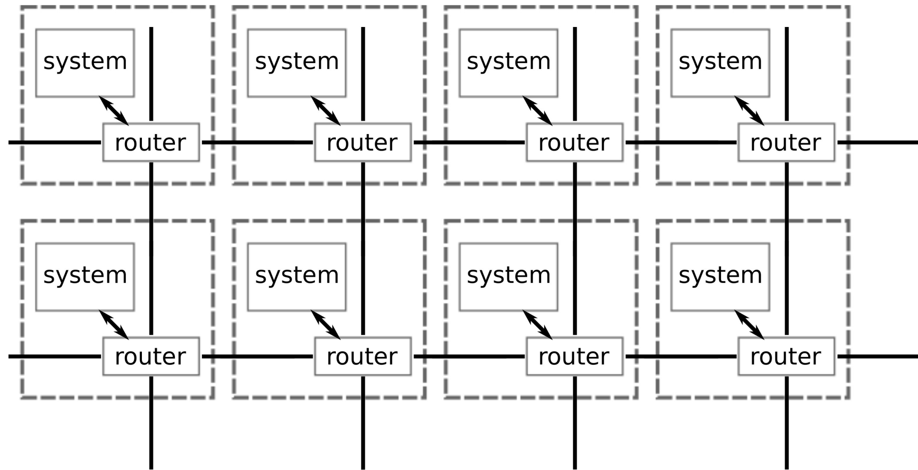


Figure 4.1: Direct connected off-chip network with the router

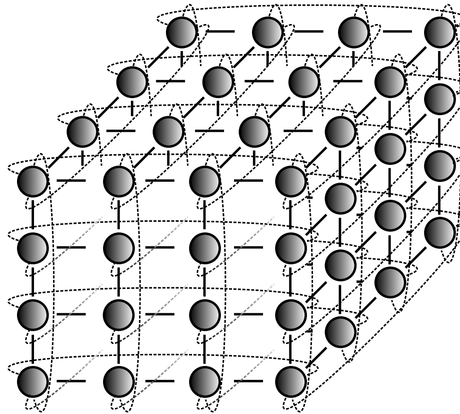


Figure 4.2: 4-ary 3-cube torus network

via the local router. Figure 4.2 shows a 4-ary 3-cube Torus network, which is the current implementation on *Spirit*. On each node, 6 out of 8 ports are used. Each port is connected to neighbor nodes in X+, X-, Y+, Y-, Z+, and Z- directions. Different routing algorithms can be applied to the off-chip network, such as dimensional routing and adaptive routing [79].

4.1.2 On-chip Network

The on-chip network is designed to provide communications between local components. At the same time, it allows on-chip components communicate with the off-chip network. Several on-chip interconnect methods can be used, such as ring, mesh, or

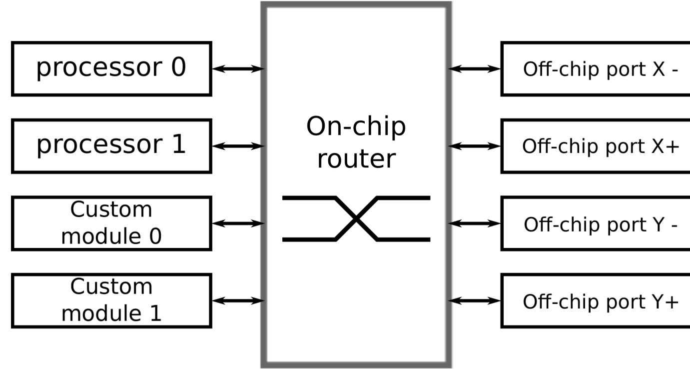


Figure 4.3: On-chip components connected around the crossbar switch

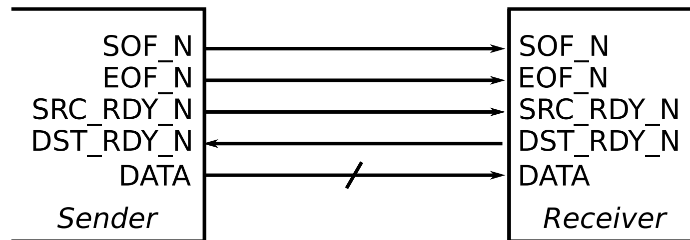


Figure 4.4: Signal interface of LocalLink

star. In previous work, a 16-port crossbar switch and routing module have been implemented, this proposed design will leverage the router and connect the on-chip components around the router, as shown in Figure 4.3.

4.1.3 Network Interface

The network interface is designed to provide a unified view to handle communication between different hierarchical components. As shown in Figure 4.4, a standard network interface — LocalLink from Xilinx [80] — is used in this design. The simple signal interface of LocalLink provides an efficient handshaking mechanism for communications, especially for streams of data. Other network interfacing protocol can be used as well.

4.1.4 Base System

The base system provides a platform for both Stage 1 and Stage 2 designs. In order to support a message passing environment, a traditional processor-bus-memory

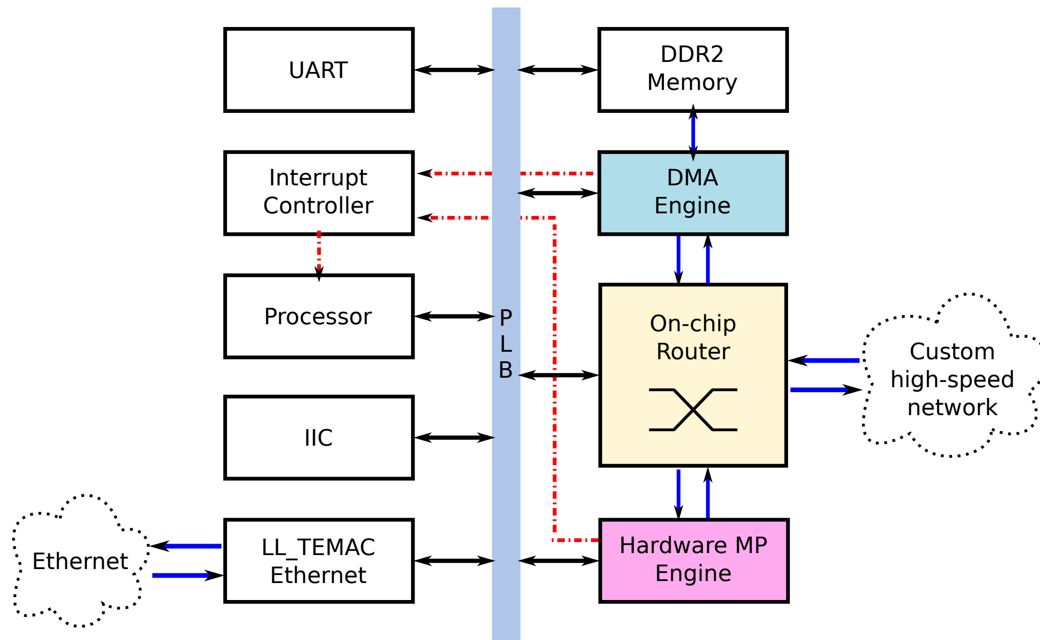


Figure 4.5: Hardware base system with the on-chip router

architecture is adopted. Using *Spirit* cluster as the infrastructure, the base system is built within the platform FPGA. Note that the low frequency embedded processor within the platform FPGA is obviously not suitable for HPC. Embedded processor is used because there is no discrete processor on the development board. However, the idea of using FPGAs to offload message-passing operations can be applied to general discrete processor systems or modern heterogeneous systems as well.

Within the FPGA, Xilinx has already provided two diffused embedded PowerPC processors. Using Xilinx tools, the Microblaze processor, a soft IP from Xilinx can also be used. The PowerPC has a higher executing frequency, whereas the Microblaze is more configurable and can be easily expanded to multiple cores on a single chip. Shown in Figure 4.5, the system bus is Processor Local Bus (PLB). The peripherals, including DDR2 memory, UART, interrupt controller, IIC, and LL_TEMAC Ethernet, are connected to the PLB. The on-chip router provides the connections for both on-chip components and off-chip system. The router also has a bus connection, which is used for setting control registers and the network ID.

4.1.5 Miscellaneous IP Cores

In order to test the functionality and measure the performance of the MPE, some supplementary hardware IPs are needed. One of the IP is called the source/sink core, which shares the same communication interface as the MPE. The source/sink core can be used as a test core connected to the on-chip router. Controlled by the processor, they can behave like any processor or hardware accelerator sending and receiving the data stream. With the source/sink core, we can test the function correctness of the MPE in the simulation or in hardware.

Another type of the IP is called the monitor core. This hardware was mentioned in [81]. In this proposed work, the monitor core is a collection of hardware counters that count the clock cycles of various operations. With the monitor core, accurate measurement can be obtained.

4.2 Stage 1: Hardware Message-Passing Engine

In Stage 1, the design is to implement the communication functions in hardware. The hardware MPE is acting like a co-processor offloading software operations from the CPU. All the ML-410 development boards are presenting the same hardware configuration.

4.2.1 Point-to-point Communication

Point-to-point communication is the basic operation, which semantically transfers one chunk of data from the sender to the receiver. In standard specification, the variants of software `send` and `receive` incorporates different buffering options, which is not necessary in hardware. To implement the point-to-point communication in hardware, while eliminating the involvement of the processors as much as possible, a hardware DMA engine is connected to the router, as shown in Figure 4.5. When a `send` request is requested, the processor passes the address and the length of the data to the DMA. The DMA will fetch the data directly from the DDR2 memory, assemble the packet and push the packet directly into the custom high-speed network. When

the data packet arrives at the receiver, the DMA engine will trigger a interrupt to notify the processor.

Similar to the software implementation, some collective communications can be easily setup using just `send` and `receive`, for example, `broadcast`, `scatter`, and `gather`.

4.2.2 Collective Communication

After studying the typical implementations of MPI collective primitives, clearly the most time consuming portion of MPI collective communications is the sequential sending and receiving of messages. Some optimizations are able to explore the parallelism within the algorithms, so that some tasks can work in parallel based on certain topologies, as mentioned in chapter 3. However, software overhead such as OS protocol stacks, ISRs, and interfacing with the network are still sequentially executed on general processors, which occupies the processor and limit the computation capability. To handle collective communications in the hardware, the MPE is designed to connect to the on-chip router. Inside the MPE, three typical operations `barrier`, `broadcast`, `reduce` are implemented.

4.2.2.1 Barrier Function

The hardware MPE implements `barrier` function, shown in Figure 4.6, with the goal to move barrier synchronization responsibilities from OS and libraries into hardware. When a `barrier` request is initiated, the processor asserts one bit in the hardware and wait for the `barrier` clear interrupt from the hardware MPE. The `barrier` message is assembled, sent and received completely in hardware. When all the tasks reach the barrier, the hardware send out interrupt signals to the processor.

4.2.2.2 Broadcast Function

The function of `broadcast` is distributing the same chunk of the data from the source task to other tasks. Since data movement could not perform well through the processor-bus combination because of the slow bus transaction, the hardware

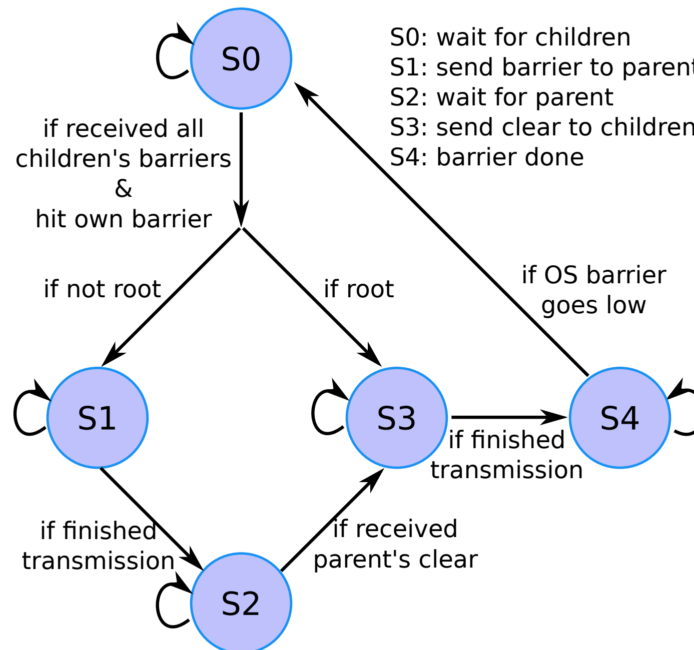


Figure 4.6: FSM of `barrier` operation

DMA engine used for point-to-point communication can also be used in `broadcast` to speedup the data operation without involving the processor and the bus. When a `broadcast` request is issued by the processor, the DMA engine fetches the data and pass the data to the hardware MPE. The MPE assembles the messages, handles the handshaking messages between the parent and children, send and receive data, and notifies the processor by interrupt when the `broadcast` request is done.

Because `broadcast` primitive operates on a vector of data. A FIFO is used as the buffer to hold the data. Because of the resource on the FPGA, the size of the FIFO is limited, which means if the data size is larger than the FIFO size, the data is divided into chunks and transmitted separately.

4.2.2.3 Reduce Function

Besides the similar but reverse data movement as the `broadcast`, the unique feature of `reduce` is that it involves a commutative and associative computation operation. The normal implementation of `reduce`, e.g. `MPI_Reduce` in OpenMPI, is that all the nodes send data to the root node. The root node receives the data in

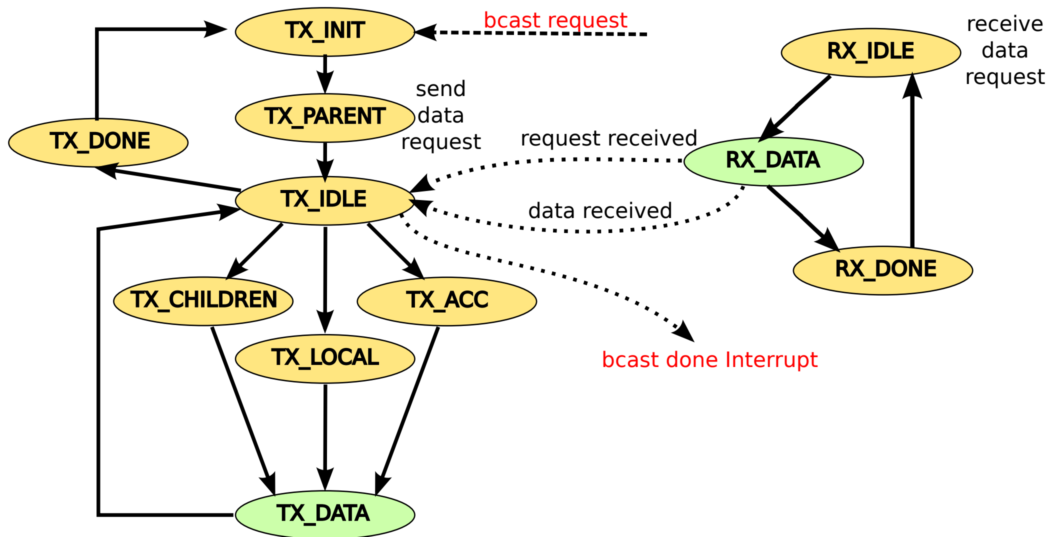


Figure 4.7: FSM of `broadcast` operation

sequence and compute the result in sequence till all the data are consumed. On one hand, the data communication is congested at the root node; on the other hand, the computation is also serialized on the root node. So the overall performance is limited by the performance of the ALU on the root node. Some modern processors have very complex pipeline design to speedup the computation. But the processor embedded in the FPGA has a low clock frequency. What makes it worse is that the embedded processor does not have an usable hardware FPU. It usually takes tens to hundreds of clock cycles to execute one floating-point operation in software, while just a few cycles to execute in hardware. Therefore, in this design, a hardware computation unit is adopted inside the `reduce` core.

4.2.2.4 Topology

As described in Chapter 3, the underlying communication topology — algorithms — sometimes plays an important role affecting the performance of certain communications. Tree-based algorithms have the advantage of low algorithm complexity and overall scalability. The proposed work will design and test some of the popular tree-based topologies.

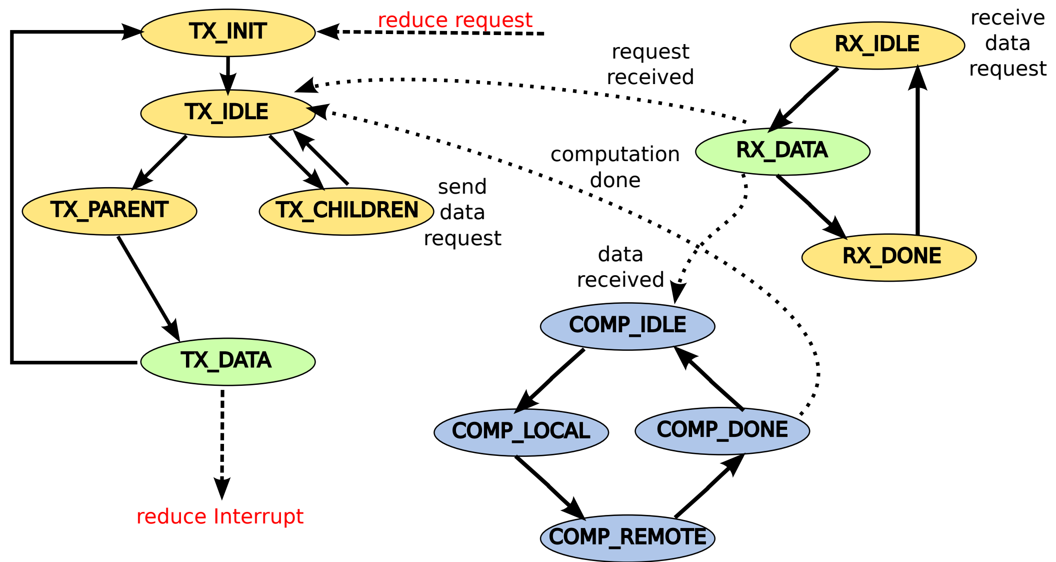


Figure 4.8: FSM of `reduce` operation

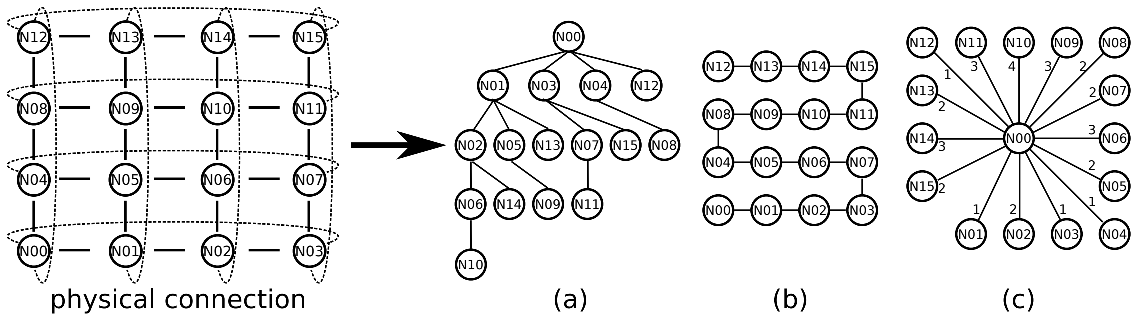


Figure 4.9: Topologies of hardware collective communication

Binomial tree structures utilize all the possible physical channels. Take 4-ary 2-cube as an example, shown in Figure 4.9a, each node has 4 neighbor nodes directly connected. Theoretically, message transmissions can happen in all the channels in parallel, which could achieve the highest topology parallelism.

Linear tree has no topology parallelism, as shown in Figure 4.9b. Every node has only one parent and one child directly connected. Messages are relayed one node to another from the leaves to the root. For simple collective operations, such as barrier, this structure is inefficient because there is no parallelism; however, for `broadcast` and `reduce`, which have multi-stage data operations, this topology creates a pipeline, which could achieve higher bandwidth than other topologies do.

Star tree structure virtually connects the root node to all the other nodes. Physical channels are reused. Messages hop through multiple nodes to the destination via the on-chip router. The number labeled in Figure 4.9c shows the number of hops for each virtual connection. This topology requires only one hardware MPE in the root node, while virtually exploring the maximum parallelism. However, when the size of the message and number of nodes is large, the sole MPE and the number of physical channels on the root node become the limiting factors that would cause contention and degrade the overall performance of the system.

4.3 Stage 2: Heterogeneous System

In Stage 2, the design is concentrating on heterogeneous systems. Instead of being the co-processor of the general processor, the hardware MPE is accessible to all the on-chip heterogeneous PEs. Heterogeneous PEs can communicate directly through the hardware MPE without involving the processor, as shown in Figure 4.10.

Figure 4.10 shows the common configuration of current heterogeneous systems. The general processor is a multi/many-core chip with its on-chip network and the connection to the main memory. The heterogeneous chip is also a multi/many-core chip with the link to the local memory. Between these two packages, high speed

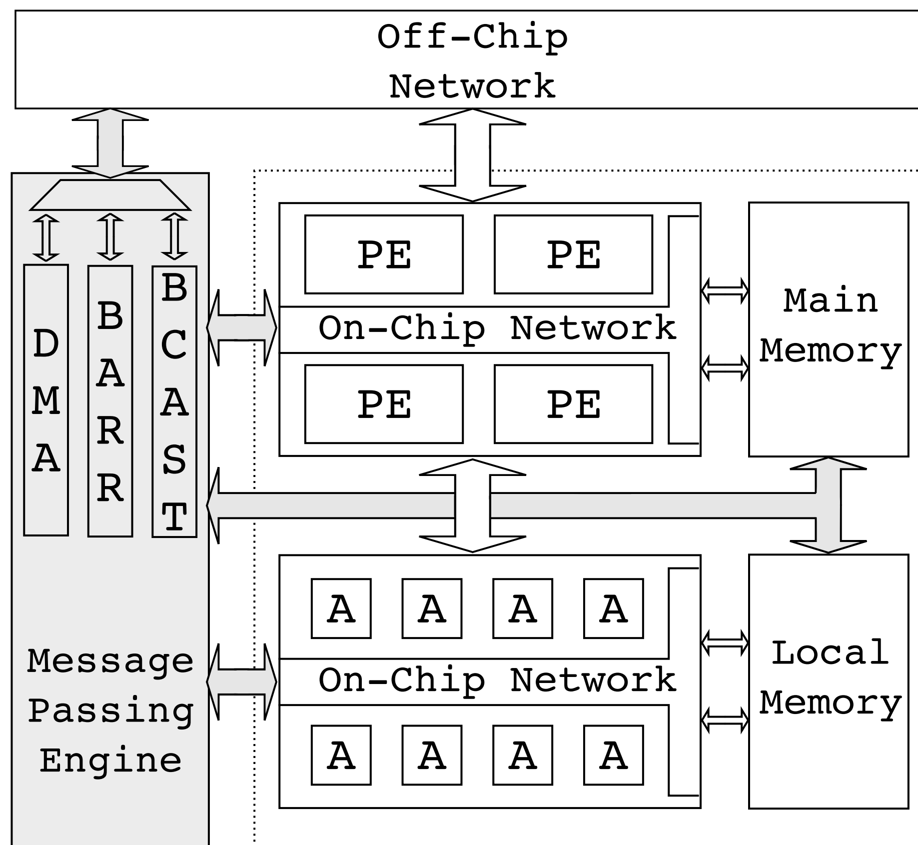


Figure 4.10: Block diagram of MPE in heterogeneous system

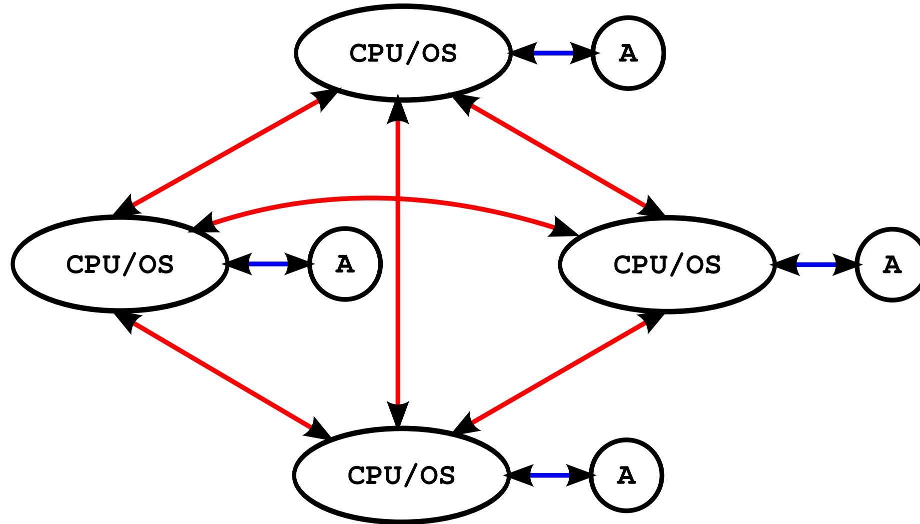


Figure 4.11: Typical programming method for parallel heterogeneous system

connections such as *PCI Express* are often used. In some configurations, the two chips can be manufactured in a single package [82].

The typical programming method for parallel heterogeneous system is relying on the OS and libraries running on the general processor. Figure 4.11 is a example of programming 4 parallel tasks. Each task is running on a general processor with OS and essential libraries. The initial data is distributed by the standard MPI function calls and stored in the main memory. As each task finishes receiving the data, the OS and the libraries assign subtasks to the heterogeneous PEs and copy the data from main memory to the local memory associated with the heterogeneous PEs. Then the heterogeneous PEs may start the computation. After the computation is finished, the OS and libraries copy the data back into the main memory. At last the general processor may process the following program.

As we can see in Figure 4.11, data are frequently transferred back and forth between the main memory and heterogeneous PEs' local memory. This has two negative impacts: First, as the OS and libraries are running on the general processor, frequent communication requests may overload the general processor. Second, the

communication requests need to pass multiple software stacks, these operations are trivial but consume the clock cycles which can be used in real computation.

To address these two negative impacts, the hardware MPE can be used, as shown shaded area in Figure 4.10. First, the hardware MPE is dedicated to communications, it can route communications directly to heterogeneous PEs without going into the main memory or overloading the general processor. Second, the hardware MPE process the communication in parallel with the general processor, that gives the general processor opportunity to process other computation.

4.3.1 Parallel FFT Operation

Fourier Transform is a transformation of one sequence of signal to another sequence of signal. Generally, forward transformation transforms time-domain signals to frequency-domain signals, whereas inverse transformation transforms signals vice versa. The commonly used Fourier Transform is Discrete Fourier Transform (DFT), which involves heavy computations on floating-point multiplication and addition. Because of the periodical characteristics of the twiddle factor and the finite sequence, Fast Fourier Transform (FFT) algorithm can calculate DFT using less computations with intermediate variables reused. One well-known FFT algorithm is the Cooley–Tukey algorithm, it recursively divides the sequence into two halves, which can be expressed as smaller FFT. Two types of decimation strategies can be used to implement FFT algorithms: Decimation-In-Time (DIT) or Decimation-In-Frequency (DIF). As shown in Figure 4.12, the DIT algorithm requires a bit reversal sorting operation performed on the input data, and the output data is in natural sequence. Figure 4.13 shows that the DIF algorithm is able to take natural sequence directly as input, and output the result in a bit reversal style.

4.3.1.1 Algorithm

In this work, a parallel FFT operation is implemented in both the software and the hardware. With the goal to stream the natural sequence data into the hardware,

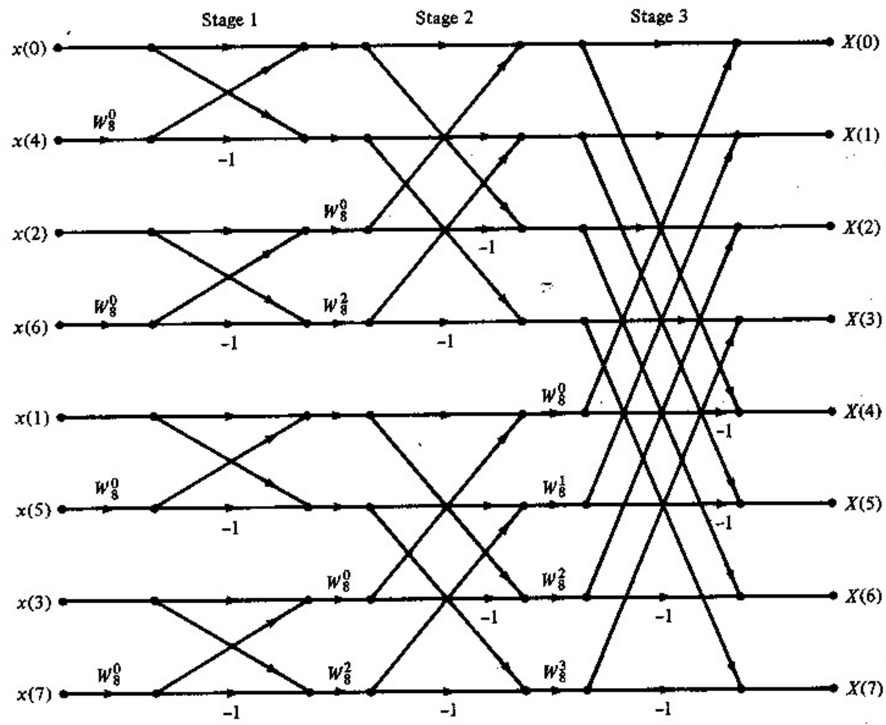


Figure 4.12: FFT Decimation-In-Time

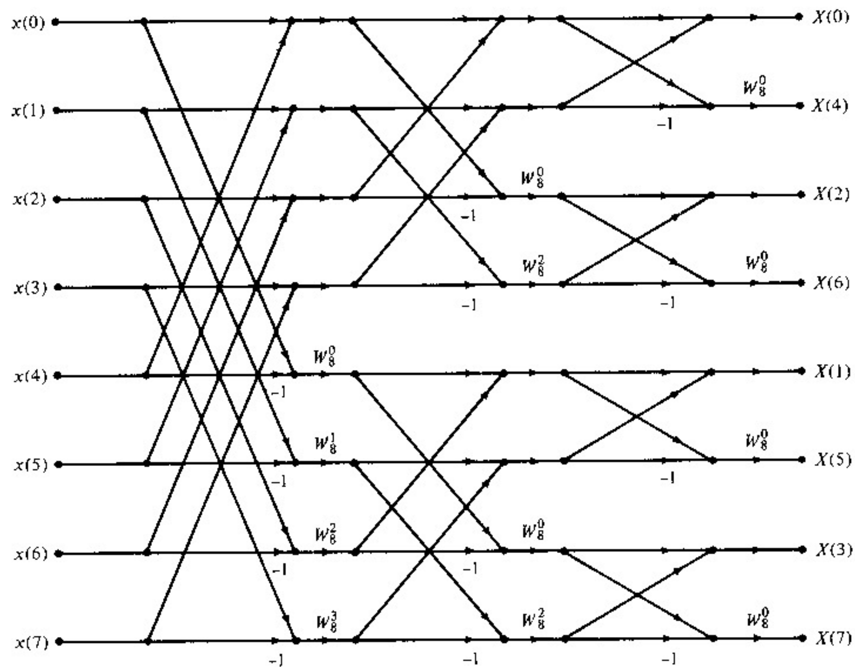


Figure 4.13: FFT Decimation-In-Frequency

DIF algorithm is implemented. The parallel FFT DIF algorithm involves two steps: an inter-node FFT DIF step and an intra-node FFT DIF step. The inter-node FFT DIF requires point-to-point communication, whereas the intra-node FFT DIF does not have communication. The parallel FFT DIF algorithm is described below:

1. Before the computation starts, twiddle factors are calculated and stored in the memory.
2. The root node generates the original data. A `scatter` operation is issued and distribute the original data to all the other nodes. The received data is used as the local data.
3. Based on how many nodes are involved, every node calculates its remote node. A point-to-point communication is initiated on each node, sending local data to remote node. The received data is treated as the remote data.
4. After every node has the local data and the remote data, an inter-node FFT DIF calculation is performed. The results are stored in the local data.
5. Check if the inter-node calculation is finished ($log_2(n)$). If yes, the intra-node FFT DIF will be performed on the local data. If not, loop to 3.

4.3.1.2 Implementation

Figure 4.14 shows the block diagram of the designed hardware FFT core. The FFT core takes three complex inputs, `cplex_a`, `cplex_b`, and `cplex_t`, which correspondingly represent the local data, the remote data, and the twiddle factor from the table. The `cplex_addsub` block instantiates 2 floating-point add/sub units for the real part and the image part of a complex number. Within the `cplex_mul` block it instantiates 4 floating-point multiplication units and 2 floating-point add/sub units. The `cplex_sreg` is a shift register designed to synchronous the table input to the add/sub input, providing exact same clock delay as the `cplex_addsub`. Based on the control signal, `cplex_addsub` adds two inputs or subtracts `cplex_b` from `cplex_a`. The output of `cplex_addsub` is feed into `cplex_mul` and multiply with the syn-

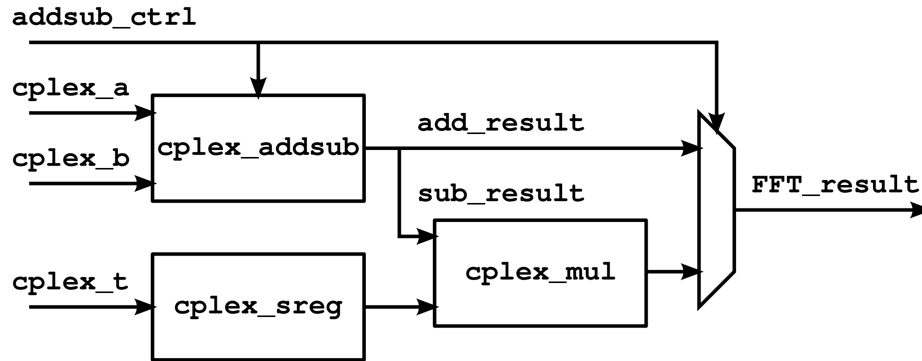


Figure 4.14: Block diagram of FFT Core

chronous table input from `cplex_sreg`. Based on the control signal, the FFT core outputs result either from the `cplex_addsub`, or from the `cplex_mul`. Figure 4.15 illustrates the upper IO level of the FFT core. Two FIFOs are used, one is used to store the local data, and the other is used to store the remote data. The `FFT_TABLE` instantiates a two-port BRAM primitives to store the twiddle factor. One port of the BRAM (`BRAM_PORT_A`) is connected to the bus, from which the PowerPC can calculate the twiddle factors and writes into the BRAM. The other port of the BRAM (`BRAM_PORT_B`) is connected to the FFT core. As both local data and remote data are ready in the FIFOs, the FSM asserts read signals to both FIFOs as well as the BRAM. When the calculated results are pipelined out of the FFT core, they are feed back into the local FIFO. When the calculation is finished, the FSM asserts read signal to local FIFO and assembles a transmission to the remote node. At the same time, the FSM receives remote data and stores it in the remote FIFO. When the inter-node FFT DIF is completed, the data in local FIFO can be dumped into the main memory or another hardware core. In this work, due to the limitation of the hardware resources, the intra-node computation is carried out on PowerPC.

4.3.2 Parallel Matrix-Vector Multiplication

In scientific applications, floating-point matrix calculation is widely used and it is often considered important performance index. Benchmarks, such as HPL, use

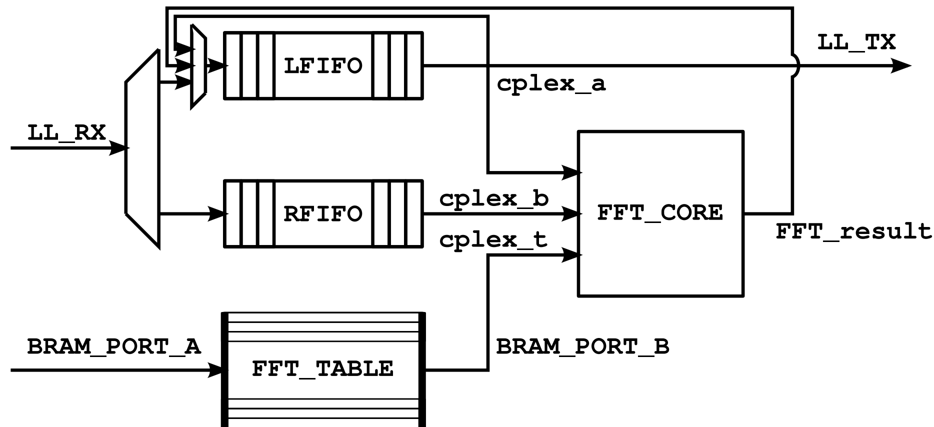


Figure 4.15: Block diagram of FFT IO

matrix calculation as the kernel calculation. Therefore, in this experiment, a floating point matrix-vector multiplication is implemented, both in the FPGA fabric and the software.

4.3.2.1 Algorithm

There are many parallel algorithms for matrix-vector multiplication. In this experiment, a row-based parallel algorithm is designed as follows:

1. Root node generate matrix A and vector B.
2. Root `scatter` matrix A in row order to all the nodes in this operation. All the nodes have partial matrix A.
3. Root `broadcast` vector B to all the nodes in this operation. All the nodes have vector B.
4. All the nodes calculate partial result vector C using partial matrix A and vector B.
5. Root `gather` partial result vector C from all the nodes and combine it into result vector C.
6. Optional: loop

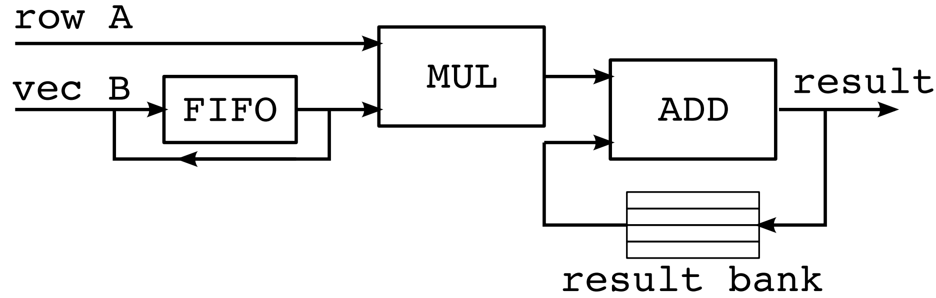


Figure 4.16: Block diagram of vector-vector multiplication

4.3.2.2 Implementation

Based on the algorithm, the matrix-vector multiplication can be broken into several vector-vector multiply-accumulate operations. Consider this multiply-accumulate operation as a stand-alone unit, two implementations are designed: the hardware MACC core, and the software MACC kernel.

The hardware MACC core is implemented in the FPGA fabric, utilizing DSP slices and block RAMs. As shown in Figure 4.16, the MACC core is designed with one FIFO, one floating-point multiplication core, and one floating-point adder core. The computation essentially involves several data streaming operations. Vector B is distributed from the root node and stored in the FIFO in all the MACC cores via the `broadcast` operation. Partial matrix A is streamed in the MACC in row order through the `scatter` operation, the FIFO synchronously pops the data and feeds the data to the floating-point multiplication core. At the same time, the output data is pushed back into the FIFO and ready for next row of partial matrix A. The register bank is used to temporarily buffer the results from the pipeline delay of the adder core. All the hardware primitives are generated using Coregen from Xilinx tools.

The software MACC kernel is simply implemented as a for-loop. On one hand, the software MACC kernel can be used as a reference for the hardware MACC core. On the other hand, a hybrid computing system can be implemented by utilizing the software MACC kernel and hardware MACC core in parallel. The total workload

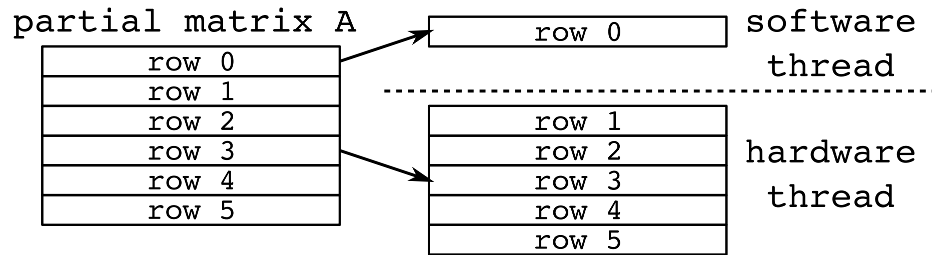


Figure 4.17: Hybrid of hardware thread and software thread

can be distributed to software and hardware at the same time. To leverage both the heterogeneous hardware and software configuration, *Pthreads* can be used, as shown in Figure 4.17.

CHAPTER 5: EVALUATION AND ANALYSIS

5.1 Evaluation Infrastructure

As described in Chapter 4, the hardware MPE design and testing infrastructure are implemented on *Spirit* cluster. The detailed specification of Xilinx ML-410 development board can be referenced in [83]. For reference, *Python* cluster, which is mentioned in Section 1.3, is used as the commodity HPC system to run the reference software tests.

5.2 Testing Methodology

A synthetic benchmark is written in C to measure the execution time of the communication primitives. The processor writes to registers to set the network ID and the communication topology before the collective communication occurs. By measuring the time for a certain number of communication calls to complete, the average execution time can be calculated for each node. The measurements will test configurations of different number of nodes and vary the problem size for `reduce` and `broadcast`.

In order to run the synthetic benchmark under Linux, custom device drivers are required to support control between the hardware and the software. The device drivers issue the network IDs to the MPE based on the node's IP address. During initialization, the application writes pre-calculated tree topology to the hardware MPE. When `reduce` or `broadcast` function call occurs, the device drivers initiate the memory operation from the DMA engine and waits for the completion interrupt from hardware. To avoid overfilling the hardware FIFOs, the device drivers calculate the length of each message, and divide long message into small messages which fit in the FIFOs. Then the device drivers issue consecutive requests to the hardware MPE.

The synthetic benchmark can be ported to use the standard software MPI. As a reference, the ported benchmark can be executed in the native Linux on FPGA, or on the commodity HPC system, such as *Python* cluster.

5.3 Stage 1 Experiment

The Stage 1 experiments measure the performance (latency and bandwidth) of the design MPE using different communication topology, specifically the Binomial Tree, the Star Tree, and the Linear Tree. Other user-defined topology such as Binary Tree is also tested. Because the Binary Tree does not show distinctive result, it is not reported. Same experiments are exercised on *Spirit* and *Python* cluster using the traditional software MPI.

5.3.1 Barrier Performance Result

Due to the nature of `barrier` — one task cannot hit next `barrier` while other tasks are still processing current `barrier` — it does not involve any pipelined operation, which means the measured results illustrate the operation latency.

Figure 5.1 shows the result of MPE `barrier` operation. It can be seen all three topologies show $2\times$ increase in latency as the number of nodes doubles. Linear tree performs worst among all three topologies. Binomial Tree and Star Tree have very close results. The results show that increasing dimensionality of the communication topology can effectively reduce the communication latency. Reusing channels in Star Tree topology does not cause congestion because the communication payload of `barrier` is small.

Figure 5.2 illustrates the traditional software MPI `barrier` on *Spirit* cluster and *Python* cluster. It can be observed that `barrier` shows quite a large latency on *Spirit* cluster due to the slow clock rate of the processor and peripherals. With a much advanced hardware architecture, *Python* cluster is able to achieve latency as low as 12μs.

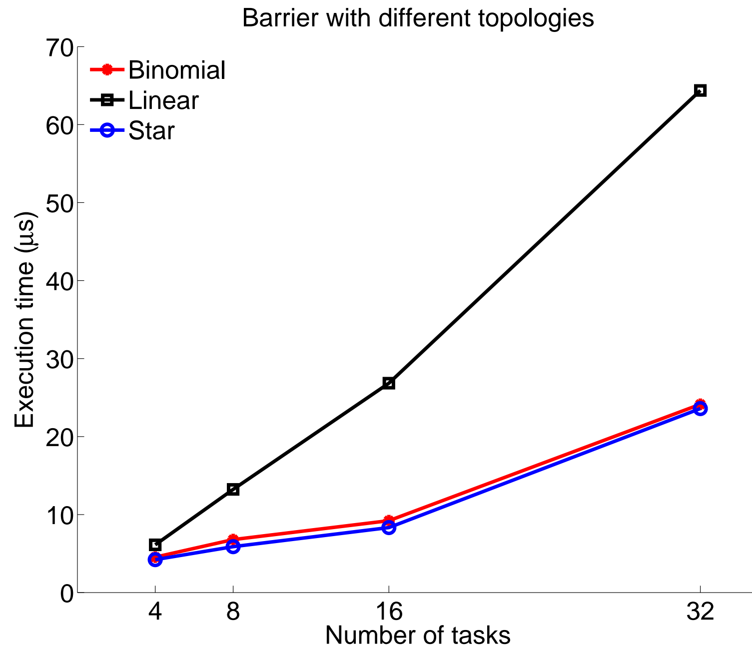
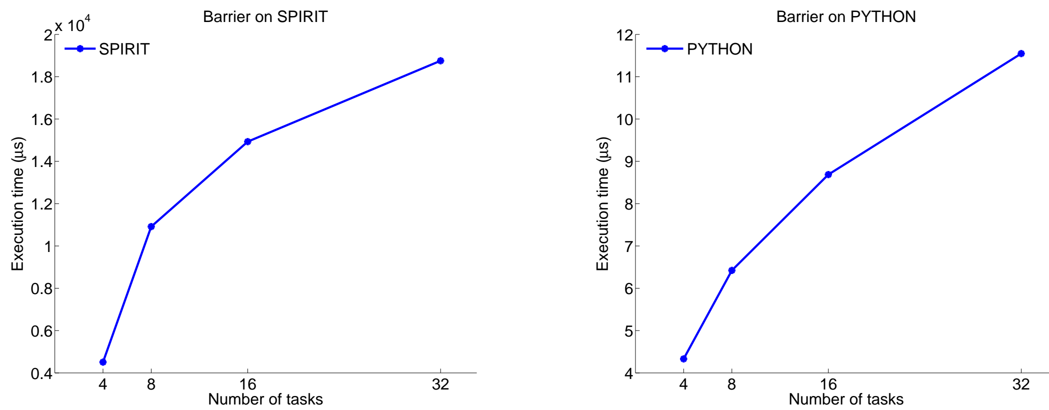


Figure 5.1: MPE `barrier` using different topologies



(a) *Spirit* cluster (b) *Python* cluster

Figure 5.2: Software `barrier`

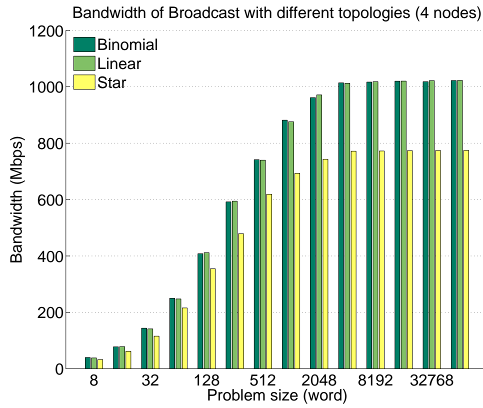
5.3.2 Broadcast Performance Result

Broadcast operation distributes data from the root task to all the other tasks. In the repeating synthetic benchmark, this unidirectional communication pattern can establish a pipelined structure — one task can start a new `broadcast` request, as long as this task finishes broadcasting to all the children. At the same time, other tasks down the line can be processing previous requests. This pipelined structure can effectively increase the bandwidth of the communication. To measure the latency, a hardware MPE `barrier` is inserted between the repeating `broadcast` requests. Then the hardware `barrier` time is subtracted from the measured results.

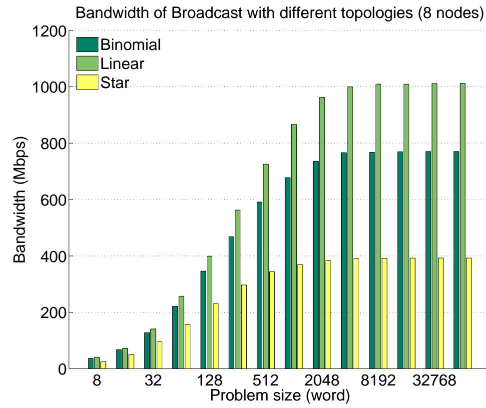
5.3.2.1 Bandwidth

Figure 5.3 presents bandwidth results of MPE `broadcast`. The bandwidth is calculated using the communication payload divided by the execution time (without `barrier` inserted). It can be seen in all the tests that when the communication payload is small, the payload cannot fully utilize the bandwidth. The bandwidth gradually rises as the payload size increases. As the payload size surpass the buffer size (4096-word), the bandwidth saturates.

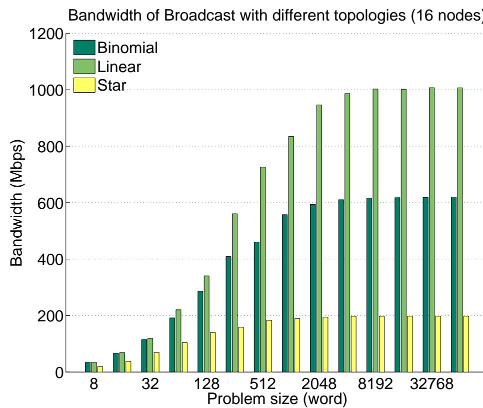
Compare all three topologies, it shows Linear Tree has the highest bandwidth, because the pipelined communication topology can have multiple `broadcast` requests on the fly. As the number of nodes increases, the bandwidth does not degrade. Star Tree has the lowest bandwidth, because it relies solely on the root node to send the data. During one transaction, all the rest nodes wait for the data and no communication parallelism can be achieved. Additionally, as more nodes are involved in the communication, the bandwidth degrades even more. Binomial Tree essentially combines the Linear Tree and the Star Tree. For communication between different topology levels, it features a pipelined structure. For nodes within the same topology level, it relies on the upper node to distribute the data, therefore making the upper node the communication bottleneck.



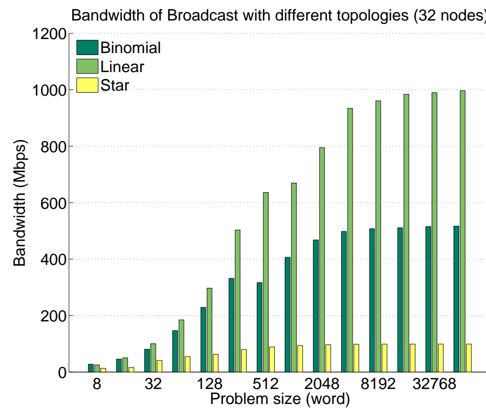
(a) 4 nodes



(b) 8 nodes

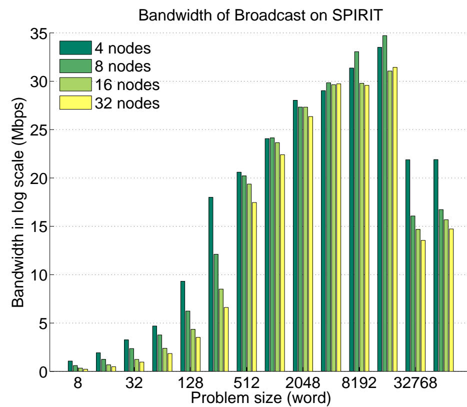


(c) 16 nodes

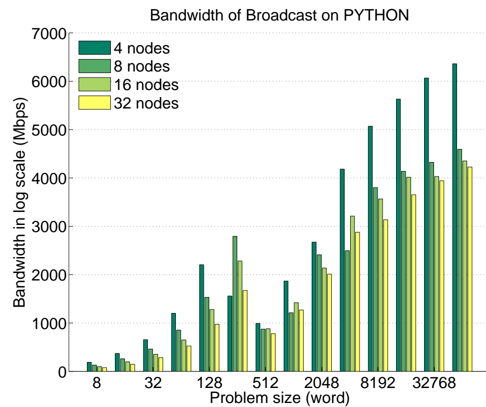


(d) 32 nodes

Figure 5.3: Bandwidth of different `broadcast` topologies



(a) *Spirit*



(b) *Python*

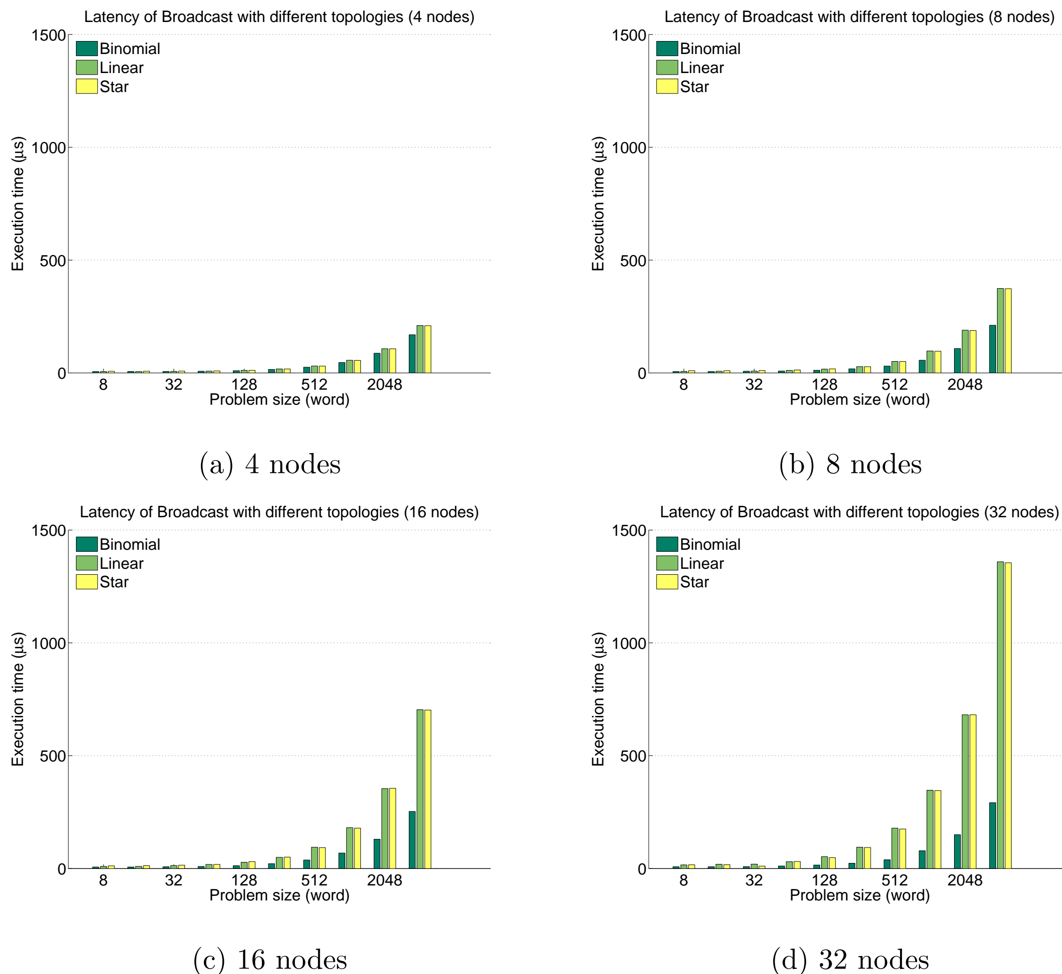
Figure 5.4: Bandwidth of software `broadcast`

Figure 5.4 exhibits the bandwidth results of software `broadcast` on *Spirit* and *Python*. Because of the 300 MHz clock rate and relatively slow Fast Ethernet (100 Mbps), *Spirit* shows bandwidth less than 35 Mbps. With a more advanced processor and system interconnect, the bandwidth of `broadcast` operation on *Python* is able to reach more than 4.0 Gbps.

5.3.2.2 Latency

Figure 5.5 shows latency results of MPE `broadcast`. Because FIFO of 4096-word is used as the buffer in the hardware, these figures only report results less than 4096-word. For problem size larger than 4096-word, data is divided into multiple 4096-word transactions, and the result is simply the corresponding multiple of 4096-word result. It can be observed that due to the long chain topology, Linear Tree has the largest latency in almost all the test cases. Star Tree shows interesting results. As the number of node is small, Star Tree performs well because of the parallelism from the topology. However, as the number of node increases, the performance of Star Tree degrades very fast, this is because Star Tree overly reuse the physical channels on the root node, which causes congestion on the root node. As the number of node approaches to 32, the performance of Star Tree is almost as bad as the Linear Tree. Binomial Tree performs the best among all the topologies, because the parallel topology utilizes all the physical channels and has no physical bottleneck on any node.

Figure 5.6 presents the result of the software `broadcast` on *Spirit* and *Python*. It can be seen that software `broadcast` on *Spirit* costs 10× more time to finish than the hardware MPE using Binomial Tree. With advanced architecture and fast interconnect, *Python* is able to achieve very small latency. Compare the MPE `broadcast` to the software `broadcast`, it can be seen that for MPE `broadcast` can effectively improve the latency by 1000× against *Spirit*. For small messages ($<$ 256 word), MPE `broadcast` can outperform *Python* cluster. However, due to the saturation of the bandwidth, the latency is dominated by the size of the payload and is surpassed by



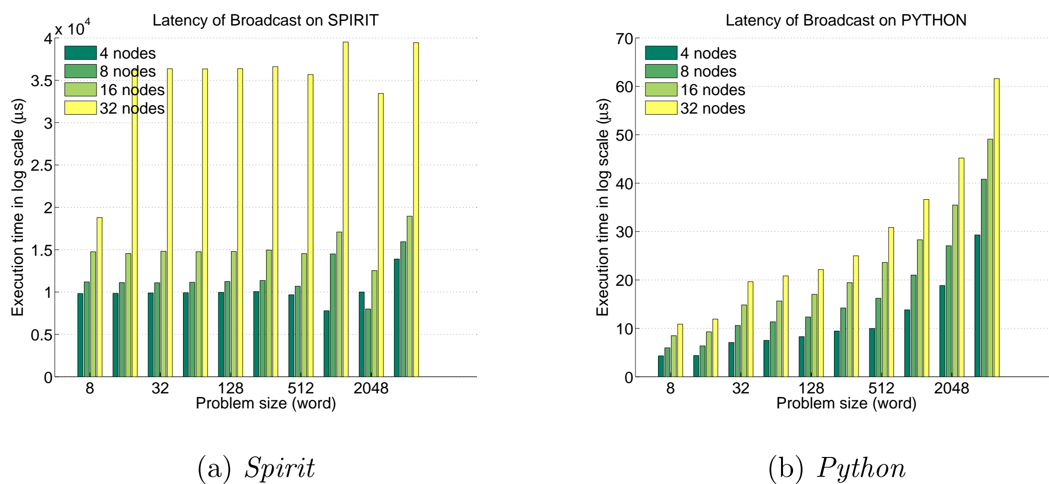
(a) 4 nodes

(b) 8 nodes

(c) 16 nodes

(d) 32 nodes

Figure 5.5: Latency of different `broadcast` topologies



(a) *Spirit*

(b) *Python*

Figure 5.6: Latency of software `broadcast`

Python for large payload.

5.3.3 Reduce Performance Result

`Reduce` operation can be considered as the reverse operation of `broadcast` — every task send the local data to the parent task, along with the communication, a commutative and associative computation is applied to the data. After the operation, the root node has the final result. Like the `broadcast`, this unidirectional communication pattern can establish a pipelined structure, which can effectively increase the bandwidth of the communication. To measure the latency, a hardware MPE `barrier` is inserted between the repeating `reduce` requests. Then the hardware `barrier` time is subtracted from the measured results.

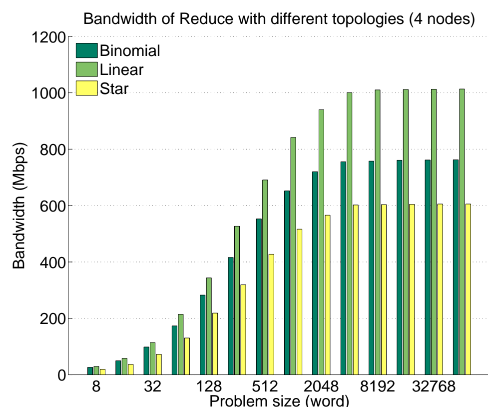
5.3.3.1 Bandwidth

Figure 5.7 shows the bandwidth results of MPE `reduce`. Similar to MPE `broadcast`, Linear Tree performs the best in all the test cases, because of the pipelined topology. Star Tree only obtains a small bandwidth, because it does not have communication parallelism. Binomial Tree performs in between the Linear Tree and the Star Tree.

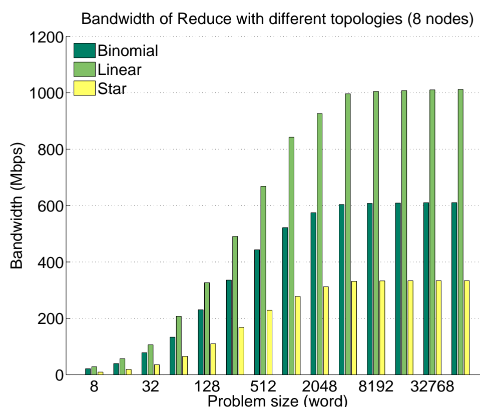
Figure 5.8 shows the bandwidth results of `reduce` on *Spirit* and *Python*. Because `reduce` operation involves a computation, as *Spirit* does not have a floating point unit, all the floating point computations are processed through the library. The bandwidth on *Spirit* can only reach 15 Mbps. *Python* is able to reach 5.0 Gbps bandwidth when the number of nodes is small. As the number of nodes reach 32, the bandwidth falls below 1.0 Gbps.

5.3.3.2 Latency

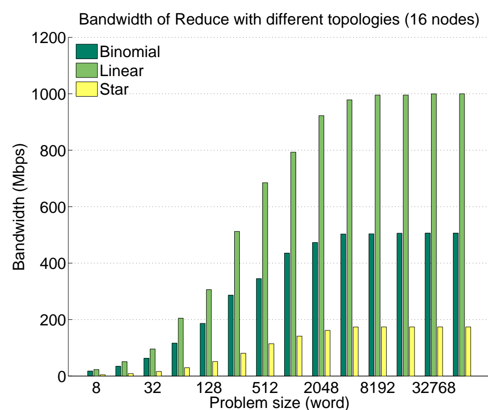
Figure 5.9 presents the latency results of MPE `reduce`. It exhibits almost identical results as `broadcast`. Binomial Tree has the best performance because maximum parallelism can be obtained from the topology, while Linear Tree does not perform well because of the relay mechanism. Star Tree shows small latency for small scale system (4 nodes), but shows huge latency for relatively large scale system (32 nodes),



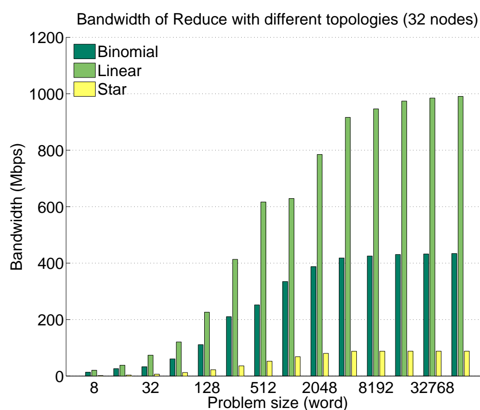
(a) 4 nodes



(b) 8 nodes

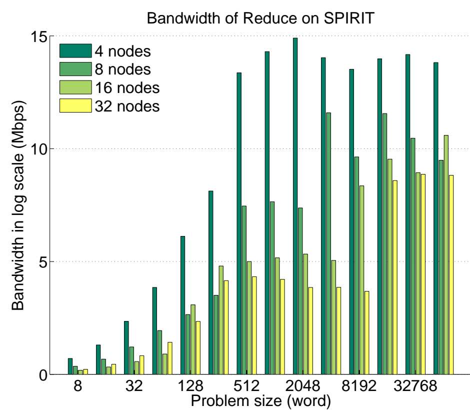


(c) 16 nodes

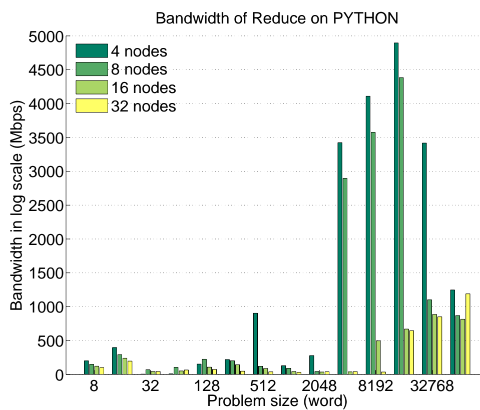


(d) 32 nodes

Figure 5.7: Bandwidth of different `reduce` topologies

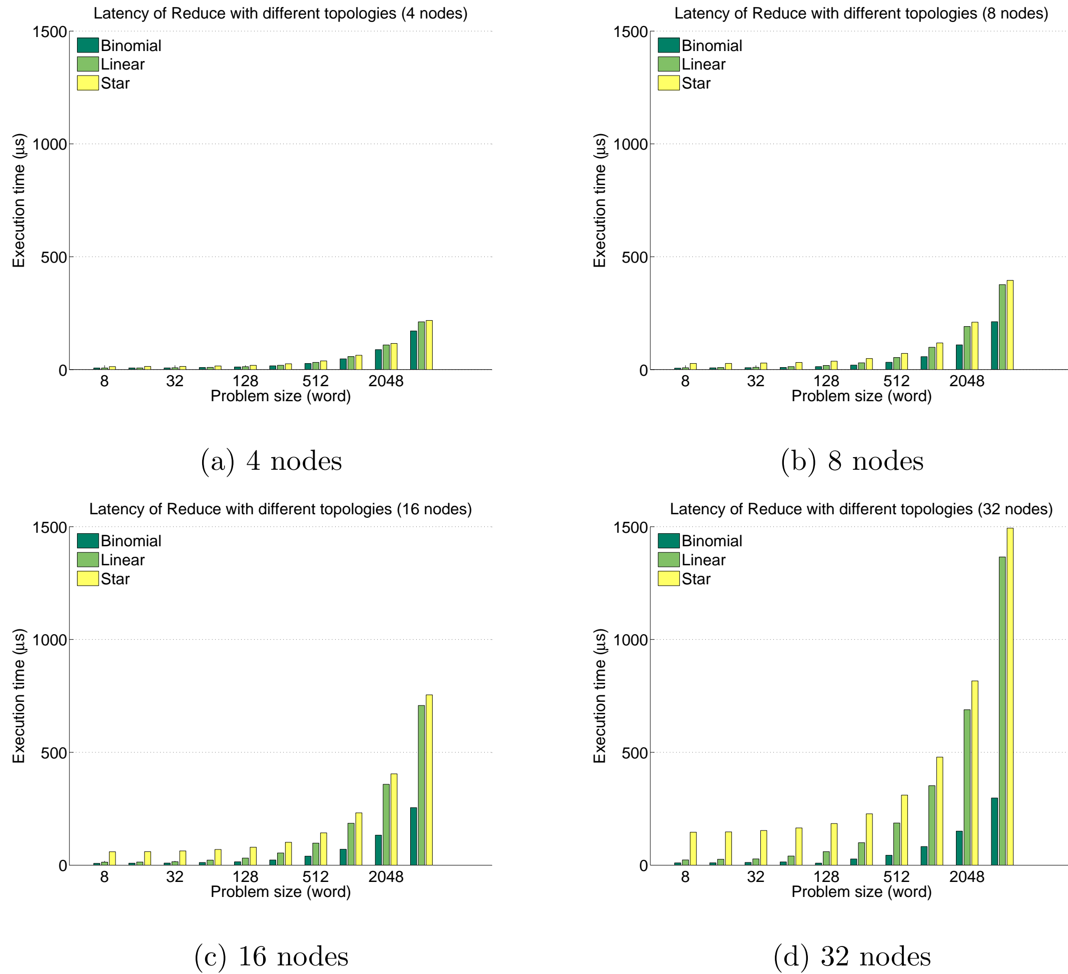


(a) *Spirit*



(b) *Python*

Figure 5.8: Bandwidth of software `reduce`



(a) 4 nodes

(b) 8 nodes

(c) 16 nodes

(d) 32 nodes

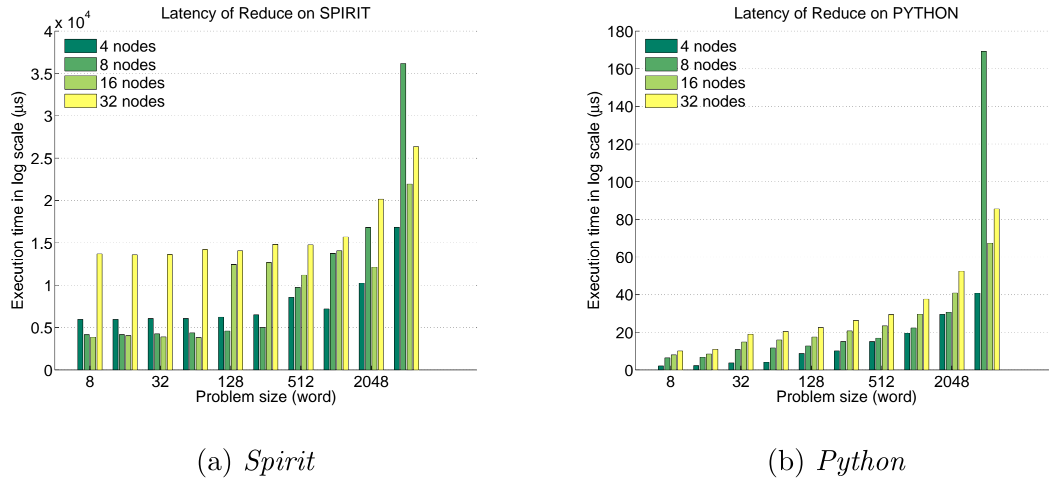
Figure 5.9: Latency of different `reduce` topologies

due to the bottleneck on the root node.

Figure 5.10 shows the measured latency result on *Spirit* and *Python*. It can be seen that hardware MPE can improve the latency by 100× against *Spirit*. For small message, MPE exhibits similar performance as *Python* cluster. For large payload, due to the saturation of the bandwidth, the latency result shows linear relationship with the payload.

5.3.4 Allreduce Performance Result

Allreduce operation can be implemented by combining `reduce` and `broadcast` — all the tasks first execute `reduce` operation, after the root task has the updated result,



(a) *Spirit* (b) *Python*

Figure 5.10: Latency of software `reduce`

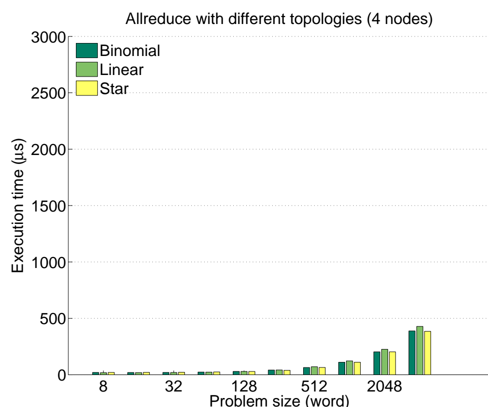
it initiates `broadcast` operation and updates the result for all the other tasks. Since `reduce` and `broadcast` operate in the reversed communication pattern, it breaks the pipelined operation flow. Only latency results are presented.

Figure 5.11 lists MPE `allreduce` results of different topologies. Like the results seen in Figure 5.1, because there is no pipelined operation in `allreduce`, the dimensionality of the network becomes the only performance factor. Therefore, Binomial Tree performs the best among all tree structures. For small messages, Star Tree performs well, but for large messages, root node becomes the bottleneck.

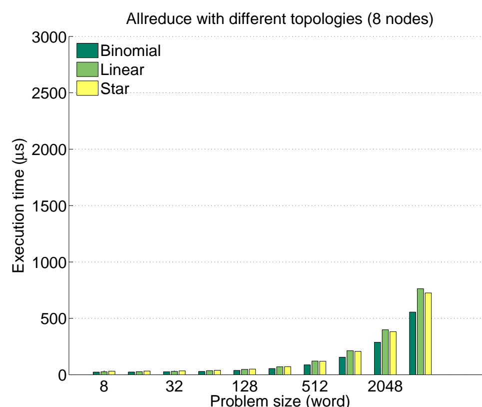
Figure 5.12 shows the traditional software `allreduce` results of *Spirit* and *Python*. It can be observed that MPE can improve the latency of `allreduce` by $\approx 50\times$ to $\approx 350\times$.

5.3.5 Summary

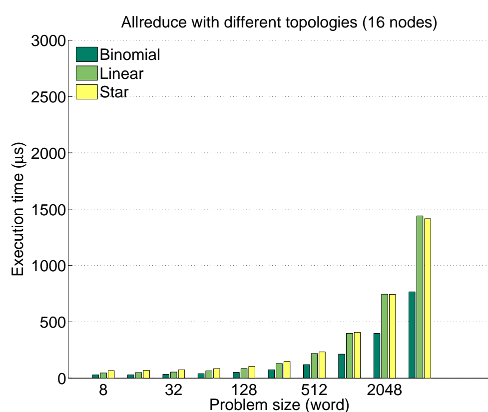
The collected results in Stage 1 show that the hardware MPE can significantly reduce the communication time. Among the 3 communication topologies, the Linear Tree is able to provide the highest bandwidth for unidirectional communication, but it costs the longest delay in every test cases. The Binomial Tree can leverage the physical channels and provide the highest parallelism, which results in the lowest



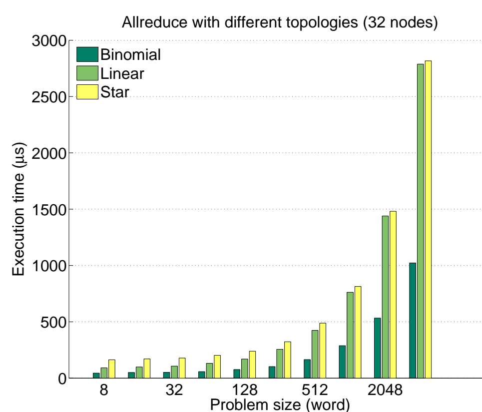
(a) 4 nodes



(b) 8 nodes

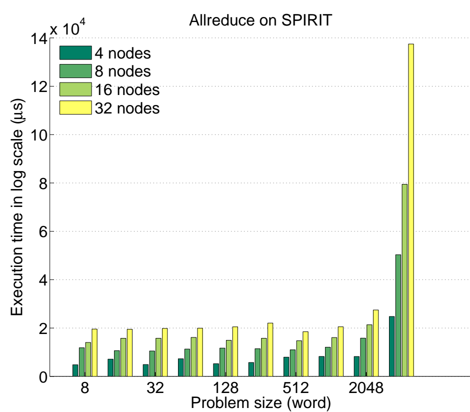


(c) 16 nodes

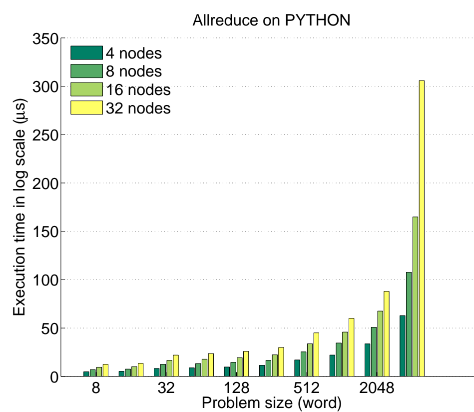


(d) 32 nodes

Figure 5.11: Execution time of different `allreduce` topologies



(a) *Spirit*



(b) *Python*

Figure 5.12: Latency of software `allreduce`

latency in all the tests and moderate bandwidth. The Star Tree reuses the physical channels and it is able to achieve good performance when both the number of nodes and the communication payload are small.

As a reference, *Python* cluster is generally performing better than the hardware MPE on *Spirit*. There are several reasons. The first reason is that *Python* has more advanced architecture and interconnect and the *Spirit* is running relatively slow processor. Though using hardware MPE can improve the raw communication performance, all the rest software stack is running at a slow frequency. Even the `MPI_Wtime()` is running at a 10× slower speed. The second reason is that the communication payload on *Spirit* is not running with cache, whereas on *Python* all the communication payload is running with cache.

To leverage the hardware MPE in real applications such as HPL and NPB, a “replacement” API of traditional MPI is used. However, since the benchmarks are not designed to test communication, there are not frequent `barrier`, `reduce`, and `broadcast` function calls. The performance improvement is not distinctive.

5.4 Communication Model

The hardware counter is inserted in all the hardware communication primitives counting the non-idle clock cycles. The hardware counter showed very close result as the `MPI_Wtime()`. This is due to the “wait state” in the FSM that caused by the asynchronism between the nodes. Equation 5.1 shows the total execution time (T_{total}) consists of 3 portions: idle time (T_{idle}), hardware processing time ($T_{running}$), and wait time (T_{wait}).

$$T_{total} = T_{idle} + T_{running} + T_{wait} \tag{5.1}$$

5.4.1 Linear Fitting for Barrier

Figure 5.13 shows the mathematic fitting for the measured `barrier` data. Because `barrier` does not involve any data operation, plus the FSM only introduces few

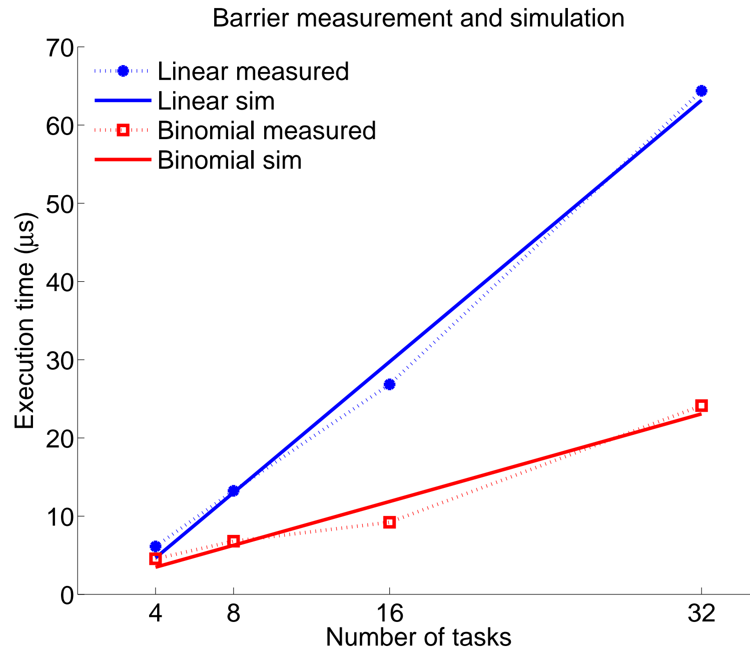


Figure 5.13: Mathematic fitting for MPE `barrier`

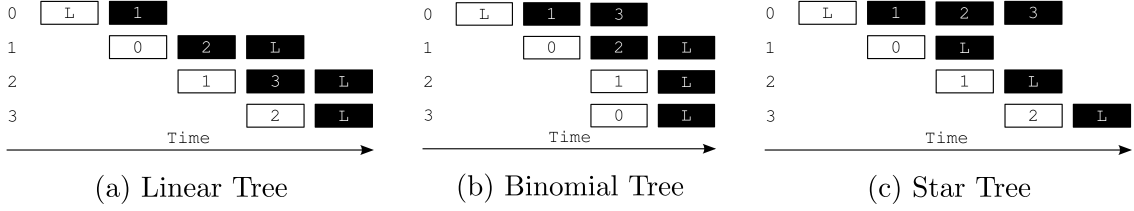
clock cycles $T_{running}$, the majority time is T_{wait} for the asynchronous nodes, which is determined by specific systems.

5.4.2 Latency Model

Unlike the `barrier`, `broadcast` and `reduce` spend quite amount of clock cycles on processing the data, which makes $T_{running}$ the major portion of the total time. Shown in Equation 5.2, $T_{running}$ can be further broken into two part: T_{fsm} and $T_{payload}$. T_{fsm} represents the time spent in the states other than "payload states". $T_{payload}$ denotes the time actually spent on processing the data .

$$T_{running} = T_{fsm} + T_{payload} \tag{5.2}$$

Figure 5.14 illustrates the time chart of `broadcast` in a viewpoint of the communication payload. White blocks represent input operations, and dark blocks represent output operations. The number in the block represents the source or the destination.



(a) Linear Tree (b) Binomial Tree (c) Star Tree

Figure 5.14: Time chart of `broadcast` operation

Table 5.1: `Broadcast` measurement vs. simulation

(a) Absolute differences

Linear	4	8	16	32
8	5.87μs	6.03μs	7.99μs	13.4μs
256	5.00μs	5.12μs	6.01μs	9.99μs
1024	5.11μs	5.12μs	6.98μs	8.93μs
4096	5.53μs	5.72μs	7.41μs	7.80μs
Binomial	**8**	**256**	**1024**	**4096**
8	5.83μs	5.92μs	6.29μs	6.89μs
256	5.01μs	5.53μs	6.08μs	5.26μs
1024	5.14μs	5.00μs	7.23μs	6.92μs
4096	5.56μs	6.53μs	7.27μs	4.87μs
Star	**4**	**8**	**16**	**32**
8	6.89μs	9.29μs	10.6μs	13.9μs
256	4.60μs	4.63μs	7.06μs	8.85μs
1024	4.48μs	3.96μs	5.21μs	7.45μs
4096	4.60μs	4.64μs	6.14μs	3.12μs

(b) Relative differences

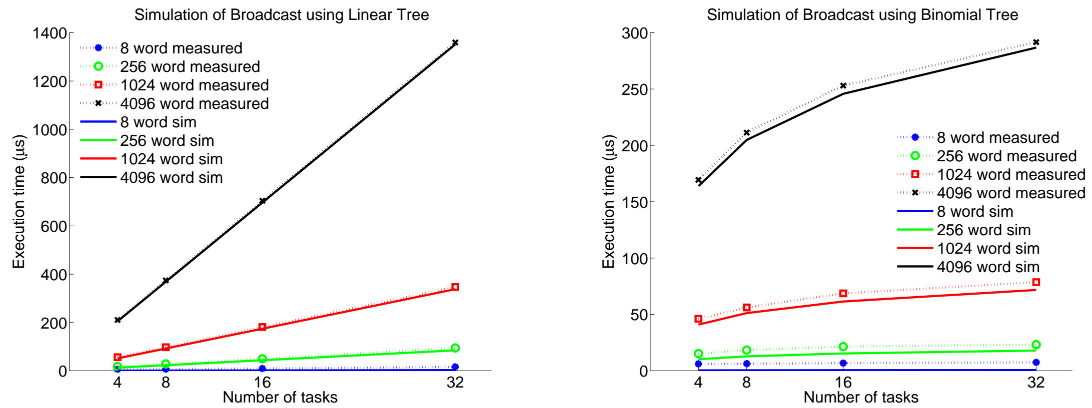
Linear	4	8	16	32
8	93.6%	89.3%	85.4%	83.5 %
256	28.0%	18.1%	12.1%	10.5 %
1024	9.08%	5.26%	3.85%	2.57 %
4096	2.63%	1.52%	1.05%	0.57 %
Binomial	**4**	**8**	**16**	**32**
8	94.7%	93.6%	92.9%	92.4 %
256	32.8%	30.1%	28.3%	22.6 %
1024	11.1%	8.90%	10.5%	8.80 %
4096	3.25%	3.08%	2.87%	1.67 %
Star	**4**	**8**	**16**	**32**
8	94.5%	92.8%	88.6%	84.0%
256	26.4%	16.7%	13.9%	9.48%
1024	8.05%	4.12%	2.90%	2.15%
4096	2.19%	1.24%	0.875%	0.230%

The letter "L" denotes the local DMA transaction. These analytic models can be expressed in following equations:

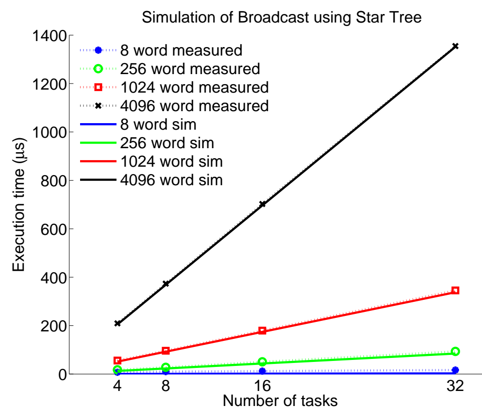
$$\begin{aligned} T_{broadcast_linear} &= (2 + n - 1) \times P \\ T_{broadcast_binomial} &= (2 + log_2(n)) \times P \\ T_{broadcast_star} &= (2 + n - 1) \times P \end{aligned} \tag{5.3}$$

In Equation 5.3, n represents the number of nodes and P represents the communication payload. The simulation results shown in Figure 5.15 exhibit close match between the analytic model and the measured value.

Table 5.1 lists the differences between the measurement and the simulation. It

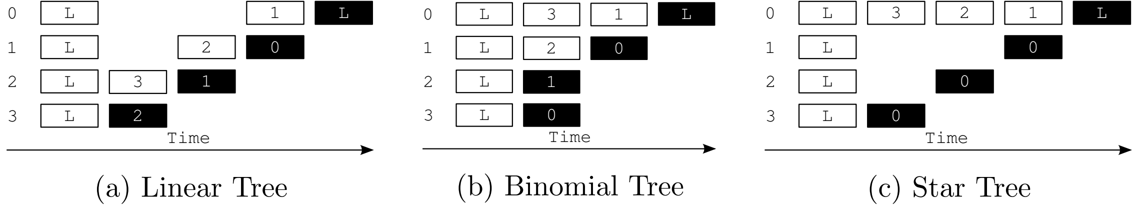


(a) Linear Tree (b) Binomial Tree



(c) Star Tree

Figure 5.15: Latency simulation of `broadcast`



(a) Linear Tree (b) Binomial Tree (c) Star Tree

Figure 5.16: Time chart of `reduce` operation

can be observed that the absolute differences range consistently from 3μs to 14μs. The time difference includes asynchronous wait, software overhead, and measurement errors. For small payload size, relative difference is large, this is because the majority of time is asynchronous wait and software overhead. For large payload size, payload time is the major portion.

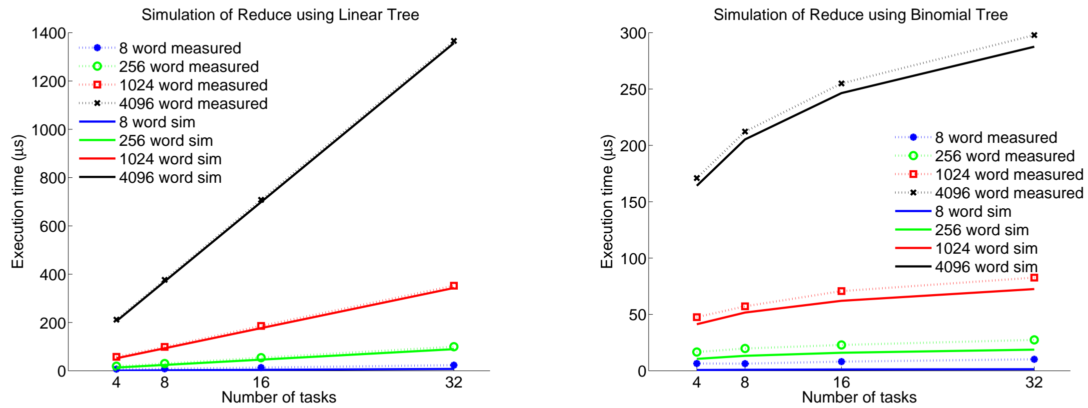
Similar to `broadcast`, Figure 5.16 illustrates the time chart of `reduce` in a viewpoint of the communication payload. These analytic models can be summarized in Equation 5.4. Note that O represents the overhead from the pipelined computation core.

$$\begin{aligned} T_{reduce_linear} &= (2 + n - 1) \times P + (n - 1) \times O \\ T_{reduce_binomial} &= (2 + log_2(n)) \times P + log_2(n) \times O \\ T_{reduce_star} &= (2 + n - 1) \times P + (n - 1) \times O \end{aligned} \tag{5.4}$$

Figure 5.17 presents the simulation result of `reduce`. Figure 5.17a shows close match between the model and measured value. Both Figure 5.17b and Figure 5.17c show increasing gap between the model and the measured value. Table 5.2 illustrates that the growing gap is caused by the handshaking behavior between the parent and children.

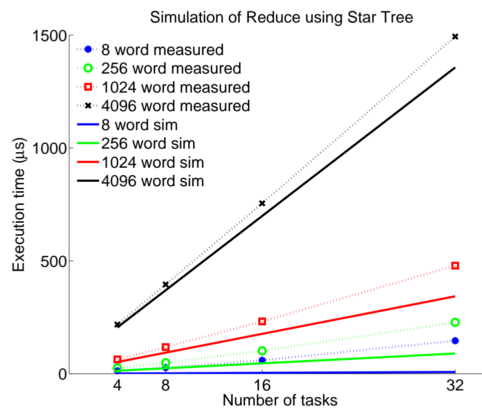
5.4.2.1 Bandwidth

The bit rate of the Aurora channel is 4.0 Gbits/s, removing the error check bits makes the actual data rate 3.2 Gbits/s. Using the time charts in Figure 5.14 and Figure 5.16, the maximum bandwidth is calculated using the max stages N_{stage} dividing



(a) Linear Tree

(b) Binomial Tree



(c) Star Tree

Figure 5.17: Latency simulation of `reduce`

Table 5.2: `Reduce` measurement vs. simulation

(a) Absolute differences

Linear	4	8	16	32
8	5.90μs	6.37μs	9.60μs	15.7μs
256	5.77μs	5.68μs	8.32μs	10.1μs
1024	5.92μs	5.63μs	9.26μs	9.46μs
4096	6.29μs	6.74μs	9.08μs	9.17μs
Binomial	**4**	**8**	**16**	**32**
8	5.83μs	5.41μs	7.01μs	8.82μs
256	6.05μs	6.45μs	6.89μs	8.69μs
1024	6.30μs	5.51μs	8.61μs	10.2μs
4096	6.83μs	6.97μs	8.54μs	10.3μs
Star	**4**	**8**	**16**	**32**
8	12.3μs	25.2μs	55.7μs	138 μs
256	12.4μs	24.6μs	55.6μs	138 μs
1024	11.8μs	24.6μs	55.3μs	136 μs
4096	12.4μs	25.7μs	56.0μs	136 μs

(b) Relative differences

Linear	4	8	16	32
8	87.0%	77.6%	71.9%	67.4%
256	30.3%	19.0%	15.3%	10.2%
1024	10.3%	5.69%	4.98%	2.69%
4096	2.97%	1.79%	1.28%	0.671%
Binomial	**4**	**8**	**16**	**32**
8	90.1%	86.0%	86.2%	86.6%
256	36.4%	32.7%	30.1%	31.7%
1024	13.2%	9.64%	12.2%	12.3%
4096	3.99%	3.28%	3.35%	3.47%
Star	**4**	**8**	**16**	**32**
8	93.3%	93.2%	93.7%	94.8%
256	48.3%	50.4%	54.8%	60.7%
1024	18.5%	20.9%	23.8%	28.4%
4096	5.70%	6.51%	7.43%	9.16%

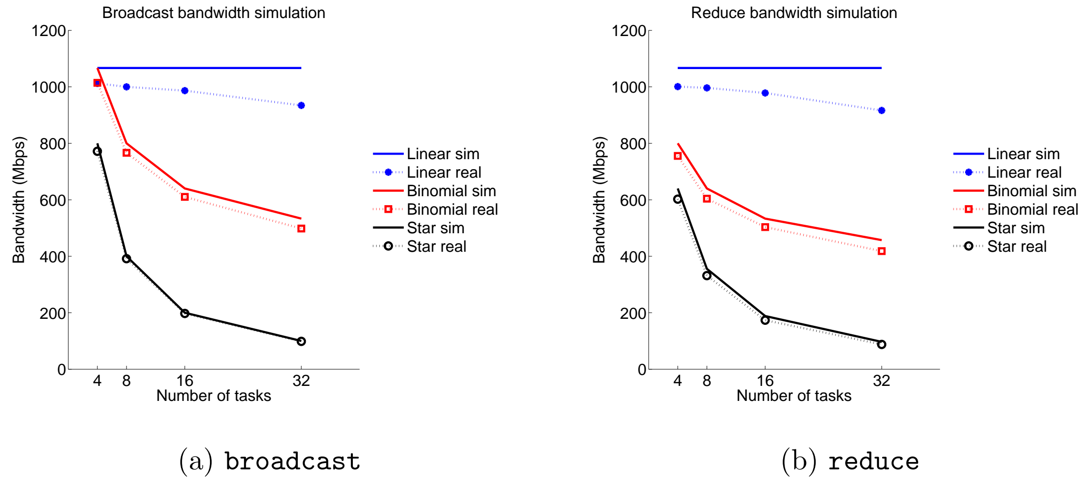
the actual data rate, shown in Equation 5.5.

$$B = 3.2/N_{stage} \tag{5.5}$$

Figure 5.18 shows the simulated max bandwidth, and Table 5.3 calculates the differences between the measurement and the simulation. It can be observed that Star Tree has the closest match between the measurement and the simulation among all the topologies. Linear Tree has growing gaps between the measurement and the simulation. This is because payload operation is the dominating operation in Star Tree, other operations are relatively constant and small to the payload. Whereas Linear Tree hides payload operations with the pipelined communication pattern, which exposes the growing gap occupied by other operations (e.g. handshaking operation).

5.5 Stage 2 Experiment

The following experiments test the hardware MPE with custom hardware accelerators. To offload the workload of the general PE, the hardware fabric can be used to accelerate both computation and the communication. Custom parallel FFT operation and parallel matrix-vector multiplication are tested.

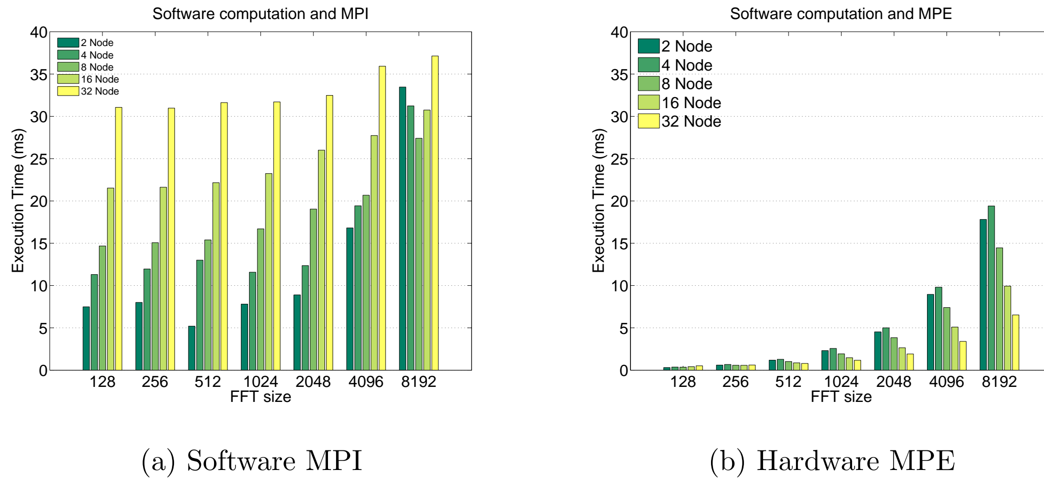


(a) `broadcast` (b) `reduce`

Figure 5.18: Max bandwidth simulation of `broadcast` and `reduce`

Table 5.3: Bandwidth measurement vs. simulation

Broadcast	4	8	16	32
Linear	54.3 Mbps	66.7 Mbps	80.2 Mbps	132 Mbps
Binomial	52.5 Mbps	33.7 Mbps	29.7 Mbps	35.1 Mbps
Star	28.2 Mbps	8.59 Mbps	2.49 Mbps	1.31 Mbps
Reduce	4	8	16	32
Linear	65.9 Mbps	70.2 Mbps	88.3 Mbps	150 Mbps
Binomial	44.6 Mbps	36.1 Mbps	30.1 Mbps	38.8 Mbps
Star	37.9 Mbps	24.1 Mbps	14.6 Mbps	9.21 Mbps



(a) Software MPI

(b) Hardware MPE

Figure 5.19: Communication impact on software FFT

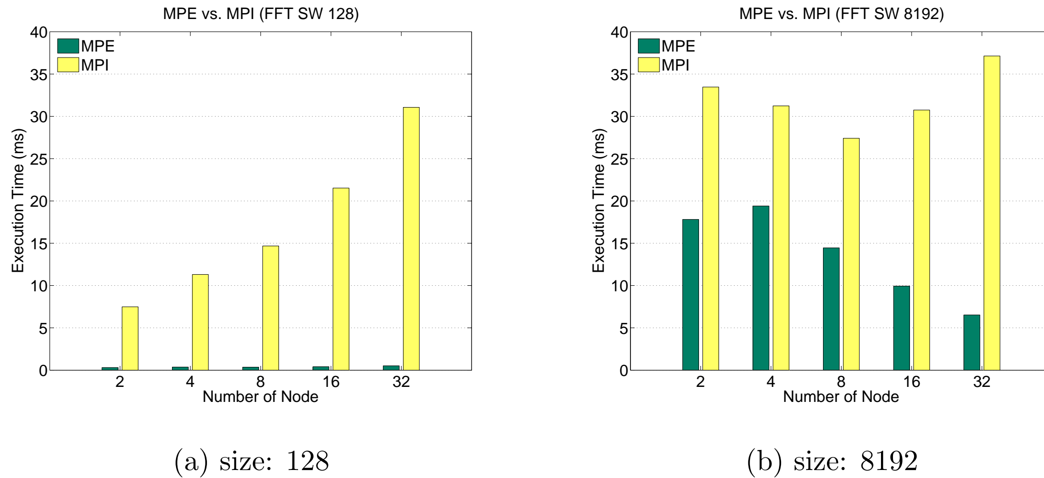
5.5.1 Parallel Fast Fourier Transformation

The parallel Fast Fourier Transformation (FFT) tests the inter-node stages of the FFT DIF algorithm, including the computation as well as the communication. Four test sets are experimented: 1. software computation and software MPI; 2. software computation and hardware MPE; 3. hardware accelerated computation and software MPI; 4. hardware accelerated computation and hardware MPE.

5.5.1.1 Communication Impact on Software FFT Computation

Figure 5.19 shows the impact of the hardware MPE and the software MPI on software FFT computation. The reported results are total execution time including the computation time and the communication time. It can be seen that hardware MPE can effectively reduce the communication time, and improve the overall execution time.

Figure 5.20 compares the software MPI and hardware MPE for certain problem size. For small problem size shown in Figure 5.20a, the test using software MPI spends the majority execution time on the communication. For large problem size in Figure 5.20b, when the number of nodes is relatively small, both hardware and software communication can help reduce the workload and reduce the overall time.



(a) size: 128

(b) size: 8192

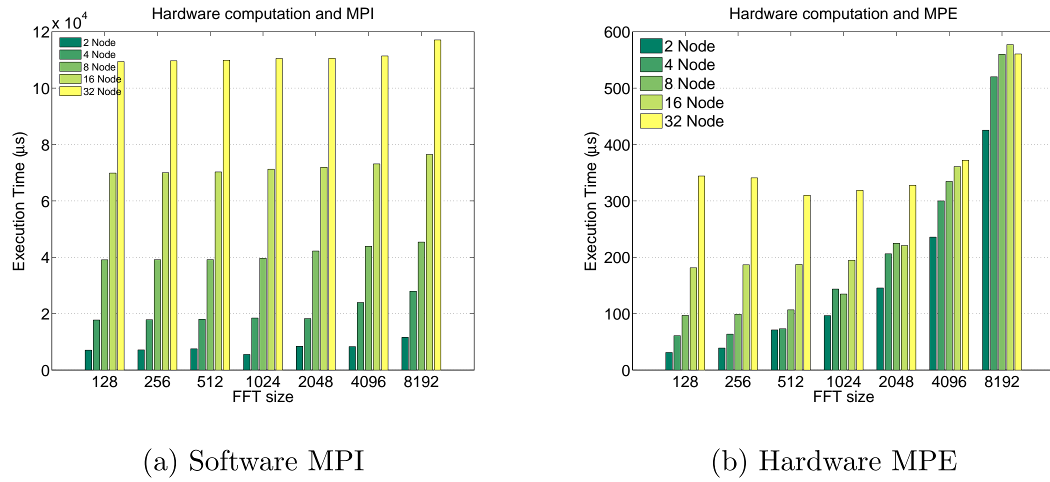
Figure 5.20: Comparison of communication with software FFT

However, as the number of nodes increase, software communication is actually adding more overhead to the execution time, whereas the hardware MPE is able to keep the trend well.

5.5.1.2 Communication Impact on Hardware FFT Computation

Figure 5.21 is using hardware FFT core to accelerate the computation. One interesting observation in Figure 5.21a is that using hardware processing elements is increasing the overall execution time. This is because the hardware accelerator requires extra communication to coordinate the hardware with the existing software. By combining the hardware MPE and hardware FFT, Figure 5.21b shows great improvement in performance ($> 20\times$) over the combination of software FFT and hardware MPE.

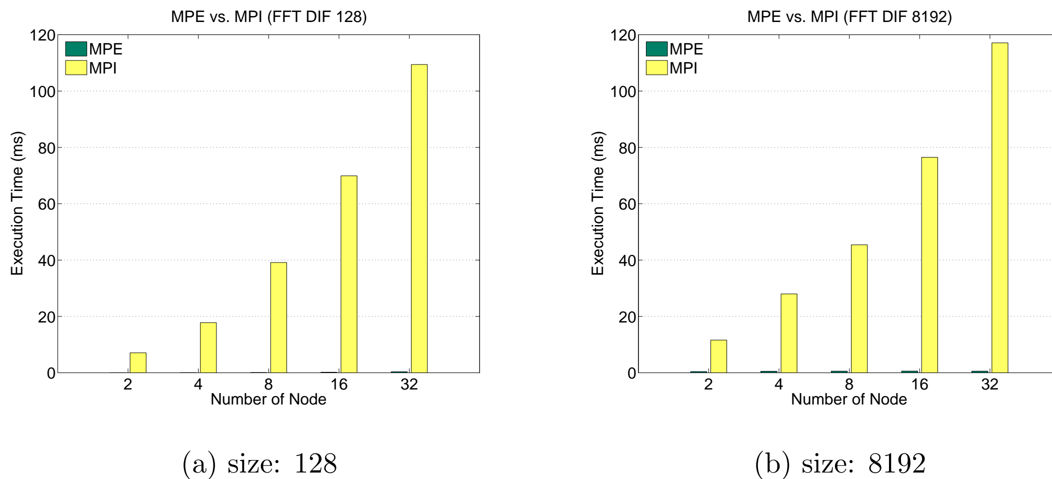
Figure 5.22 illustrates the communication impact of hardware MPE and software MPI on the hardware accelerated FFT computation. It can be observed that because the hardware FFT computation only occupies a small amount of time on actual computation. All the rest of the execution is spent on the software communication.



(a) Software MPI

(b) Hardware MPE

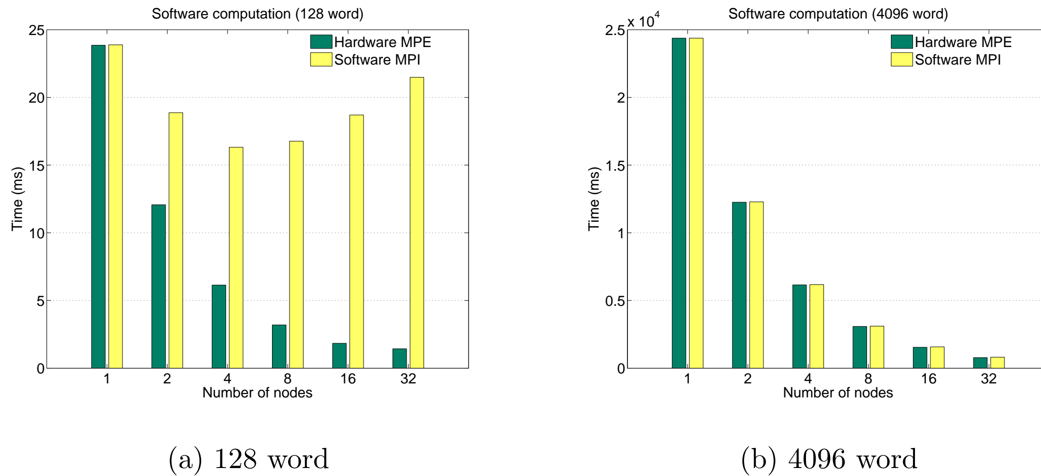
Figure 5.21: Communication impact on hardware FFT



(a) size: 128

(b) size: 8192

Figure 5.22: Comparison of communication with hardware FFT



(a) 128 word

(b) 4096 word

Figure 5.23: Communication impact on software MACC computation

5.5.2 Parallel Matrix-Vector Multiplication

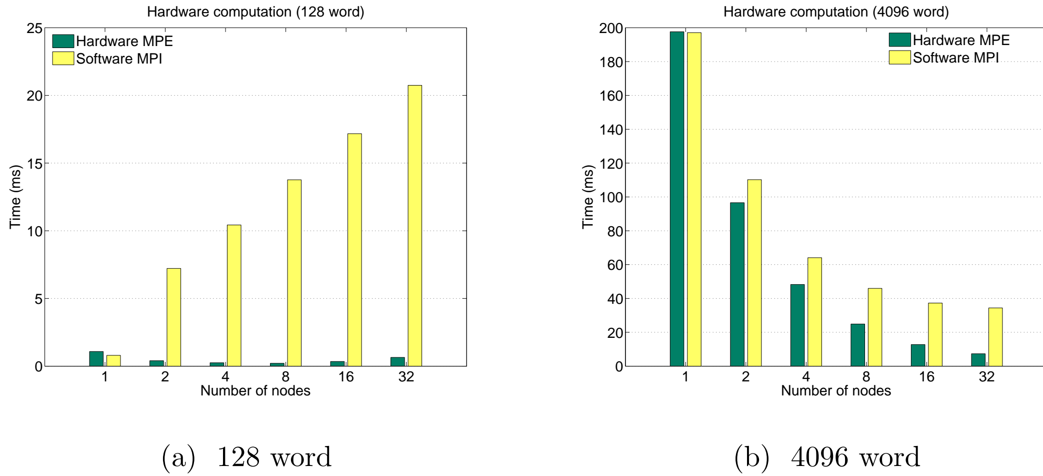
Similar to FFT, the parallel Matrix-Vector Multiplication have tested 4 sets of test. Additionally, because the row-partition gives a uniform view to the problem, a hybrid computing system using hardware and software is tested.

5.5.2.1 Communication Impact on Software MACC Kernel

Presented in Figure 5.23 are comparing the impact of the hardware MPE and the software MPI on software computation. The reported results are total execution time including the computation time and the communication time.

Figure 5.23 shows the classic parallel processing result for matrix size of 4096 words: As more nodes are involved, the total problem is divided into smaller pieces, and the total execution time is reduced. Comparing the hardware MPE and software MPI, there is no distinct difference between the hardware MPE and the software MPI. This is because the computation on software MACC kernel occupies almost the entire execution time (> 100 ms), which makes the communication time indistinguishable.

For small matrix size of 128 words, the result is interesting because it shows contradicting trend compared to the large matrix size. When the number of node is small, both the hardware MPE and the software MPI help distribute the matrix



(a) 128 word (b) 4096 word

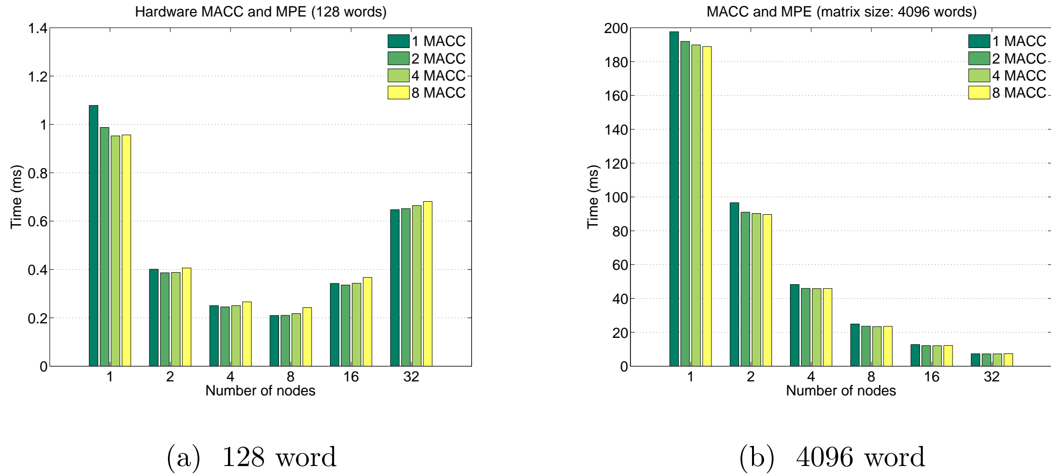
Figure 5.24: Communication impact on accelerated MACC computation

and reduce the total execution time. However, as more nodes are involved, the total execution time using software MPI increases instead of decreasing; while the total execution time using hardware MPE keeps decreasing as expected. This is because the growing number of the node effectively reduces the actual computation (< 25 ms) on each node. Relatively, the increasing software communication time is dominating the overall execution time. But the fast hardware MPE keeps helping reducing the overall execution time.

5.5.2.2 Communication Impact on Hardware MACC Core

Figure 5.24 exhibits the results of hardware MACC core with different communication methods.

Though running at 100 MHz, the hardware MACC core leverages the DSP slices within the FPGA and process the data in a pipeline style. On the contrary, the software MACC kernel runs at 300 MHz, but it lacks of the floating-point unit, and it fetches the data through the bus. From Figure 5.24, it can be seen that the hardware MACC core is able to improve the performance by $\approx 100\times$ for matrix size of 4096 words, either using hardware MPE or using software MPI. For small matrix size of 128 words, it can be observed that software MPI does not improve the performance



(a) 128 word

(b) 4096 word

Figure 5.25: Hardware MACC and MPE

very much, especially for large number of nodes. Similar to the results observed in Figure 5.23, most of the execution time is spent on the software MPI, which makes the performance improvement not obvious. The hardware MPE scales well and the performance improvement ($\approx 20\times$) can be clearly observed.

Figure 5.25 presents hardware MPE with different number of MACC configurations. Under close examination, it can be seen that like software MPI, the hardware MPE also scales up as the number of node increases, but in a much slower speed compared to the software MPI. Figure 5.25 also illustrates how the number of MACC is affecting the computation. There is no significant performance impact of using different number of MACC. This is due to the fact that the switch and MPE has only one memory interface, so multiple message streams are lined up for each MACC core on the switch. As a result, the computation time is largely determined by the number of transactions on the switch, which is fixed number for certain matrix size. An estimated mathematic model is summarized as Equation 5.6:

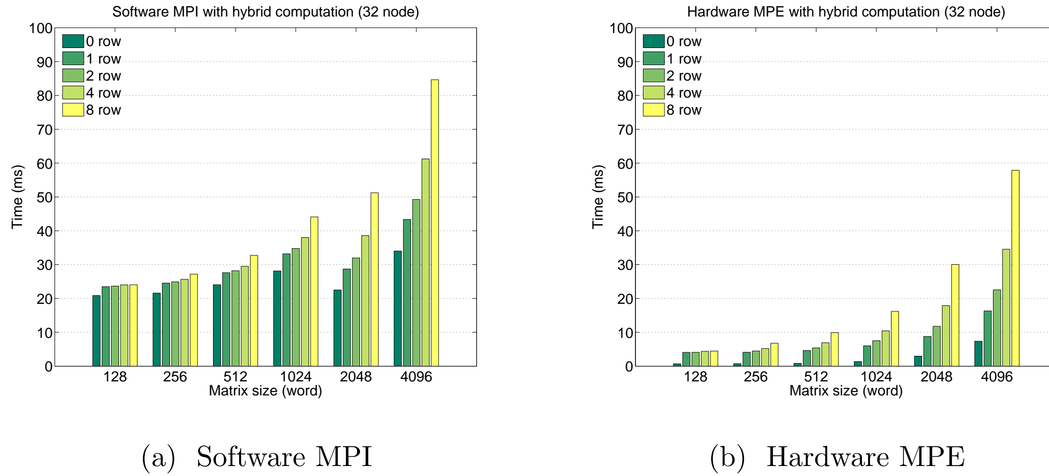
$$\left.\begin{aligned} T &= T_{vector_B} + T_{partial_A} \\ T_{vector_B} &= N_{macc} \times L_{row} \\ T_{partial_A} &= (L_{row} \times N_{macc} + O_{macc}) \times N_{iter} \\ N_{iter} &= L_{row}/N_{nodes}/N_{macc} \end{aligned}\right\} \Longrightarrow$$

$$\begin{aligned} T &= \frac{L_{row}^2}{N_{nodes}} + L_{row} \times N_{macc} + \frac{O_{macc} \times L_{row}}{N_{nodes} \times N_{macc}} \\ &\geq \frac{L_{row}^2}{N_{nodes}} + 2L_{row} \times \sqrt{O_{macc}/N_{nodes}} \end{aligned} \tag{5.6}$$

The equations above divide the computation time into the time for broadcast vector B (T_{vector_B}) and the time for processing the partial matrix A ($T_{partial_A}$). For partial matrix A of size (L_{row}/N_{nodes}) larger than the number of MACC (N_{macc}), the hardware MACC requires multiple iterations of computation (N_{iter}). Because the computations on MACC are running in a pipelined style, only one overhead from the MACC (O_{macc}) is considered. From this model, it can be concluded that the execution time is determined by the size of the matrix ($O(L_{row}^2)$). This result may suggest that scaling up the number of MACC cannot further improve the performance. However, this result is actually due to the single memory interface. For other applications, the hardware accelerators may scale well.

5.5.3 Hybrid Computing System

Perhaps the most interesting result of this work is the hybrid computing system. Since matrix-vector multiplication can be broken into multiple uniform vector-vector multiply-accumulation operations, both the hardware MACC core and the software MACC kernel are able to independently compute their results in parallel. In this experiment, hardware MPE is used as the communication method, 8 MACC cores are implemented in hardware and various sizes of workload are tested on the software MACC kernel. Two threads are generated from the Pthreads library, one is the hardware MACC thread, and the other is the software MACC thread.



(a) Software MPI (b) Hardware MPE

Figure 5.26: Communication impact on hybrid computing system

Figure 5.26a shows that the software MPI adds additional workload to the hybrid computing system, while the hardware MPE does not. It can also be observed in Figure 5.26 that the software MACC thread actually is slowing the whole system down. There are two major reasons for the slowing done: First, using Pthreads adds software overhead. Second, the PowerPC used in the test has only one processor core, which has to process the software MACC computation as well as the Pthreads overhead. However, these two issues can be resolved in future heterogeneous multi/many-core systems. Software threads may run on one or several separate processor cores which have fast clock rate and better floating-point units. Thereby using hardware MPE is more meaningful as the hybrid computing system may purely focus on the computation while the hardware MPE will facilitate the communication.

5.6 Validation

To answer the thesis question "Can hardware be used to provide a unified view of the heterogeneous system and provide message-passing function to the chip as well as to the cluster?". Chapter 1 further divides the thesis question into following 5 questions. This section answers the 5 questions by summarizing the experimental results.

Table 5.4: Resource utilization of hardware MPE

	Used	Available	Percentage
Number of Slices:	1717	25280	6%
Number of Slice Flip Flops:	1283	50560	2%
Number of 4 input LUTs:	2843	50560	5%
Number of FIFO16/RAMB16s:	16	232	6%
Number of DSP48s:	4	128	3%

Table 5.5: Performance improvement of hardware MPE

	MPE	Software MPI on *Spirit*	Improvement
`Barrier` 4 nodes	4.54 μs	4509 μs	993×
`Barrier` 32 nodes	24.10 μs	18755 μs	740×
`Broadcast` 4 nodes	169.4 μs	13900 μs	82×
`Broadcast` 32 nodes	291.6 μs	39440 μs	135×
`Reduce` 4 nodes	171 μs	16840 μs	98×
`Reduce` 32 nodes	298 μs	26360 μs	88×
`Broadcast` 4 nodes	1010 Mbps	29 Mbps	35×
`Broadcast` 32 nodes	934 Mbps	29.7 Mbps	31×
`Reduce` 4 nodes	1000 Mbps	14 Mbps	71×
`Reduce` 32 nodes	916 Mbps	3.86 Mbps	237×

1. Is the hardware MPE practical and feasible?

   Yes, it is functioning correctly and Table 5.4 shows it occupies reasonable hardware resources.

2. Can the hardware communication engine improve the overall performance?

   Yes, Table 5.5 shows that hardware MPE can improve the performance by $\approx 30\times$ to $\approx 1000\times$.

3. Is the hardware communication engine scalable when the system grows?

   Yes, model shows it fully utilizes the bandwidth of the physical infrastructure. With small amount of overhead, latency is dominated by the communication payload.

Table 5.6: Performance improvement of MPE on heterogeneous systems

Software computation +	MPE	software MPI	Improvement
FFT 2 nodes	17.8 ms	33.5 ms	188%
FFT 32 nodes	6.53 ms	37.1 ms	568%
MVM 2 nodes	1230 ms	1230 ms	100%
MVM 32 nodes	774 ms	811 ms	104%
Hardware computation +	MPE	software MPI	Improvement
FFT 2 nodes	0.425 ms	11.6 ms	2729%
FFT 32 nodes	0.561 ms	117 ms	20855%
MVM 2 nodes	89.6 ms	110 ms	122%
MVM 32 nodes	7.36 ms	34.3 ms	466%

4. Can the hardware MPE be used in heterogeneous system?

   Yes, the heterogeneous system tests show it can be used in heterogeneous system, distributing data directly to heterogeneous hardware.

5. In the heterogeneous system, can hardware communication engine bring performance gain?

   Yes, Table 5.6 shows MPE can improve the performance by $\approx 200\times$ for certain computation.

CHAPTER 6: CONCLUSION

Heterogeneous multi/many-core chips are widely used in today's top tier supercomputers. Within the heterogeneous chips, on-chip network often plays a major role by connecting the processing elements together. However, as the system scales up, traditional programming methods, such as MPI, may not effectively use the on-chip network and therefore could make communication the performance bottleneck.

This dissertation designed a MPI-like Message Passing Engine (MPE) as part of the on-chip network, providing point-to-point and collective communication primitives in hardware. On one hand, the MPE offloads the communication workload from the general processing elements. On the other hand, the MPE provides direct interface to the heterogeneous processing elements which can eliminate the data path going around the OS and libraries.

The proposed design has been implemented and experimented on a parallel FPGA system. The footprint of the MPE occupies 6% of hardware resources on Virtex 4 FX60 FPGA. The experimental results have shown that the MPE can significantly reduce the communication time and improve the overall performance. Specifically, within 3 communication topologies, Binomial Tree, Star Tree, and Linear Tree, Binomial Tree exhibits the lowest latency in all experiments. For unidirectional operations such as `broadcast` and `reduce`, Linear Tree is able to pipeline the operation, and thereby achieve sustained bandwidth in all experiments. In addition to the experiments, theoretical studies of the communication primitives have shown the ideal performance match the measured values well.

To investigate how the hardware MPE is integrated with the heterogeneous system. Two heterogeneous configurations are designed and implemented in the FPGA.

The experimental results have shown that the hardware MPE can be tightly coupled with the computing cores, thereby increase the total performance of the parallel computing system. Additionally, a hybrid "MPI+Pthreads" computing system is tested and it shows MPE can effectively offload the communications and let the processing elements play their strengths on the computation.

In summary, the hardware MPE can effectively improve the communication performance in parallel computing systems. The usage of hardware MPE is not limited to FPGA, but can be applied to general multi/many-core processors. Specifically, in future heterogeneous systems with "Big–little" configurations, the hardware MPE can be integrated into "little" processors to assist the communication without the support from the OS.

REFERENCES

[1] D. Patterson and J. Hennessy, *Computer Organization and Design, Revised Fourth Edition: The Hardware/Software Interface*, ser. Morgan Kaufmann Series in Computer Graphics. Elsevier Science, 2011. [Online]. Available: http://books.google.com/books?id=DMxe9AI4-9gC

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[3] T. Sterling, D. Savarese, D. J. Becker, B. Fryxell, and K. Olson, "Communication overhead for space science applications on the beowulf parallel workstation," in *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*, ser. HPDC '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 23–. [Online]. Available: http://dl.acm.org/citation.cfm?id=822081.823047

[4] Intel. Last accessed April 11, 2013. [Online]. Available: http://www.intel.com/

[5] AMD. Last accessed April 11, 2013. [Online]. Available: http://www.amd.com/

[6] Nvidia. Last accessed April 11, 2013. [Online]. Available: http://www.nvidia.com/

[7] IBM. Last accessed April 11, 2013. [Online]. Available: http://www.research.ibm.com/cell/

[8] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the petaflop era: the architecture and performance of roadrunner," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1:1–1:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413372

[9] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," DARPA Information Processing Techniques Office (IPTO) sponsored study, Tech. Rep. TR-2008-13, 2008, http://www.exascale.org/iesp/IESP:Documents.

[10] Intel. Last accessed April 11, 2013. [Online]. Available: http://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html

[11] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek, "Intel® quickpath interconnect architectural features supporting scalable system architectures," in *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*, ser. HOTI '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1109/HOTI.2010.24

[12] AMD. Last accessed April 11, 2013. [Online]. Available: http://www.amd.com/us/products/technologies/hypertransport-technology/Pages/hypertransport-technology.aspx

[13] B. Holden, D. Anderson, J. Trodden, and M. Daves, *HyperTransport 3.1 Interconnect Technology*. Mindshare Press, 2008.

[14] Last accessed April 11, 2013. [Online]. Available: http://www.beowulf.org/

[15] L. W. Tucker and G. G. Robertson, "Architecture and applications of the connection machine," *Computer*, vol. 21, pp. 26–38, August 1988. [Online]. Available: http://dx.doi.org/10.1109/2.74

[16] S. Borkar, R. Cohn, G. Cox, S. Gleason, and T. Gross, "Warp: an integrated solution of high-speed parallel computing," in *Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '88. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 330–339. [Online]. Available: http://dl.acm.org/citation.cfm?id=62972.63015

[17] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, and D. M. Dias, "Sp2 system architecture," *IBM Syst. J.*, vol. 34, pp. 152–184, April 1995. [Online]. Available: http://dl.acm.org/citation.cfm?id=209136.209139

[18] F. Petrini, E. Frachtenberg, A. Hoisie, and S. Coll, "Performance evaluation of the quadrics interconnection network," *Cluster Computing*, vol. 6, pp. 125–142, April 2003. [Online]. Available: http://dx.doi.org/10.1023/A:1022852505633

[19] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, "Myrinet: A gigabit-per-second local area network," *IEEE Micro*, vol. 15, pp. 29–36, February 1995. [Online]. Available: http://dl.acm.org/citation.cfm?id=623261.623898

[20] W. Futral, *InfiniBand architecture development and deployment: a strategic guide to server I/O solutions*, ser. Engineer to Engineer series. Intel Press, 2001. [Online]. Available: http://books.google.com/books?id=qIkAAAAACAAJ

[21] InfiniBand Trade Association. Last accessed April 11, 2013. [Online]. Available: http://www.infinibandta.org/

[22] UNC Charlotte, "University research computing," Nov. 2011, URL: http://www.urc.uncc.edu/urc/.

[23] Open MPI Development Team, "Open MPI: open source high-performance computing," Jun. 2010, URL: http://www.open-mpi.org/.

[24] M. S. Mller, A. Knpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel, "Developing scalable applications with vampir, vampirserver and vampirtrace." in *PARCO'07*, 2007, pp. 637–644.

[25] S. Datta, P. Beeraka, and R. Sass, "Rc-blastn: Implementation and evaluation of the blastn scan function," in *Proceedings of the 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, ser. FCCM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 88–95. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2009.15

[26] R. Pottathuparambil and R. Sass, "A parallel/vectorized double-precision exponential core to accelerate computational science applications," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '09. New York, NY, USA: ACM, 2009, pp. 285–285. [Online]. Available: http://doi.acm.org/10.1145/1508128.1508198

[27] R. Sass and A. Schmidt, *Embedded Systems Design with Platform FPGAs: Principles and Practices*, ser. Morgan Kaufmann. Elsevier Science & Technology, 2010. [Online]. Available: http://books.google.com/books?id=Ki7zs-Ex2d0C

[28] B. A.S., K. R.A., K. D.B., R. J.H., and S. G.M., "Jaguar: The world's most powerful computer." Cray User's Group, 2009. [Online]. Available: http://www.nccs.gov/wp-content/uploads/2010/01/Bland-Jaguar-Paper.pdf

[29] X.-J. Yang, X.-K. Liao, K. Lu, Q.-F. Hu, J.-Q. Song, and J.-S. Su, "The tianhe-1a supercomputer: Its hardware and software," *Journal of Computer Science and Technology*, vol. 26, pp. 344–351, 2011, 10.1007/s02011-011-1137-8. [Online]. Available: http://dx.doi.org/10.1007/s02011-011-1137-8

[30] M. Yokokawa, F. Shoji, A. Uno, M. Kurokawa, and T. Watanabe, "The k computer: Japanese next-generation supercomputer development project," in *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, ser. ISLPED '11. Piscataway, NJ, USA: IEEE Press, 2011, pp. 371–372. [Online]. Available: http://dl.acm.org/citation.cfm?id=2016802.2016889

[31] A. Yonezawa, T. Watanabe, M. Yokokawa, M. Sato, and K. Hirao, "Advanced institute for computational science (aics): Japanese national high-performance computing research institute and its 10-petaflops supercomputer "k"," in *State of the Practice Reports*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 13:1–13:8. [Online]. Available: http://doi.acm.org/10.1145/2063348.2063366

[32] Argonne National Laboratory, "MPICH2: high-performance and widely portable MPI," Jun. 2009, URL: http://www.mcs.anl.gov/research/projects/mpich2/.

[33] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks — summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 158–165. [Online]. Available: http://doi.acm.org/10.1145/125826.125925

[34] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, S. Fineberg, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "THE NAS PARALLEL BENCHMARKS," NASA Ames Research Center, Moffett Field, CA, Technical report RNR-94-007, Mar. 1994.

[35] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[36] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "Logp: towards a realistic model of parallel computation," in *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '93. New York, NY, USA: ACM, 1993, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/155332.155333

[37] M. Small and X. Yuan, "Maximizing mpi point-to-point communication performance on rdma-enabled clusters with customized protocols," in *ICS '09: Proceedings of the 23rd international conference on Supercomputing*. New York, NY, USA: ACM, 2009, pp. 306–315.

[38] A. R. Mamidala, A. Vishnu, and D. K. Panda, "Efficient shared memory and rdma based design for mpi_allgather over infiniband." in *PVM/MPI'06*, 2006, pp. 66–75.

[39] M. Saldana and P. Chow, "TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs," in *Proc. of International Conference on Field Programmable Logic and Applications*, 2006, pp. 1–6.

[40] R. Rabenseifner, "Automatic MPI counter profiling of all users: First results on a CRAY t3e 900-512," in *Message Passing Interface Developer's and User's Conference*, 1999.

[41] R. Rabenseifner, "Optimization of collective reduction operations," in *Computational Science - ICCS 2004, Springer-Verlag LNCS 3036*, 2004, pp. 1–9.

[42] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm, "A Survey of Barrier Algorithms for Coarse Grained Supercomputers," *Chemnitzer Informatik Berichte*, vol. 04, no. 03, Dec. 2004.

[43] E. Freudenthal and A. Gottlieb, "Process coordination with fetch-and-increment," *SIGARCH Comput. Archit. News*, vol. 19, no. 2, pp. 260–268, 1991.

[44] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra, "Automatically tuned collective communications," in *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000, p. 3.

[45] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie, "Distributing hot-spot addressing in large-scale multiprocessors," *IEEE Trans. Comput.*, vol. 36, no. 4, pp. 388–395, 1987.

[46] M. L. Scott and J. M. Mellor-Crummey, "Fast, contention-free combining tree barriers for shared-memory multiprocessors," *Int. J. Parallel Program.*, vol. 22, no. 4, pp. 449–481, 1994.

[47] I. Eugene D. Brooks, "The butterfly barrier," *Int. J. Parallel Program.*, vol. 15, no. 4, pp. 295–307, 1986.

[48] R. e. a. Gupta, "Efficient barrier using remote memory operations on via-based clusters," in *IEEE International Conference on Cluster Computing*, 2002, p. 83.

[49] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: theory, practice, and experience: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 19, pp. 1749–1783, September 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1285358.1285359

[50] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of mpi collective operations," *Cluster Computing*, vol. 10, no. 2, pp. 127–143, 2007.

[51] J. Pješivac-Grbović, G. Bosilca, G. E. Fagg, T. Angskun, and J. J. Dongarra, "Mpi collective algorithm selection and quadtree encoding," *Parallel Comput.*, vol. 33, no. 9, pp. 613–623, 2007.

[52] B. L. Payne, M. Barnett, R. Littlefield, D. G. Payne, and R. V. D. Geijn, "Global combine on mesh architectures with wormhole routing," in *Proc. of 7 th Int. Parallel Proc. Symp*, 1993.

[53] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng, "Optimization of mpi collective communication on bluegene/l systems," in *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*. New York, NY, USA: ACM, 2005, pp. 253–262.

[54] A. Faraj, S. Kumar, B. Smih, A. Mamidala, J. Gunnels, and P. Heidelberger, "Mpi collective communications on the blue gene/p supercomputer: algorithms and optimizations," in *Proceedings of the 23rd international conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 489–490. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542344

[55] H. Subramoni, K. Kandalla, S. Sur, and D. K. Panda, "Design and evaluation of generalized collective communication primitives with overlap using connectx-2 offload engine," in *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*, ser. HOTI '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 40–49. [Online]. Available: http://dx.doi.org/10.1109/HOTI.2010.22

[56] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony, "The percs high-performance interconnect," in *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*, ser. HOTI '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 75–82. [Online]. Available: http://dx.doi.org/10.1109/HOTI.2010.16

[57] R. Brightwell, K. T. Pedretti, K. D. Underwood, and T. Hudson, "Seastar interconnect: Balanced bandwidth for scalable performance," *IEEE Micro*, vol. 26, pp. 41–57, May 2006. [Online]. Available: http://dx.doi.org/10.1109/MM.2006.65

[58] R. Alverson, D. Roweth, and L. Kaplan, "The gemini system interconnect," in *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*, ser. HOTI '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 83–87. [Online]. Available: http://dx.doi.org/10.1109/HOTI.2010.23

[59] R. Brightwell, T. Hudson, R. Riesen, and K. Underwood, "Implementation and performance of portals 3.3 on the Cray XT3," in *IEEE International Conference on Cluster Computing (CLUSTER'05)*, Sep. 2005.

[60] R. Riesen, R. Brightwell, K. Pedretti, K. Underwood, A. B. Maccabe, and T. Hudson, "The Portals 4.0 message passing interface," Sandia National Laboratories, Technical report SAND2008-2639, Apr. 2008.

[61] K. D. Underwood, W. B. L. III, and R. Sass, "Analysis of a prototype intelligent network interface," *Concurrency and Computation: Practice and Experience*, pp. 751–777, 2003.

[62] R. Brightwell, S. P. Goudy, A. Rodrigues, and K. D. Underwood, "Implications of application usage characteristics for collective communication offload," *Int. J. High Perform. Comput. Netw.*, vol. 4, no. 3/4, pp. 104–116, 2006.

[63] D. Buntinas, D. K. Panda, and P. Sadayappan, "Fast nic-based barrier over myrinet/gm," in *Proceedings of the 15th International Parallel &*

Distributed Processing Symposium, ser. IPDPS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 52–. [Online]. Available: http://portal.acm.org/citation.cfm?id=645609.663267

[64] W. Yu, D. K. Panda, and D. Buntinas, "Scalable, high-performance nic-based all-to-all broadcast over myrinet/gm." in *CLUSTER'04*, 2004, pp. 125–134.

[65] W. Yu, D. Buntinas, R. L. Graham, and D. K. Panda, "Efficient and scalable barrier over quadrics and myrinet with a new nic-based collective message passing protocol." in *IPDPS'04*, 2004, pp. –1–1.

[66] W. Yu, D. Buntinas, and D. K. Panda, "High performance and reliable nic-based multicast over myrinet/gm-2." in *ICPP'03*, 2003, pp. 197–204.

[67] D. Buntinas, D. K. Panda, and P. Sadayappan, "Fast nic-based barrier over myrinet/gm." in *IPDPS'01*, 2001, pp. –1–1.

[68] D. P. Vinod Tipparaju, Jarek Nieplocha, "Fast collective operations using shared and remote memory access protocols on clusters," in *In International Parallel and Distributed Processing Symposium*, 2003.

[69] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, pp. 25–35, March 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=623304.624515

[70] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams," in *Proceedings of the 31st annual international symposium on Computer architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 2–. [Online]. Available: http://dl.acm.org/citation.cfm?id=998680.1006733

[71] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin, "Programming the intel 80-core network-on-a-chip terascale processor," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 38:1–38:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413409

[72] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight communications on intel's single-chip cloud computer processor," *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 73–83, February 2011. [Online]. Available: http://doi.acm.org/10.1145/1945023.1945033

[73] J. Wawrzynek, M. Oskin, C. Kozyrakis, D. Chiou, D. A. Patterson, S. lien Lu, J. C. Hoe, and K. Asanovic, "Ramp: Research accelerator for multiple processors," in *In Proceedings of Hot Chips 18*, 2006.

[74] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Ge, C. Kozyrakis, and K. Olukotun, "A practical fpga-based framework for novel cmp research," in *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, ser. FPGA '07. New York, NY, USA: ACM, 2007, pp. 116–125. [Online]. Available: http://doi.acm.org/10.1145/1216919.1216936

[75] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P.-Y. Droz, "Ramp blue: A message-passing manycore system in fpgas." in *FPL'07*, 2007, pp. 54–61.

[76] R. Sass, et al., "Reconfigurable computing cluster RCC project: Investigating the feasibility of FPGA-based petascale computing," in *FCCM '07: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'07)*, 2007, pp. 127–138.

[77] W. V. Kritikos, "Feasibility of serial ata cables for the physical link in high performance computing clusters," Master's thesis, University of Kansas, May 2007.

[78] A. G. Schmidt, S. Datta, A. A. Mendon, and R. Sass, "Productively scaling i/o bound streaming applications with a cluster of fpgas," in *Application Accelerators in High-Performance Computing (SAAHPC), 2010 Symposium on*, july 2010.

[79] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks.* Morgan Kaufmann, 2002.

[80] Xilinx, "Locallink interface specification," www.xilinx.com/products/design_resources/conn_central/locallink_member/sp006.pdf.

[81] B. Huang, A. Schmidt, A. Mendon, and R. Sass, "Investigating resilient high performance reconfigurable computing with minimally-invasive system monitoring," in *High-Performance Reconfigurable Computing Technology and Applications (HPRCTA), 2010 Fourth International Workshop on*, nov. 2010, pp. 1 –8.

[82] AMD, "AMD Accelerated Processing Units."

[83] Xilinx, Inc., "ML410 embedded development platform user guide," September 2008.