ABSTRACT

YAGNA, KARTHIK. Efficient Collective Communication for Multi-core NOC Interconnects. (Under the direction of Dr. Frank Mueller.)

Massive multi-core embedded processors with network-on-chip (NoC) architectures are becoming common. These architectures provide higher processing capability due to an abundance of cores. They provide native core-to-core communication that can be exploited via message passing to provide system scalability. Despite these advantages, multicores pose predictability challenges that can affect both performance and real-time capabilities.

In this work, we develop efficient and predictable group communication using message passing specifically designed for large core counts in 2D mesh NoC architectures. We have implemented the most commonly used collectives in such a way that they incur low latency and high timing predictability making them suitable for balanced parallelization of scalable high-performance systems and real-time systems alike. Experimental results on a 64 core hardware platform show that our collectives can significantly reduce communication times by up to 95% for single packet messages and up to 98% for longer messages with superior performance for sometimes all message sizes and sometimes only small message sizes depending on the group primitive. In addition, our communication primitives have significantly lower variance than prior approaches, thereby providing more balanced parallel execution progress and better real-time predictability.

Efficient Collective Communication for Multi-core NOC Interconnects

by
Karthik Yagna

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Networking

Raleigh, North Carolina

2013

APPROVED BY:

Dr. Huiyang Zhou	Dr. Yan Solihin

Dr. Frank Mueller
Chair of Advisory Committee

DEDICATION

To my family.

BIOGRAPHY

Karthik Yagna was born in Bangalore, India, on July 18th, 1985 and was raised in a wonderful family. He completed his schooling in Bangalore, India and graduated in 2004. He studied Telecommunication Engineering at R.V College of Engineering receiving a Bachelor degree in 2007. After completing college, he worked as a Software Engineer in the field of Computer Networks and Real-Time Embedded system for over 4 years. He has been part of various teams at Nortel Networks and Cisco Systems. With a goal to specialize in the field of Computer Networks and embedded systems, he decided to pursue his Masters at North Carolina State University in 2011. He has been part of the System Research lab since fall 2012 and focuses on NoC run-time system design. He will be joining Riverbed Technologies after his Masters to continue the journey.

ACKNOWLEDGEMENTS

I would like to thank Dr. Frank Mueller, for his invaluable guidance throughout my thesis work. Your open door policy and availability always encouraged discussions. And our discussions have always helped me approach the problems with a fresh perspective. Over the last two semesters, I was able to have a complete research experience : working on challenging problems on next generation platforms, attending conferences, paper submissions and even giving a talk. I am forever indebted for providing me with such an opportunity. Thank you again.

I would also like to thank Dr. Huiyang Zhou and Dr. Yan Solihin for serving on my thesis committee.

Sandhya, thank you for your patience and constant encouragement. You believed in me even when I doubted myself. You have always been there to cheer me up and give me company on those long difficult days. You are a wonderful partner, looking forward to the rest of our journey.

Mom and Dad, thank you for support and putting up with me through this. I am forever indebted to my Uncle for all his help. Shreyas, for always being there and making this Master's program a walk in the park. Without all you wonderful people I would have never lived my dream.

Chris Zimmer, thank you for your constant help and guidance. Without your awesome code and setup this thesis would have never been possible. You are a wonderful person and always wish you the best.

My crazy roommates Varun, Raj and Prashanth, thank you for making every day fun. I would not have been sane without the constant discussion on ridiculous things.

My fellow lab mates, Nishanth, Srinath, Araash, David, Srinivas, Amir and James, thank you for making the lab a fun place to be.

Finally, I would like to thank my friends, Shireesh, Suman, Pranam, Ranjitha and Kishore for the countless fun filled experiences.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Network-On-Chip Architectures

The system architecture has been constantly evolving to meet the computing needs. Initially, the clock frequency of uni-processor architecture was scaled to make the system faster. However, the combined pressures from increased power consumption and the diminishing performance returns led to the adoption of multi-core processor architectures [14]. Currently, multi-core architectures are widely used in both general-purpose computing chips and application-specific Systems-on-Chip (SoC). These multi-core architectures mainly use bus or point-to-point interconnects for information exchange between the cores. This approach has kept the system design simple, but has resulted in overheads due to increased contention over the interconnect. In systems with lesser number of cores, this overhead is small and is offset by the improved performance of using multiple cores.

Over the past several years, the number of cores has been increasing and this trend is expected to continue. As the number of cores increases, the contention over the interconnect results in significant performance degradation. This has motivated the design of scalable and high-bandwidth interconnects and memory layouts [28]. Inspired by the traditional networks, data switching and packet routing mechanisms was introduced into on-chip communications. Such on-chip networks have routers/switches at every node. The nodes are connected to their neighbors via short local interconnects. This is illustrated in Figure 1.1. The use of routers allows the connected nodes to access the bus immediately without arbitration in most cases. The routers handle the delivery of data from source to destination according to the chosen switching protocols and routing policies.

NoC architectures provide several key benefits. NoC provides greater flexibility in laying out interconnects. NoC topologies includes rings, mesh, torus, and trees, each with its own benefits. 9-core IBM Cell [19] uses two packet-switched rings. 8-core Sun Niagara [21] uses a crossbar

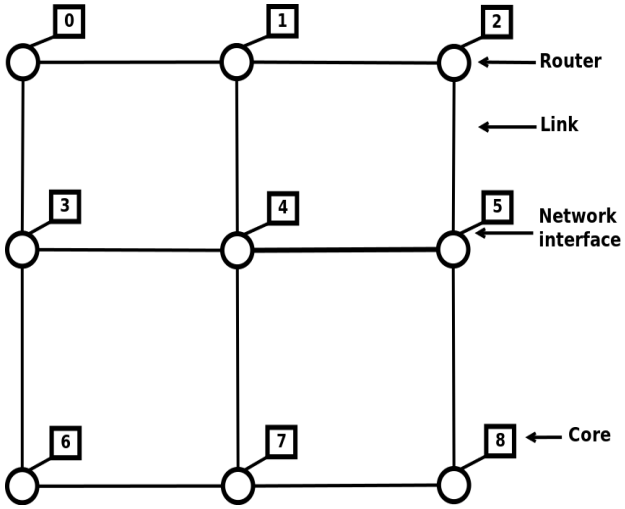

Figure 1.1: Network-on-Chip Architecture

interconnect. 64-core Tilera TILE64 [8] uses packet-switched meshes. Figure 1.2 shows three commonly used NoC topologies. NoCs are expected to be less non-deterministic [25] because of their regular topology. NoC topologies can reduce the complexity of designing wires for predictable speed, power and reliability. NoCs can decouple the computation and communication, making the communication services available transparently to the cores. This also makes the system modular and reusable via standard interfaces [13, 10]. NoCs offer higher bandwidth and parallel communication opportunity making them highly scalable [9, 18].

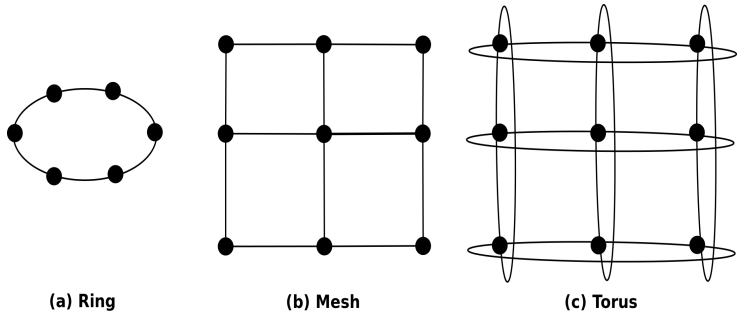

Figure 1.2: Common NoC Topologies

On the downside, NoC accesses results in non-uniform latencies depending on the number of hops between source and destination nodes. More importantly, they suffer from contention-based delay at switching level.

1.2 Wormhole Routing

In wormhole routing [27], each packet is divided into a number of fixed size flits. Each router has buffer and physical channels at flit level instead of packet. The flit is the smallest unit on which flow control is performed. A packet is divided into a header flit, body flit(s) and a tail flit. The header flit has the routing information which is used to route the flits from source to destination. The router use cut-through flow control, allowing flits to move on to the next router as soon as there is sufficient buffering for this flit. Figure 1.3 illustrates a packet decomposition and transmission.

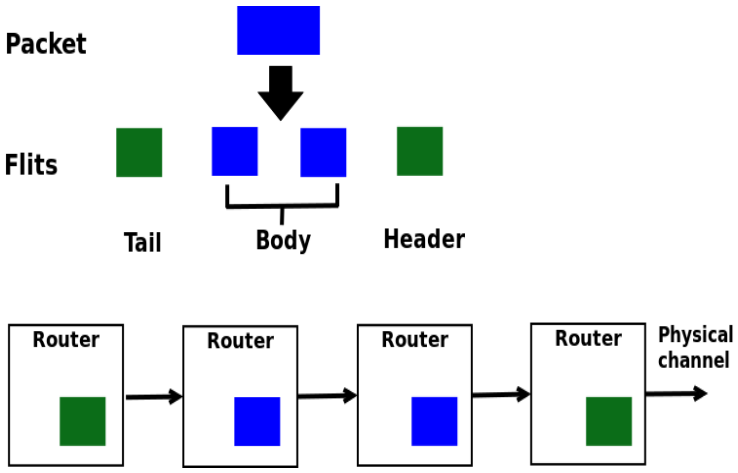

Figure 1.3: Network-on-Chip Architecture

While wormhole routing results in low buffer cost, low network latency and efficient buffer usage, it makes inefficient use of link bandwidth. This is because a link is held for the duration of a packet's lifetime in the router. When a packet is blocked, all the physical links held by that packet are left idle. Other packets queued behind the blocked packet are unable to use the idle physical link reducing the overall throughput. This effect is called chain-blocking. Chain-blocking coupled with non-uniform latency inherent in NoC topology makes the design of communication between nodes critical to exploit the performance benefits provided by the NoCs.

1.3 XY dimension ordered routing

Dimension ordered routing is widely used due to its simplicity. XY dimension ordered routing in a two-dimensional topology such as mesh in Figure 1.4, sends packets along the X-dimension first, followed by the Y-dimension. A packet traveling from (0,0) to (2,1) Will first traverse two

hops along X-dimension, arriving at (2,0), before traversing one hop along Y-dimension to reach its destination. XY dimension ordered routing is an example of deterministic routing algorithm. All messages from node A to node B will traverse through the same path. XY dimension ordered routing is also deadlock-free as there is turn restriction preventing going from Y link to X link. This ensures that there are no cycles and hence, no deadlocks.

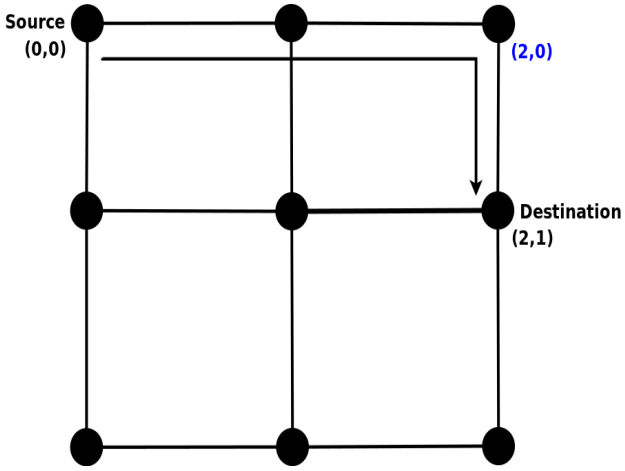

Figure 1.4: XY Dimension Order Routing

1.4 Collective communication

Multi-core platforms typically use message passing to communicate with each other since it is scalable and more efficient than using shared memory. For computation-intensive tasks, parallel applications are typically employed which leverage underlying parallel architecture. These parallel applications employ multiple co-operating and communicating processes to speed up the computation. Communication operations may be either point-to-point, which involves single source and a single destination, or collective, in which more than two processes participate. Collective operation is executed by having all processes in the group call the communication routine with matching parameters.

Figure 1.5 depicts examples of collective operations for a group of four processes. The Alltoall collective results in all the tasks in the group to exchange messages with each other. A barrier synchronizes a group of tasks. Each task, when reaching the barrier call, blocks until all tasks in the group reach the same barrier call. A broadcast sends a message from the process with rank "root" to all other processes in the group. Reduce applies a reduction operation on all tasks in the group and relays the result to one task. AllReduce combines values from all processes

(reduce) and distributes the result back to all processes (broadcast).

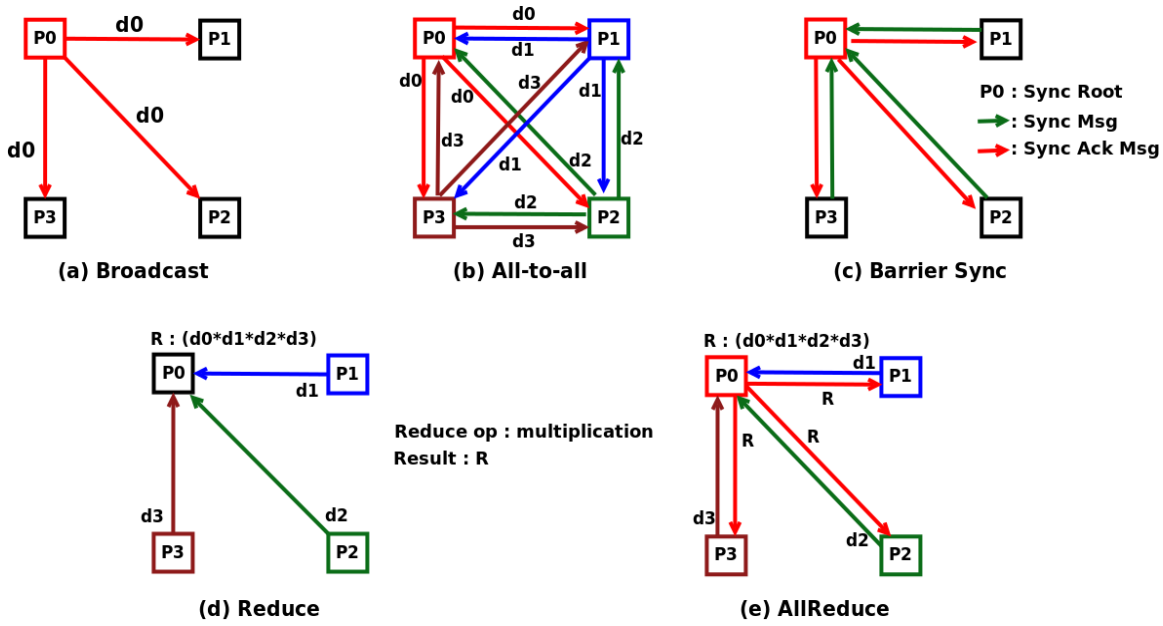

Figure 1.5: Collective operations among four processes

Collective communication operations are particularly important to scientific computing, where large data arrays are typically partitioned and distributed over different nodes. In such applications, nodes use collective operations to broadcast, gather and exchange data, to synchronize with one another at specific points in program, and to perform global compute operations on distributed data. Collective operations are used in numerous sorting, search and graph algorithms [22]. Collectives are also used in variety of matrix-related algorithms and parallel numerical algorithms. The importance of collective operation has resulted in their inclusion and standardization in Message Passing Interface (MPI) [15]. Table 1.1 summarizes the typical use of collective operations.

Table 1.1: Typical usage of collective communication primitives

Category	Primitive	Description
data movement	broadcast	one task sends message to all other tasks
	alltoall	every task sends message to every other task
process control	barrier	all tasks must reach point before any can proceed
global operation	reduce	perform global operation on distributed data
	allreduce	reduce and broadcast result to all tasks

Collective communication operations may involve many messages and may result in exis-

tence of concurrent messages in the interconnect network. These messages may simultaneously require the use of a particular link, resulting in channel contention. The channel contention among the messages may be exacerbated by the use wormhole routing due to chain-blocking. This problem increases with the increase in number of cores participating in the collective communication. Therefore, for massive multi-core platforms with NoC architecture employing wormhole routing efficient design of collective communication becomes critical for parallel applications.

1.5 Motivation

Massive multi-core platforms with NoC architectures are starting to penetrate high-performance systems, three-tier servers, network processing and embedded/real-time systems. These architectures provide a significant advancement due to an abundance of cores. This allows a large number of cooperating tasks to be scheduled together. These tasks can employ group communication via message passing over the NoC to achieve scalability and reduced latency.

However, poor collective communication implementations can result in increased and highly variant latency due to NoC contention resulting in loss of predictability and imbalance in execution progress across cores. Consider the case where tasks on different cores are performing an all-to-all communication using message passing. One way to implement all-to-all is to have one task send its message to all other tasks, followed by the next one and so on. This implementation is not efficient and can be improved by allowing multiple partners to communicate in each round. Yet, such an optimization may lead to contention. For example, consider 9 cores taking part in all-to-all communication as in Figure 1.6. The task on core 3 is trying to send to the task on core 8, and the task on core 4 is trying to send to the task on core 2. This results in 2 messages, one from $3 \rightarrow 8$ and another from $4 \rightarrow 2$. When sent at the same time, contention on link $4 \rightarrow 5$ results in a delay for one of these messages due to arbitration within the NoC hardware routers. As a result, sending tasks experience highly variable latencies. The effect shown in this example is amplified with increasing NoC mesh sizes. Such situations can be avoided using intelligent scheduling of each round of message exchanges.

Additionally, implementations that do not leverage the underlying NoC capabilities result in under utilization of the NoC hardware. Typically, NoC architectures provide multiple message queues and networks [4, 5, 38, 1]. On the TilePro64 [5], there are five distinct message queues and two distinct networks available for users. One of them is the User Dynamic Network (UDN), and another is the Static Network (SN), both of which are freely programmable (in contrast to the remaining networks). UDN uses dynamic routing to forward messages from a source core to a destination core. SN, in contrast, uses statically configured routes to forward packets received on each link. SN is faster than UDN in terms of packet forwarding speed, but is difficult to

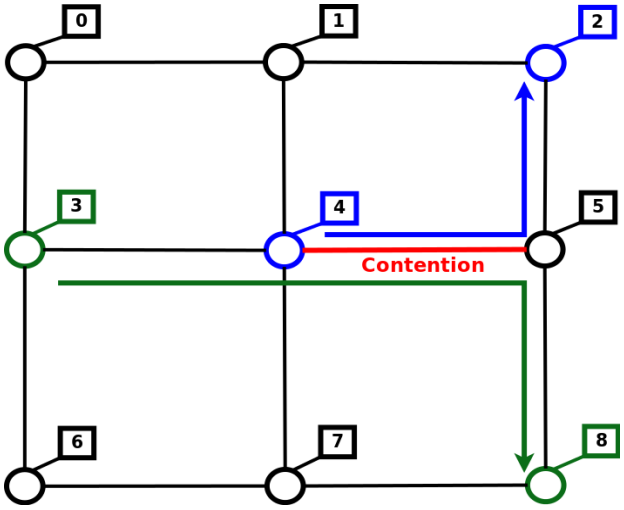

Figure 1.6: NoC Contention

program and has route setup overhead. Hence, UDN is used for all core-to-core communication purposes, leaving SN unused. Implementations that can leverage such unused hardware features can intelligently extract additional hardware performance.

1.6 Our Approach

This work contributes the design and implementation of collective communication for large core counts utilizing 2D mesh NoC architectures. In our implementation, we employ efficient algorithms to reduce communication latency and exploit advanced NoC hardware features to provide better performance. We furthermore ensure that communication uses contention-free paths and that no deadlock may occur. We have implemented five commonly used collective communication primitives, namely Barrier, Broadcast, Reduce, AllReduce and Alltoall [15].

We have used different approaches for each collective communication primitive to demonstrate that NoC-based systems support reliable timing under reduced latency. Our implementation of Barrier, Broadcast and Reduce uses a communication tree in which the cores are arranged as nodes and share a parent-child relationship. The communication tree is used to send messages to/from the root. The Barrier and Reduce implementations utilize the UDN, whereas Broadcast uses the SN. Our implementation of Alltoall uses a bottom-up approach in which the communication proceeds from smaller segments to larger segments, but it does not require dividing the grid into smaller sub-meshes [33]. Other approaches require either dynamic route calculations or offline pre-calculations to store large routing tables [12]. In contrast, our implementation exploits simple pattern-based communication, common in MPI [15] run-time system implementations, to send messages concurrently, yet without contention, to reduce com-

munication latency. This neither requires dynamic computation of a routing schedule nor incurs scheduling overhead or memoization of large routing tables.

Our implementation uses message passing over the NoC of a TilePro64 but is generic enough to be adopted to any 2D mesh based NoC architecture. Most significantly, our design generalizes to arbitrary 2D NoCs, and while prior related work generally assumed ideal symmetry with wrap-around links on the 2D boundaries, our work addresses realistic 2D meshes without wrap-around, such as present in contemporary NoC hardware designs [4, 5, 38, 1, 3].

1.7 Hypothesis

With the increasing number of cores, NoC architectures become a key factor in maintaining higher processing capability, flexibility and scalability of computing system. In order to extract maximum performance in such systems, we need to address their predictability challenges. The key factor contributing to this is NoC contention. Eliminating NoC contention becomes particularly challenging for collective operations. Since, most scientific applications use collective operations, implementing highly efficient and predictable collective communication becomes critical. Current collective communication implementations are either inefficient or very complex to implement. We aim to address this challenge in this thesis. The hypothesis of this thesis is :

Scalable contention-free collective communication results in better performance and predictability than collectives with contention on massive multi-core NoC platforms without adding significant complexity and contributes to balanced parallel execution benefiting both HPC applications and real-time systems.

1.8 Contributions

Our contributions are as follows :

- We show that NoC-based systems can support reliable timing under reduced latency.
- We provide different approaches that can be used for collective communication implementation on NoC-based systems.
- We provide an implementation of commonly used collectives on the Tilera TilePro64 hardware platform. This implementation is generic and can be easily extend to any 2D mesh based NoC platform.

We used micro-benchmarks and NAS Parallel Benchmarks to compare the performance of our implementation against OperaMPI [20], a reference MPI implementation for the Tilera platform. Experimental results on the TilePro hardware platform show that our implementation

has lower latencies and lower timing variability than prior work. Performance improvements of up to 95% are observed in communication for single packet messages with significantly high timing predictability, which supports more balanced execution progress for high-performance computing (HPC) and helps to meet deadlines in real-time applications.

Chapter 2

Background

In this work, we focus on 2D mesh-based NoC architectures. We have designed and implemented efficient group communication on Tilera's TilePro64 NoC platform. We used OperaMPI for comparing the performance of our implementation. This section provides a brief overview of the TilePro64, NAS parallel benchmark and high level implementation details of collectives in OperaMPI.

2.1 TilePro64

2.1.1 Architecture overview

The TilePro64 is a multi-core processor manufactured by Tilera. It consists of 64 programmable compute engines (each referred to as a tile), connected by means of multiple two-dimensional mesh networks. Each tile is a powerful, full-featured computing system that can independently run an entire operating system, such as SMP Linux. It implements a 32-bit integer processor engine utilizing a three-way Very Long Instruction Word (VLIW) architecture with its own program counter (PC), cache, and DMA subsystem. An individual tile is capable of executing up to three operations per cycle. Each tile in the two-dimensional array connects to other tiles using multiple mesh networks implemented by the network routers in each tile. The Tile Processor architecture is scalable and provides high bandwidth and extremely low latency communication among tiles. Each tile in a TilePro64 operates at 700 MHz. The TilePro64 does not support native floating point operation.

2.1.2 Inter-Tile networks

Tile Processor provides a set of hardware networks for sending messages between cores. These include the I/O Dynamic Network (IDN), used to communicate with I/O devices; the User

Dynamic Network (UDN), used for user space messages; and the Static Network (SN), which can transmit individual words between user space tasks running on adjacent cores. Most applications use only the UDN because it is available to user programs and more flexible than the static network.

These networks transmit packets across a mesh using XY dimension ordered routing. At the destination, the packet data words are stored in a demux FIFO queue with a capacity of 118 words. Each network packet contains 1 to 128 data words. Cores inject packets into the network by writing words to special registers. When data arrives at the destinations demux buffer, it is routed to one of four demux queues. The receiving core then reads the incoming data by inspecting one of four registers, each mapped to a different demux queue. By using these networks the data is sent directly from registers on one tile, across the network, to registers on another tile, without having to go through the cache subsystem (which can take 10 cycles on each tile) for improved performance.

2.2 OperaMPI

2.2.1 Overview

OperaMPI is an implementation of the MPI 1.2 specification for the Tilera platform. It is layered over Tileras iLib, an inter-tile communication library that utilizes the UDN NoC network. The iLib library is vendor-supplied software and allows developers to easily take advantage of many features provided by the Tilera architecture.

OperaMPI was evaluated using MPI benchmarks, namely the IBM test suite, the Intel test suite, the MPICH test suite and the SPEC MPI. The results show that for sending and receiving small sized messages the implementation has a latency of about 30μs with a cold instruction cache and about 6.8μs with a warm instruction cache. As the data size increases, the initial overhead is amortized and the data transfer time per word reduces. Just like any other implementation, OperaMPI suffers from communication overhead. This consists of header generation and processing overhead, cache miss cost, a lower bound on the overhead of sending one word per cycle, iLib overhead and MPI overhead.

2.2.2 Collective implementation

Broadcast

OperaMPI implements Broadcast using a tree-like communication pattern, where the root task initiates the broadcast by sending the message to another task. The two tasks send the message to another two tasks. This transitive distribution of messages continues and eventually termi-

nates after $log(N)$ steps, where N is number of tasks. Figure 2.1 shows this procedure for 32 tasks.

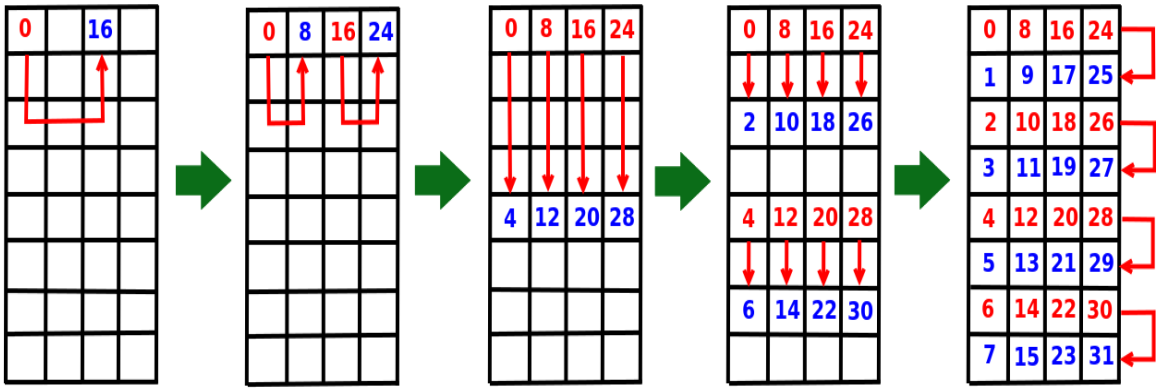

Figure 2.1: OperaMPI Broadcast Example

Reduce and AllReduce

Reduce is implemented using a tree-like communication pattern if the reduction operation is associative. This implementation is effectively the inverse of broadcast. For non-associative reduction operations serial communication is used, wherein each task sends to root tasks in a synchronized fashion. The reduction operation (sum) for 32 tasks is shown in Figure 2.2.

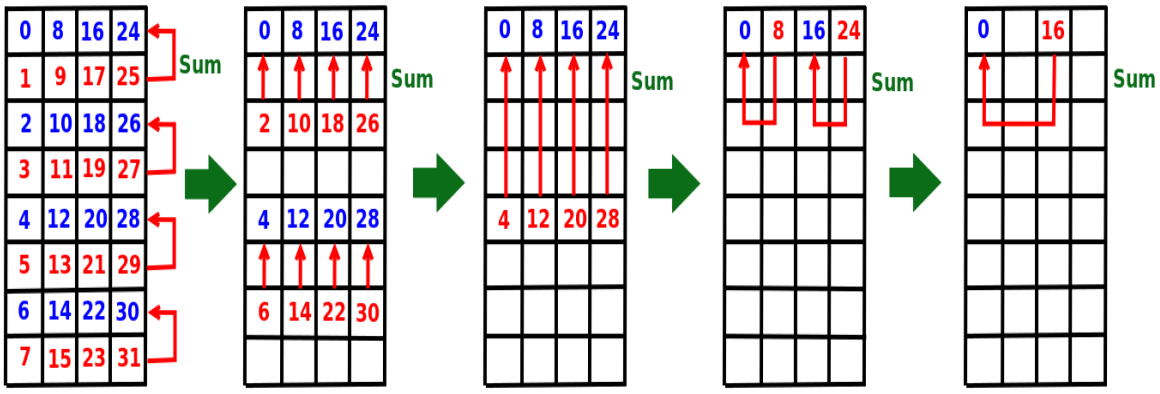

Figure 2.2: OperaMPI Reduction Example

Allreduce is implemented as an extension of the Reduce collective. It consists of a reduction operation followed by a broadcast. The number of cycles for the Reduce collective is much larger than the broadcast since it involves more processing such as an element-wise reduction

operation.

Alltoall

OperaMPI's Alltoall implementation is split into N-1 stages, where N is the total number of tasks. At each stage, one task takes a turn to send to a partner. Depending on the message size, the implementation uses different communication algorithms. If the message size exceeds 1Kilo Word (KW), then half the tiles send to the other half. If the message size is less than 1KW, tasks use a non-blocking send followed by a blocking receive. While their setup is subject to contention to create a virtual channel, transmission proceeds without contention once a channel has been created.

2.3 NAS Parallel Benchmarks

The NAS parallel benchmarks (NPB) [6] were developed at the NASA Ames research center to evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications and originally consist of five kernels and three pseudo-applications. The benchmark suite has been extended to include new benchmarks for unstructured adaptive meshes, parallel I/O, multi-zone applications, and computational grids.

Table 2.1: Communication Characteristics of NPB

Benchmark	Alltoall	Alltoallv	Barrier	Broadcast	Allreduce	Reduce	Send	Isend
EP	-	-	1	-	4	-	-	-
CG	-	-	1	-	-	1	10	-
MG	-	-	9	6	6	1	12	-
FT	3	-	2	2	-	1	-	-
IS	1	1	-	-	1	5	1	-
LU	-	-	1	9	6	-	12	-
BT	-	-	2	3	2	-	-	12
SP	-	-	2	3	2	1	-	-

The original eight benchmarks specified in NPB 1 mimic the computation and data movement in CFD applications:

- Five kernels
  - IS - Integer Sort, random memory access

  - EP - Embarrassingly Parallel
  - CG - Conjugate Gradient, irregular memory access and communication
  - MG - Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive
  - FT - discrete 3D fast Fourier Transform, all-to-all communication

- Three pseudo applications
  - BT - Block Tri-diagonal solver
  - SP - Scalar Penta-diagonal solver
  - LU - Lower-Upper Gauss-Seidel solver

Table 2.1 shows the communication characteristics of different benchmarks. A detailed description of the benchmarks can be found in [2].

Chapter 3

NoCMsg Collectives

3.1 Design

Our work assumes a generic, generalized 2D mesh NoC switching architecture similar to existing fabricated designs with high core counts [4, 5, 38, 3]. Each core is composed of a compute core, network switch, and local caches. The network switch uses XY dimension-ordered routing to forward messages.

3.1.1 NoC Architecture

NoC architectures use the network-on-chip to replace the conventional system bus or other topologies of connecting cores. This means that all memory, messaging, and IO communication occur over the NoC, often through physically separate networks to reduce contention. Most NoC architectures feature multiple networks for this purpose. Adapteva [1] features three networks while Tilera [5] features five/six networks in their TilePro/GX, respectively. In this work, we focus on building group communication over the messaging networks.

3.1.2 NoC Message Layer

Our implementation provides an MPI-style message passing interface on top of the NoC. This facilitates basic point-to-point communication and supports our group communication. The NoC message layer implementation optionally provides flow control support. In our design, we turn off flow control when not required by program logic to further improve performance.

3.1.3 Group Communication Primitives

The key ideas behind our design of group communication primitives are :

1. Reduce contention in the NoC

2. Exploit pattern-based communication to exchange messages concurrently

3. Reduce the number of messages by aggregation

4. Leverage hardware features to improve performance

We have used different approaches for each group communication primitive to demonstrate the ways a NoC-based system can support timing reliability and reduced latency. These approaches are summarized in Table 3.1.

Table 3.1: Summary : Design approaches

Collective	Approach
Alltoall, Alltoallv	pattern-based communication, contention-free exchange using UDN
Barrier	k-ary tree-based, uses small synchronization messages, uses UDN
Broadcast	tree-based, tree mapped onto NoC in contention-free manner, uses SN
Reduce	tree-based, tree mapped onto NoC in contention-free manner, uses UDN
AllReduce	Extension of other collectives: Reduce followed by Broadcast, uses UDN and SN

Alltoall

The Alltoall collective results in all the tasks in the group to exchange messages with each other. The prototype for this collective is as follows:

```
int NoCMsg_Alltoall(void *sendbuf, int sendcount,
                    NoCMsg_Datatype sendtype,
                    void *recvbuf, int recvcount,
                    NoCMsg_Datatype recvtype,
                    NoCMsg_Comm comm)
```

In our design, we exploit pattern-based communication to concurrently exchange messages between partners. The entire exchange is split into multiple rounds. In each round, a subset of tasks exchanges messages using Manhattan-path (dimension-ordered) routing [**?**]. The tasks in each round are scheduled in such a way that they do not result in link contention. In each round, the number of hops the message is forwarded to is incremented until all the tasks are covered.

Barrier

A barrier synchronizes a group of tasks. Each task, when reaching the barrier call, blocks until all tasks in the group reach the same barrier call. The prototype for this collective is as follows:

```
int NoCMsg_Barrier(NoCMsg_Comm comm)
```

In order to provide scalable barriers, we designed tree-based barriers that distribute the work evenly among nodes. This also helps minimize the cycle differences upon barrier completion. Our design utilizes rooted k-ary trees to this end, where k is configurable. On the TilePro64, $k = 3$ provides optimal performance in experiments.

Broadcast

A broadcast sends a message from the process with rank "root" to all other processes in the group. The prototype for this collective is as follows:

```
int NoCMsg_Bcast(void* buffer, int count,
                 NoCMsg_Datatype sendtype,
                 int root, NoCMsg_Comm comm)
```

Our design utilizes the SN to implement broadcasts. We designed a tree-based broadcast rooted at the task where the broadcast message originates. Tree branches are mapped onto the NoC in a contention-free manner. The static route of each task is configured inside the broadcast primitive such that the message from the root flows to each leaf task. To minimize the overhead of route configuration, our design requires only a single route configuration per task, again using contention-free paths.

Reduce

This collective applies a reduction operation on all tasks in the group and relays the result to one task. The prototype for this collective is as follows:

```
int NoCMsg_Reduce(void *sendbuf, void *recvbuf,
                  int count,
                  NoCMsg_Datatype datatype,
                  NoCMsg_Op op, int root,
                  NoCMsg_Comm comm)
```

We designed our reduce collective similar to the barrier. The reduction operation is performed along the tree. Each task receives values from its children and performs a partial reduction. Tasks then send their partial result toward the root. The root will reduce partial results to obtain the final result.

AllReduce

This collective combines values from all processes (reduce) and distributes the result back to all processes (broadcast). The prototype for this collective is as follows:

```
int NoCMsg_AllReduce(void *sendbuf, void *recvbuf,
                     int count,
                     NoCMsg_Datatype datatype,
                     NoCMsg_Op op,
                     NoCMsg_Comm comm)
```

AllReduce is is designed as an extension to Reduce. The AllReduce consists of a reduce followed by a broadcast.

Alltoallv

The Alltoallv collective sends data from each tasks to all (other) tasks; each task may send a different amount of data and provide displacements for the input and output data. The prototype for this collective is as follows:

```
int NoCMsg_Alltoallv(void *sendbuf, int sendcount,
                     int *senddisplacement,
                     NoCMsg_Datatype sendtype,
                     void *recvbuf, int recvcount,
                     int *recvdisplacement,
                     NoCMsg_Datatype recvtype,
                     NoCMsg_Comm comm)
```

Alltoallv is designed as an extension of Alltoall.

3.2 Implementation

This section provides details on the implementation, called NoCMsg, of each group communication primitive. Our implementation of these collectives have an MPI-like API for easy usability.

We implemented the group communication on the Tilera TilePro64. Nonetheless, our implementation is generic and can be extended to any 2D mesh NoC architectures.

3.2.1 Alltoall and Alltoallv

Alltoall/Alltoallv are the most demanding collectives in terms of network contention, yet they provide opportunities for flow-control elimination within their implementation. Based on the particular internal send/receive orders in these collectives, it is possible to guarantee flow-control free communication for transfers between each pair of cores. Further optimization is provided by employing pattern-based communication, which allows several sets of tasks to exchange messages concurrently without contention. The entire exchange is split into multiple rounds.

The rounds are comprised of (1) direct (2) left and (3) right rounds. The direct round is further split into two sub-rounds. In sub-rounds, each task sends messages only along a straight path to its partner task. Tasks exchange messages along X direction in direct sub-round 1 and along Y direction in direct sub-round 2. In left rounds, each task sends messages along the X direction followed by the Y direction such that their path follows a counter-clockwise direction. In right rounds, each task sends messages along the X direction followed by the Y direction such that their paths follow a clockwise direction. These cases are depicted in Figure 3.1. The XY dimension routing ensures that these directions are maintained consistently.

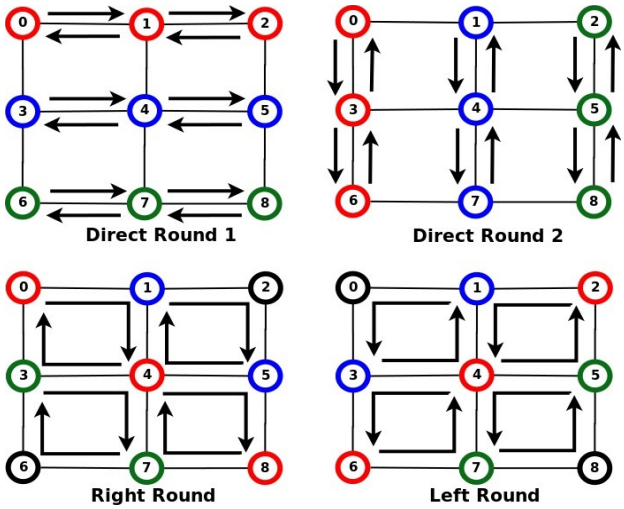

Figure 3.1: Alltoall Rounds

The implementation details are sketched in Algorithm 1. In each round, the number of hops the message is forwarded is incremented until all tasks are covered. To begin, each task starts the direct sub-round one with one hop (lines 5-13). The current column, which can take part in an exchange, is selected by function Select-col (line 7). Each task then compares its column number with the currently selected column. Tasks which are on such columns exchange messages with their neighbors one hop away along the X direction. This is done to ensure that the exchange is free of contention. Once the round has been completed, tasks increment their

hop count and exchange messages with a neighbor two hops away. This is repeated until the entire width of the grid is covered. After an exchange along the X direction has finished, tasks start direct sub-round two by sending messages along the Y direction in a similar fashion (lines 14-23). This set of rounds is followed by a left round and a right round (lines 24-49), thereby covering the entire grid. The logic of the algorithm is depicted in the Figure 3.2. An example of Alltoall round progression is depicted in Figure 3.3. Tasks exchanging messages in each round are highlighted using same color.

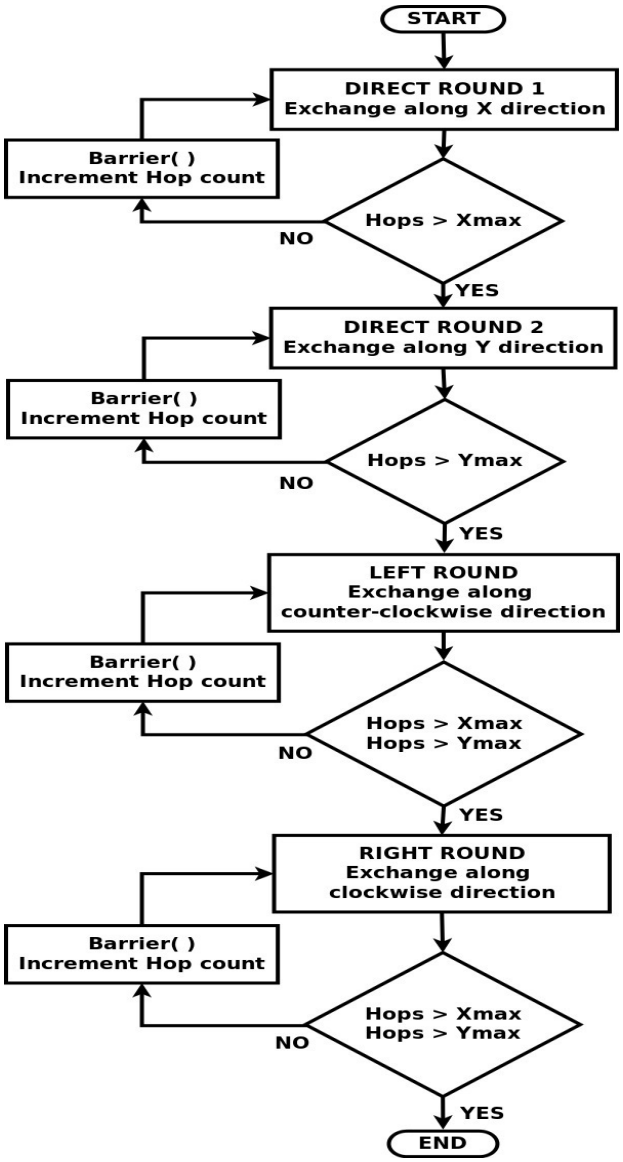

Figure 3.2: Alltoall Algorithm

Algorithm 1 Alltoall

```
1: function NoCMsg-Alltoall
2:     Xmax ← gridwidth
3:     Ymax ← gridheight
4:     for xhops ← 1, Xmax do                          ▷ Direct subround 1 (DR1)
5:         currcol = Select-col(DR1, xhops)             ▷ select column
6:         if mycol == currcol then                     ▷ my column's turn
7:             UDN-xchg(x+xhops, y)
8:             UDN-xchg(x-xhops, y)
9:         end if
10:        Barrier()
11:    end for
12:    for yhops ← 1, Ymax do                          ▷ Direct subround 1 (DR2)
13:        currrow = Select-row(DR2, yhops)             ▷ select row
14:        if myrow == currrow then                     ▷ my row's turn
15:            UDN-xchg(x, y+yhops)
16:            UDN-xchg(x, y-yhops)
17:        end if
18:        Barrier()
19:    end for
20:    for yhops ← 1, Ymax do                          ▷ Left round (LR)
21:        for xhops ← 1, Xmax do
22:            currrow = Select-row(LR, yhops)          ▷ select row
23:            currcol = Select-col(LR, xhops)          ▷ select column
24:            if myrow, mycol == currrow, currcol then
25:                UDN-xchg(x+xhops, y+yhops)
26:                UDN-xchg(x-xhops, y-yhops)
27:            end if
28:            Barrier()
29:        end for
30:    end for
31:    for yhops ← 1, Ymax do                          ▷ Right round (LR)
32:        for xhops ← 1, Xmax do
33:            currrow = Select-row(RR, yhops)          ▷ select row
34:            currcol = Select-col(RR, xhops)          ▷ select column
35:            if myrow, mycol == currrow, currcol then
36:                UDN-xchg(x+xhops, y+yhops)
37:                UDN-xchg(x-xhops, y-yhops)
38:            end if
39:            Barrier()
40:        end for
41:    end for
42: end function
```

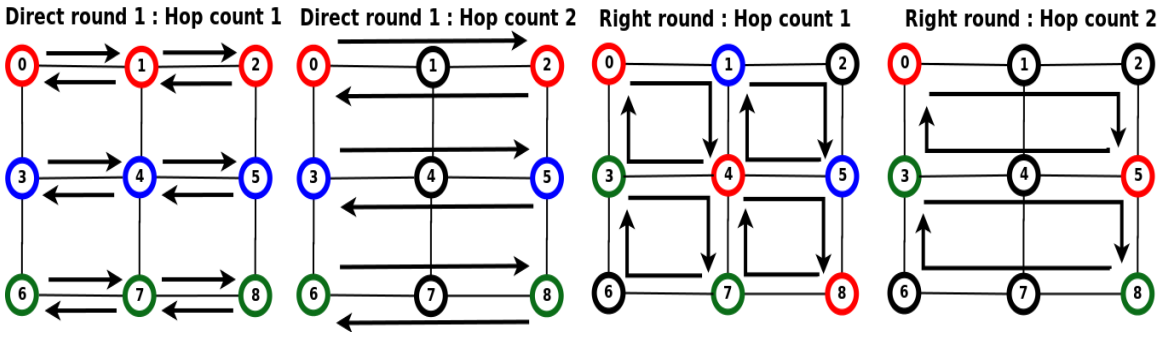

Figure 3.3: Alltoall Rounds Example

3.2.2 Barriers

We utilize a modified 3-ary tree-based barriers that distribute the work evenly among nodes to minimize cycle differences upon barrier completion. The root of this tree is placed in the center of the NoCMsg grid to minimize latency (hops). The tree is constructed as part of the initialization process. The process of synchronization involves the children notifying their parents when they have entered the barrier, up to the root. Once the root has received notifications from all children, it broadcasts a notification back down the tree by replying to its children and exits, as do the children. UDN is used to send and receive synchronization packets and their replies. The implementation details are sketched in Algorithm 2. Figure 3.4 shows an example of barrier tree constructed for 16 cores. The root node is shown in red and its neighboring nodes are shown in blue. The nodes along the root's column, highlighted in green act as secondary roots and will have upto 3 children. Hence, the tree is 3-ary on the interior, 4-ary for the root (to be precise) and of lower degree (2/1/0) for nodes close to the leaves and leaves themselves.

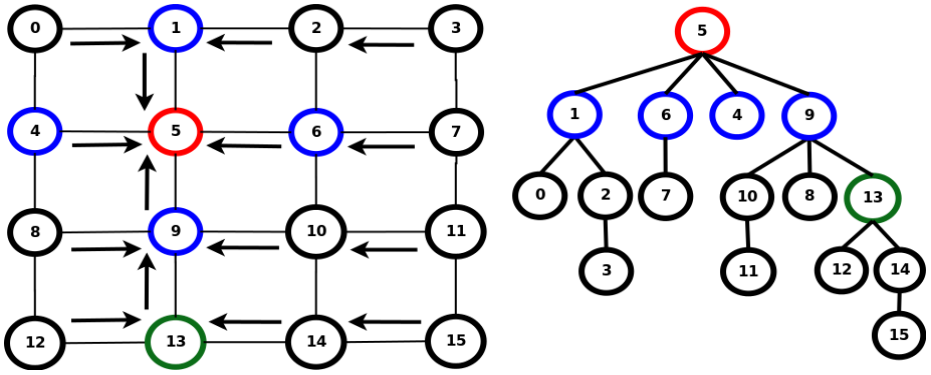

Figure 3.4: Barrier Tree: Modified 3-ary Based

Algorithm 2 Barrier

```
1: function NocMsg-Init
2:     . . . . .
3:     root ← gridcenter                                  ▷ grid center becomes root
4:     k ← 3
5:     Build-barrier-tree(root, k)
6:     . . . . .
7: end function
8:
9: function NoCMsg-Barrier
10:    // Recv sync packet from children
11:    for n ← 0, num − children do
12:        UDN-recv(child)
13:    end for
14:    if myrank ≠ root then
15:        // Send sync packet to parent
16:        UDN-send(parent)
17:        // Recv sync reply packet from parent
18:        UDN-recv(parent)
19:    end if
20:    // Send sync reply packet to children
21:    for n ← 0, num − children do
22:        UDN-send(child)
23:    end for
24: end function
```

Algorithm 3 AllReduce

```
1: function NocMsg-AllReduce
2:     // Perform reduce at rank 0
3:     NoCMsg-Reduce(rank0)
4:     // Broadcast the reduction result
5:     NocMsg-Bcast()
6: end function
```

Flow control is not needed in the barrier as the prerequisite of entering into the barrier is that all outstanding sends/receives of local cores have completed. The synchronization packet is small enough to fit into the output queue, *i.e.*, the core can drop an entire synchronization packet into its output queue. It can subsequently begin a blocking send operation that halts the core's pipeline until synchronization packets become available. This technique significantly reduces synchronization costs when all cores are ready, yet conserves power when they are not.

3.2.3 Broadcast

Our Broadcast implementation uses the SN of the TilePro64. The SN is more intricate to program and suffers from route setup overhead. However, message forwarding incurs zero overhead (due to a static route configuration). Since broadcast has a single sender and multiple receivers, the number of route configurations is low. This was the motivation behind using SN for the broadcast implementation.

Algorithm 4 describes the Broadcast logic. We designed a tree-based algorithm rooted at the task performing the broadcast. Each task determines the root's row and column (line 3 and 4) and invokes SNsetroute, which configures the SN route (line 6). The route setup in the root is such that the message from the core is sent on its available links. All the tasks in the same column as the root have their route configured such that they receive from the root along the Y direction and send the message along other available links. Tasks in other columns receive along one X direction and send the message along the other X link.

For example, let the task with rank 5 initiate a broadcast. Then, its routes are set up to send the message from the core to all the links. The routes of tasks on cores in column one will be set up such that they send out the received message along the X and Y directions. The routes in all the other tasks will be set up in such a way that they will receive and forward along the X direction. This results in a broadcast tree as shown in Figure 3.5. Different nodes have different route setup depending on its relative position to the root node. The root node is highlighted in red. The nodes highlighted in blue have route configuration such that they receive along Y direction and send the message along X direction (East and West). The nodes

highlighted in blue, have route configuration such that they receive along X direction (from West) and send the message along X direction (towards East). All the other nodes have route configuration to receive along X direction.

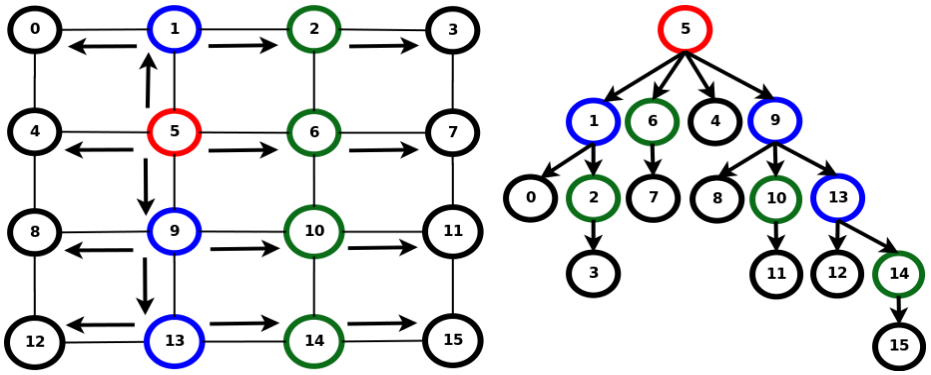

Figure 3.5: Broadcast Tree: Static Routes Configuration

The static route of each task is configured inside the Broadcast call such that the message from the root flows to each leaf task. Our current implementation requires only a single route configuration per task and is contention-free.

Algorithm 4 Broadcast

```
1:  function NoCMsg-Bcast
2:      // Get root's row and col from rank
3:      rootcol = Get-root-Col(root)
4:      rootrow = Get-root-row(root)
5:      // Set SN routes
6:      SN-setroute(x, y, rootrow, rootcol)
7:      if myrank == root then
8:          SN-send( )
9:      else
10:         SN-recv( )
11:     end if
12: end function
```

3.2.4 Reduce and AllReduce

We designed our Reduce collective similar to the barrier. The reduction operation is performed along the tree. Each child task sends its partial result upward toward the root. The root reduces the partial results to obtain the final result. The construction of the reduction tree is different from that of the Barrier. The reduction tree maps to a NoC grid such that the root task becomes the root of the tree. The tasks along its row become first-level children. The tasks in each column become second-level children to the first-level ones.

For example, let rank 5 be the root for the reduction operation. Rank 5 becomes the root of the reduction tree. The tasks along its row become the first-level children (in this case, tasks with rank 4,6 and 7). These first-level children become children of the root. Each column will therefore have a root or a first-level child. All the other tasks become children of the root or first-level children along their column. In the example, rank 5 becomes the root with ranks 1,4,6,7,9 and 13 as its second-level children. Rank 4, a first-level child, will have ranks 0,8 and 12 as children etc. This reduction-tree setup is shown in Figure 3.6.

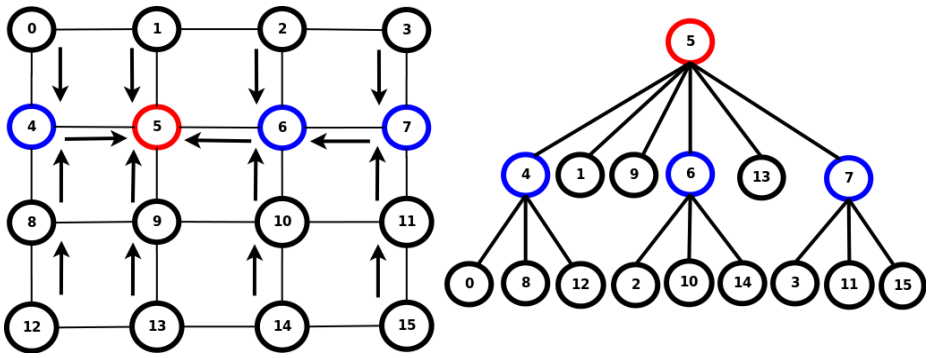

Figure 3.6: Reduction Tree: Setup

A reduction tree constructed in this fashion has two major advantages: (1) The implementation is simple and scalable and (2) the entire reduction takes place in two steps irrespective of the size of the NoC grid. The first step occurs in parallel for the root and its 1st-level children, where they receive and reduce values from their respective 2nd-level children. In the second step, the root will receive partial results from the 1st-level children and perform the reduction operation.

The reduction tree is constructed as part of the Reduce primitive by the Build-reduction-tree function (line 6) shown in Algorithm 5. The function takes as arguments the root's row and column information along with the task's position information (x, y) in the NoC grid. Once the

tree is constructed, child tasks send their values to the root or 1st-level children (line 23-26). Lines 14 to 22 correspond to the actions performed by the 1st-level child tasks. A 1st-level child will receive values from all their 2nd-level children and perform the reduction operation to obtain partial result (line 16-20). They then send the partial result to the root (line 22). The root receives the values from its 2nd-level children along its column and 1st-level children in its row and performs the reduction operation to arrive at the final result (line 7-13).

Algorithm 5 Reduce

```
 1: function NoCMsg-Reduce
 2:     // Get root's row and col from rank
 3:     rootcol = Get-root-col(root)
 4:     rootrow = Get-root-row(root)
 5:     // Build reduction tree
 6:     Build-reduction-tree(x, y, rootrow, rootcol)
 7:     if myrank == root then
 8:         // Root of the reduction tree
 9:         for n ← 0, num − children do
10:             // Recv values from children
11:             val = UDN-recv(child)
12:             res = Perform-reduce-op(val)
13:         end for
14:     else if num − child > 0 then
15:         // 1st-level child
16:         for n ← 0, num − children do
17:             // Recv values from 2nd-level child
18:             val = UDN-recv(child)
19:             partial-res = Perform-reduce-op(val)
20:         end for
21:         // Send partial result to parent
22:         UDN-send(parent, partial − res)
23:     else
24:         // Send value to parent
25:         UDN-send(parent, val)
26:     end if
27: end function
```

AllReduce is an extension of Reduce. It is implemented by performing a Reduce relative to rank 0, followed by a broadcast from rank 0 to all other tasks in the group. The implementation details are sketched in Algorithm 3.

Chapter 4

Experimental Results

We evaluated our group communication using micro benchmarks and NAS parallel benchmarks on the Tilera TilePro64. We compare the performance of our implementation against OperaMPI, an MPI library specific to the Tilera platform.

4.1 Microbenchmarks

Micro-benchmarks have multiple calls to the respective group communication primitive. The number of times the execution time of the primitive must be measured is configurable. In each experiment, we determined the average time elapsed in completing the group communication. The basic template of micro-benchmark is as follows:

```
NoCMsg_Init(int argc, char **argv)
...
count = 0
while(count < MAX_TRIAL)
     NoCMsg_Barrier(NoCMsg_Comm comm)
     NoCMsg_Timer_start(int timer_num)
     NoCMsg_Bcast(void* buffer, int count,
                  NoCMsg_Type datatype, int root,
                  NoCMsg_Comm comm)
     NoCMsg_Timer_stop(int timer_num)
     exec_time = NoCMsg_Timer_read(int timer_num)
     total_exec_time = total_exec_time + exec_time
avg_exec_time = total_exec_time/MAX_TRIAL
...
```

We designed one microbenchmark per collective operation. The timer library returns the time in microsecond resolution. The same microbenchmark was extended to test the behavior under varying message sizes.

4.2 Single packet messages

The benchmark timing results for single packet messages are depicted in Figures 4.1-4.5 for alltoall, reduce, allreduce, barrier and broadcast (in that order). Time on the y-axis is plotted in microseconds for averaged benchmark runs over different number of tasks (equal to cores) in the range from 4..49 for both our NoCMsg implementation and OperaMPI, the reference implementation. (Recall that 64 core runs cannot be conducted since at least two cores are reserved by Tilera's hypervisor for administrative tasks.) Execution time variances for each micro-benchmark for varying number of tasks are shown in Tables 4.1 and 4.2 for NocMsg and OperaMPI, respectively.

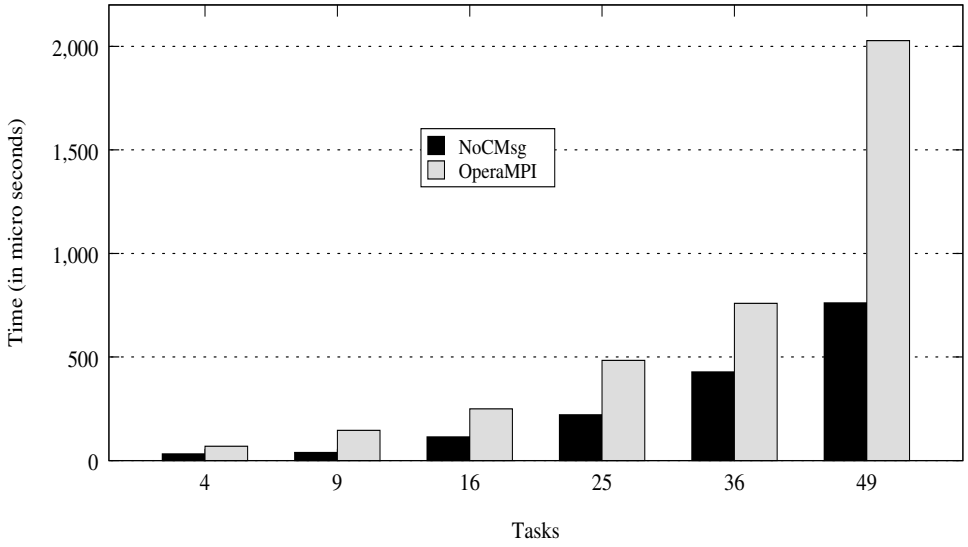

Figure 4.1: Timing Results for Alltoall

We observe that experimental results follow a common trend. As the number of tasks increases, the execution time of group communication increases. In case of Opera, the increase in runtime is significant for larger number of tasks. In comparison, our NoCMsg implementation is highly efficient, and increases in runtime are gradual. Alltoall is the most demanding collective in terms of network contention. Our pattern-based approach effectively eliminates network contention resulting a reduction of execution time by about 62% for the grid size of 7x7. Our

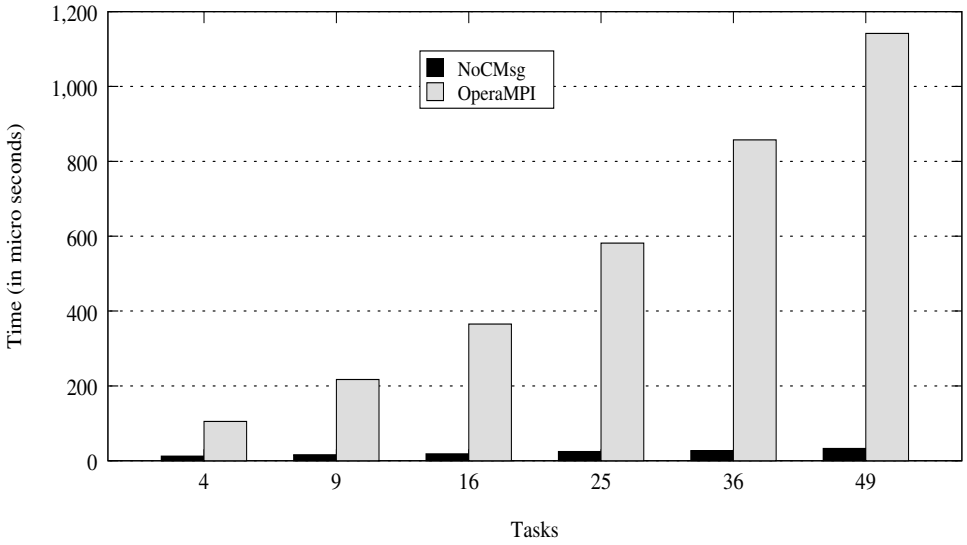

Figure 4.2: Timing Results for Reduce

implementation of Alltoall has a variance ranging from 0.4 to 5.6, depending on the numbers of cores involved in the collective. This variance is several orders of magnitude lower than that of the OperaMPI implementation particularly for larger number of cores.

Barrier and Broadcast are our most efficient collectives with up to 98% reduction in execution time. By mapping the communication pattern onto the NoC in a contention-free manner, our implementation reduced the communication time by up to 95% resulting in reduced execution time. Broadcast uses the SN with a single route setup (to configure the communication tree) and minimal routing overhead. The SN is typically faster than the UDN, which makes Broadcast our most efficient and predictable collective in comparison. Execution time increases only by a factor of 3.5 as the grid size is gradually changed from 2x2 to 7x7 with a variance of less than 0.6 for all cases.

Our implementations of Reduce and Allreduce have 97% and 98% lower execution time, respectively, than the OperaMPI implementation for all tested grid sizes. However, they have larger variance than other collectives. This is due to the two-step reduction employed by the Reduce collective. The root receives partial results from the first-level children in a specific order (increasing order of ranks). If any of them are busy computing the partial result, the overhead for the reduction primitive increases as well. In contrast, if the lower ranks have already calculated their partial result when the root posts a receive, then the root can continue with the reduction primitive without any delay. Overall, our group communication primitives have lower execution time and variance for all grid sizes. The lower variance of our implementation results in better timing predictability making our implementation ideal for real-time applications.

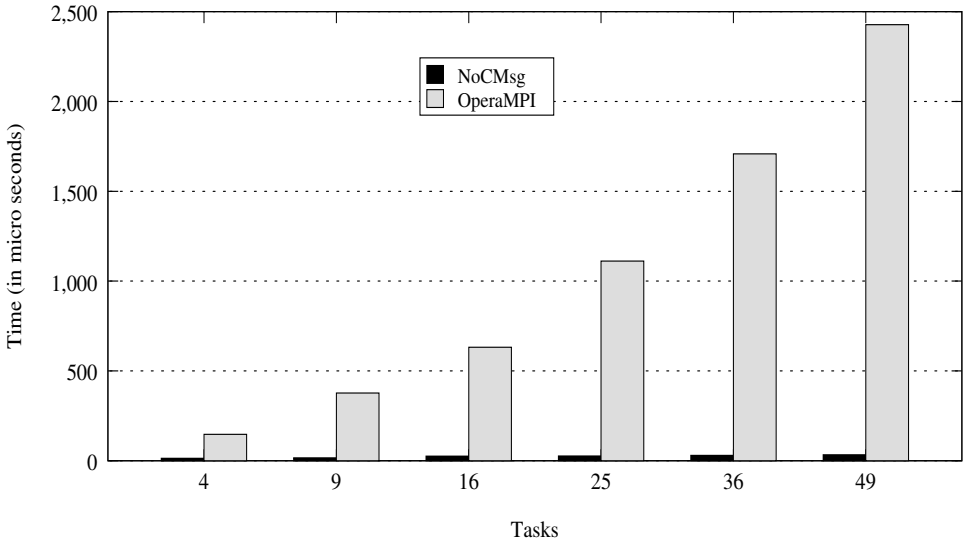

Figure 4.3: Timing Results for AllReduce

Table 4.1: NoCMsg Execution Time Variance

Num tasks	4	9	16	25	36	49
Alltoall	0.7	0.4	0.7	5.6	1.3	1.6
Barrier	0.5	0.8	0.4	1.6	1.1	5.6
Broadcast	0	0	0.2	0.24	0.53	0.12
Reduce	7.96	1.26	2.53	13.1	2.77	4.77
AllReduce	3.96	4.49	12.24	36.77	3.92	4.86

Table 4.2: OperaMPI Execution Time Variance

Num tasks	4	9	16	25	36	49
Alltoall	2.81	983.9	18.2	2276.8	133329.8	622903
Barrier	750.2	302.9	29384.5	1838.2	2910.7	32117
Broadcast	7.3	56.9	259.2	4540.8	3003.7	3869
Reduce	545.26	686.2	21.39	2007.06	9979.96	3430.69
AllReduce	11.14	50.98	49.36	3839.44	5536.16	7517.2

4.3 Varying message sizes

Figures 4.6-4.10 depict the averaged performance for varying message sizes and number of tasks (cores) for both our NoCMsg implementation and OperaMPI, the reference implementation. Notice that execution times are plotted on a logarithmic scale on the y-axis. The solid lines represent execution times for NoCMsg while the dotted lines represent execution times for OperaMPI. The legend further indicates the number of tasks, i.e., key N4 represents NoCMsg

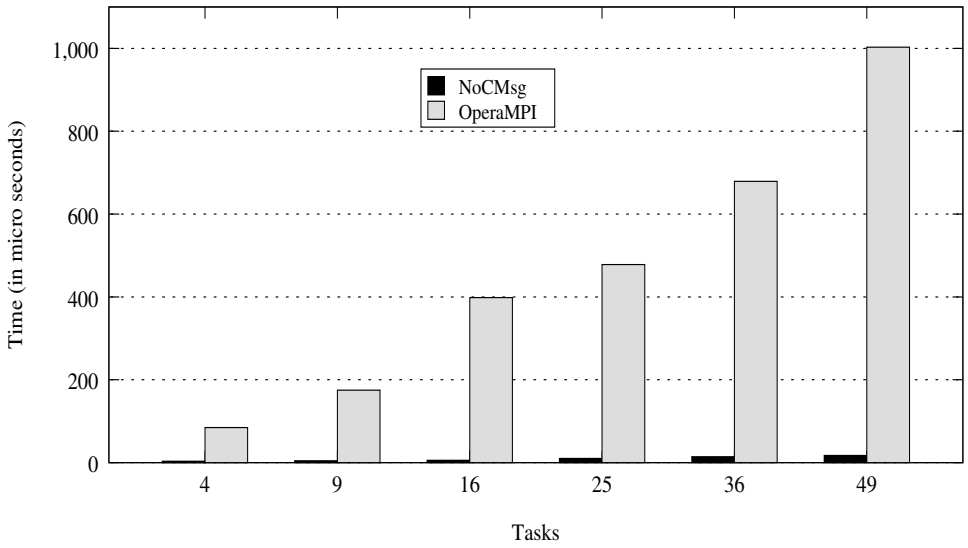

Figure 4.4: Timing Results for Barrier

with 4 tasks while O4 depicts OperaMPI with 4 tasks with the same color coding for identical grid sizes (in the same order as the line graphs). The range of grid sizes ranges from 4 to 49 total number of tasks (cores).

Figure 4.6 shows the execution time of the Alltoall collective for message sizes up to 4KB, which is an inset to Figure 4.7 the latter of which extends to 1MB sizes. The execution and communication times increase with an increase in message size for both NoCMSg and OperaMPI. Our NoCMsg implementation of Alltoall performs very well for small messages with savings between 43%-62% up to a threshold (256 bytes to 4KB depending on message size and number of tasks, see Figure 4.6). Yet, as message sizes increase, performance degrades, and for message sizes greater than this threshold, OperaMPI outperforms NoCMsg (see Figure 4.7). This is because our Alltoall implementation is split into rounds of exchanges followed by barrier synchronization to ensure absence of contention. For large messages, this results in noticeable overhead. OperaMPI's Alltoall implementation is split into N-1 stages, where N is the total number of tasks. At each stage, one task takes a turn to send to a partner. While their setup is subject to contention to create a virtual channel, transmission proceeds without contention once a channel has been created, which provides higher bandwidth for large messages. Yet, prior work has shown that typical applications tend to utilize collectives with very small message payloads [37], which indicates that our NoCMsg covers the critical path for most applications and nicely complements OperaMPI's advantage for large messages.

The timing results for Reduce and Allreduce are shown in Figures 4.8 and 4.9, respectively. The execution time of our implementation is 48%-98% lower than that of OperaMPI for all

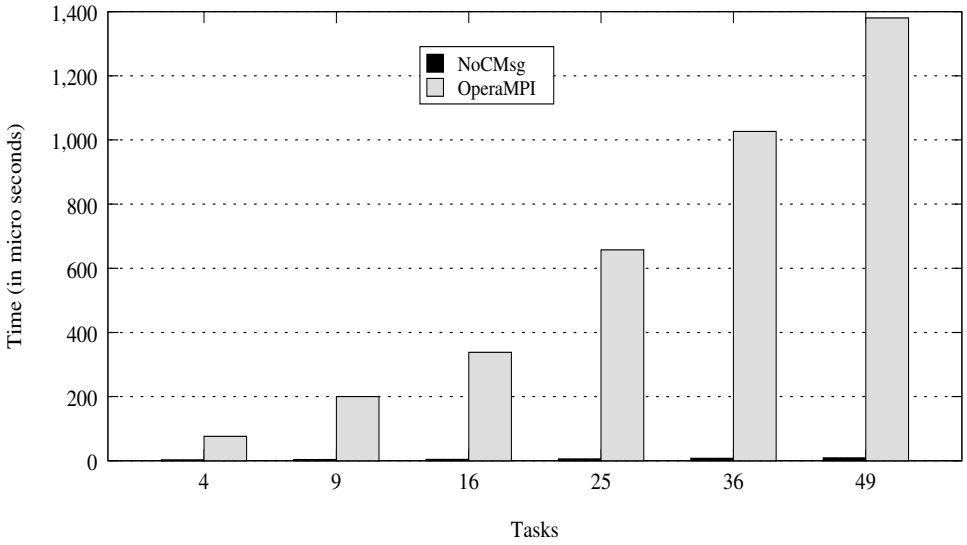

Figure 4.5: Timing Results for Broadcast

message sizes up to 1MB (and beyond). However, the gap gradually decreases. Asymptotically, the performance results of the two implementations approach each other for very large (but, in practice, unrealistic) message sizes. The implementation of Reduce in OperaMPI uses a communication tree but does not map it to the NoC in a contention-free manner. The resulting contention causes larger communication/execution times. The same observation also holds for AllReduce, which is a Reduce followed by a Broadcast. Since the Reduce operation dominates the communication and execution time in AllReduce, its behavior is same as Reduce.

Figure 4.10 represents the execution time of Broadcast for different message sizes. OperaMPI implements Broadcast using a tree-like communication pattern, where the root task initiates the broadcast by sending the message to another task. The two tasks send the message to another two tasks. This transitive distribution of messages continues and eventually terminates after $log(N)$ steps, where N is number of tasks. This communication tree approach is efficient but does not map to the NoC in a contention-free manner. Similar to Reduce, there is always contention resulting in larger communication and execution time. Our Broadcast implementation uses SN unlike OperaMPI, which uses UDN. Routing overhead in SN is lower than that in UDN. This also contributes to better performance and lower execution time. From the NoCMsg curve, we can see that the execution time remains constant for message sizes up to 256 bytes. Beyond 256 bytes, the execution time of NoCMsg Broadcast increases at a higher rate than that of OperaMPI. This continues up to a message size of 128KB, after which the rate of increase in execution time with increase in message size is nearly same for both NoCMsg and OperaMPI. Again, the execution times of the two implementations approach each other asymptotically for

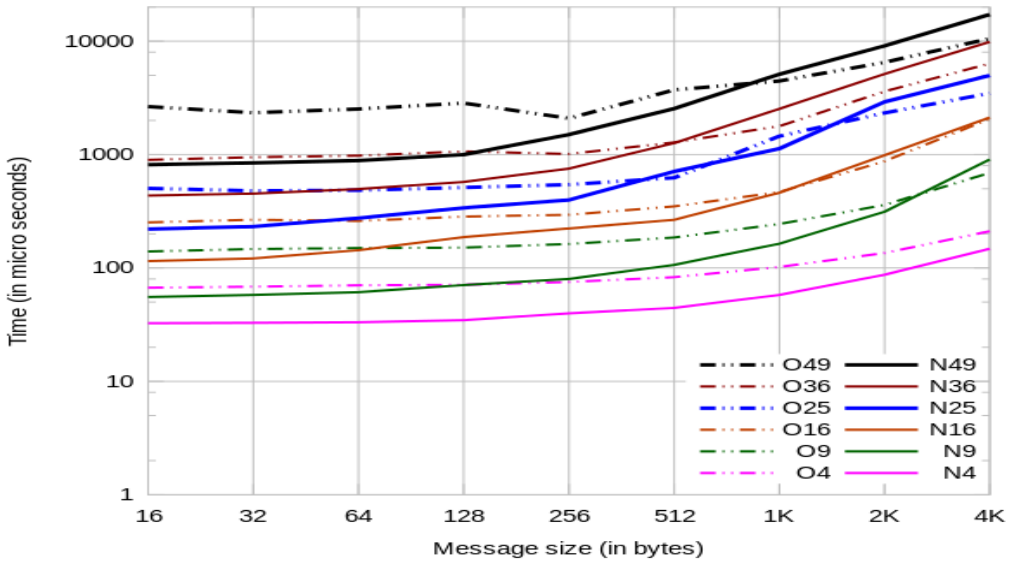

Figure 4.6: Alltoall: Inset for Message Sizes up to 4 KB

very large (but, in practice, unrealistic) message sizes.

Overall, these results show that our NoCMsg implementation is ideal for all / small message sizes depending on the collective primitive. As prior work has indicated, typical MPI applications utilize collectives with very small message payloads [37], and real-time applications follow a similar trend for numerical, actuator-based control systems. This underlines the contribution of our work for HPC and real-time applications alike as NoCMsg provides better performance and timing predictability than prior related work for the common case, and, moreover, for realistic 2D meshes without wrap-around network links at grid boundaries.

4.4 NAS Parallel Benchmarks

We used NPB version 3.3 to evaluate our implementation. NPB by default uses strong scaling, where the input size stays fixed for different number of cooperating cores. We used strong scaling for MG benchmark and for all other benchmarks we used our own weak scaling inputs [16] where the number of keys per core is a fixed size. Weak scaling ensures that the computational work per core remains the same as the number of cores cooperating in a parallel application is increased.

Figure 4.11 depicts the results for MG with strong scaling. MG is memory intensive and uses long and short-distance inter-processor communication. The number of processes grows as power of 2 giving 5 different grid sizes. We observe that NoCMsg is faster than OperaMPI for all the grid sizes. The strong scaling of input size causes the total time to reduce as the

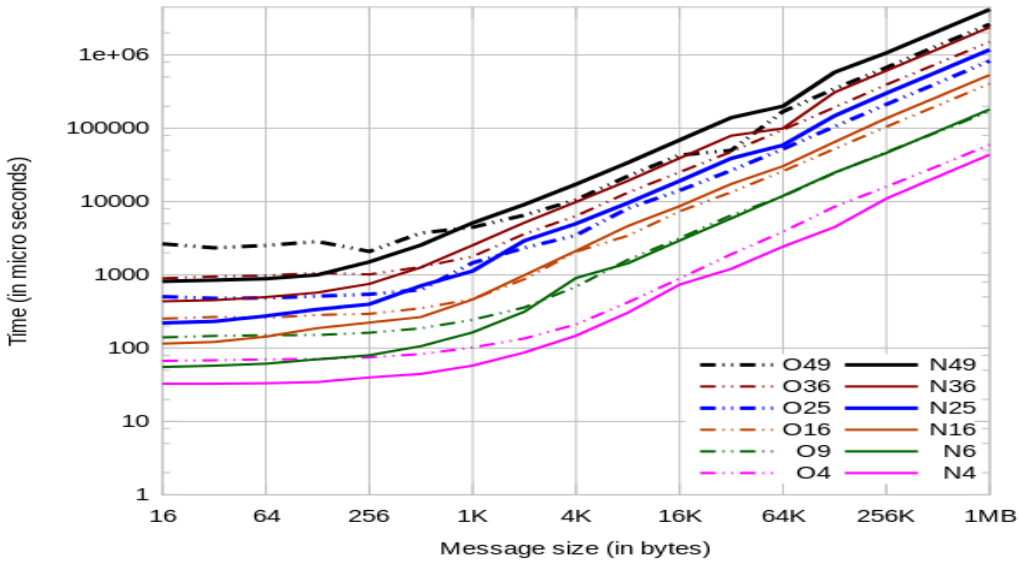

Figure 4.7: Alltoall: Varying Message Sizes

number of tasks increases. For small task sizes NoCMsg is much faster than OperaMPI, but as the number of tasks increases, the difference between the total execution times decreases. This is because MG is memory intensive with limited inter-process communication, for large grid sizes the performance improvement due to efficient communication reduces.

We used weak scaling for other benchmarks, namely IS, FT, CG and LU. FT is a discrete 3D Fast Fourier Transform solver for partial differential equations. CG estimates Eigen values using the conjugate gradient method. FT uses all-to-all communication whereas CG uses irregular memory accesses and communication. IS features high number of collective communication. These benchmarks exhibit less computation and more inter-task communication. The results of IS and CG benchmarks are shown in figures 4.12 and 4.13 respectively. In both cases, the execution time of NoCMsg is lower than the execution time of OperaMPI. The difference in execution time increases with increase in number of tasks showing that the inter-process communication dominates the results for these benchmarks. In case of FT, the difference in execution time is low between NoCMsg and OperaMPI as shown in figure 4.14.

Figure 4.15 shows the results for the LU pseudo application. For small number of tasks when computation dominates total execution time, OperaMPI is faster than NoCMsg. As the number of tasks increases, the inter-task communication starts to dominate the total execution time. The execution time of NoCMsg grows slower than that of OperaMPI, indicating that for larger number of tasks NoCMsg provides better performance.

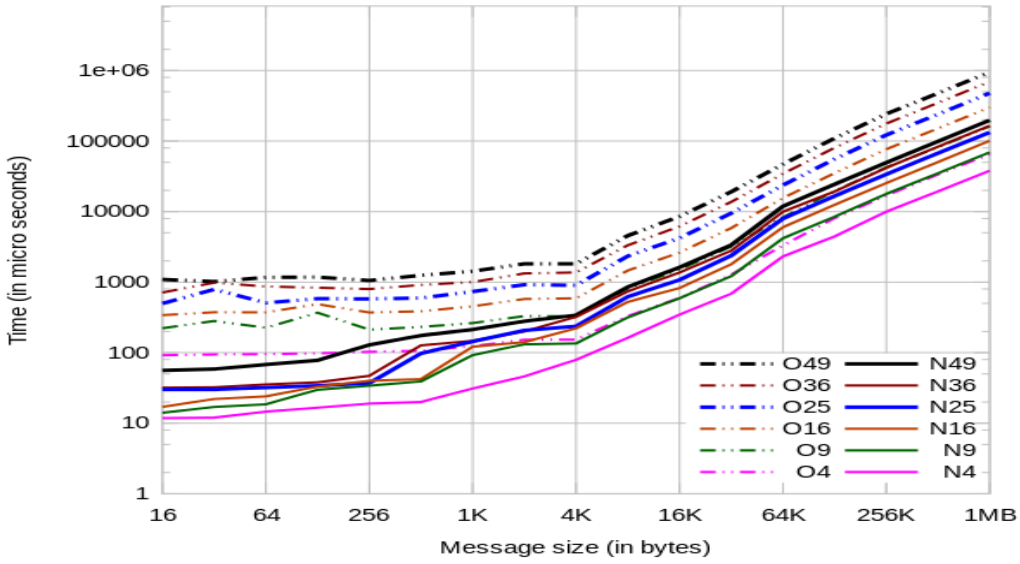

Figure 4.8: Reduce: Varying Message Sizes

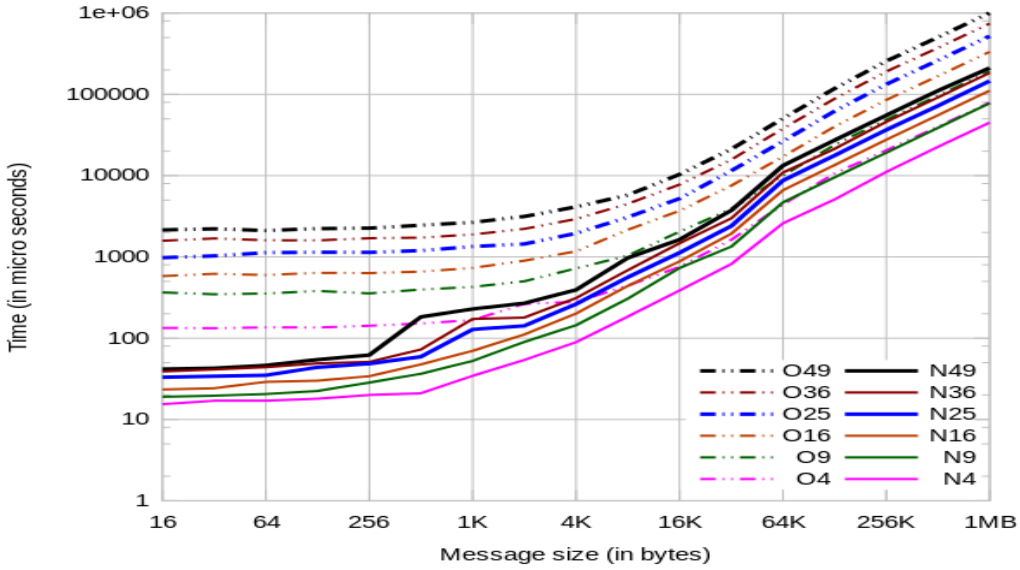

Figure 4.9: AllReduce: Varying Message Sizes

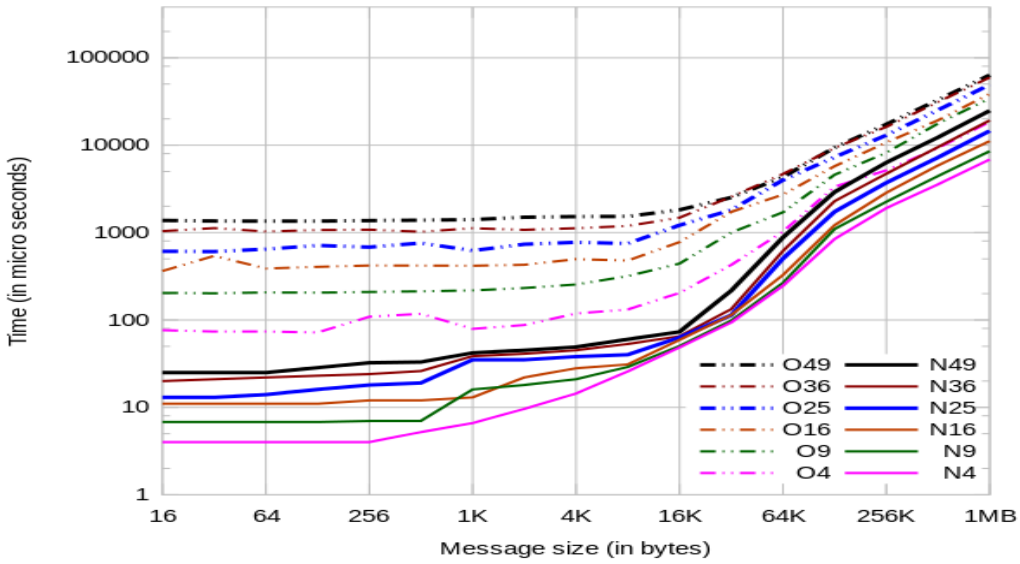

Figure 4.10: Broadcast: Varying Message Sizes

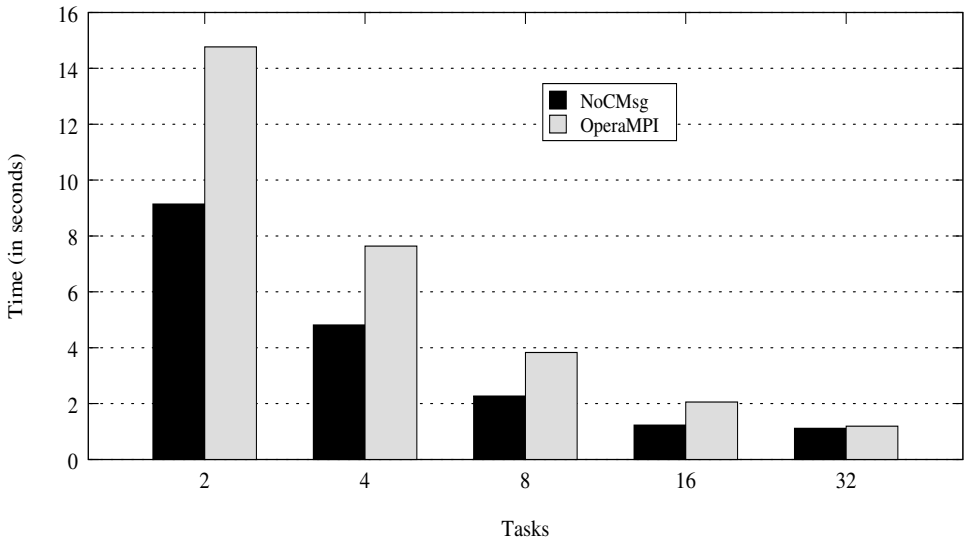

Figure 4.11: NPB MG : Strong Scaling

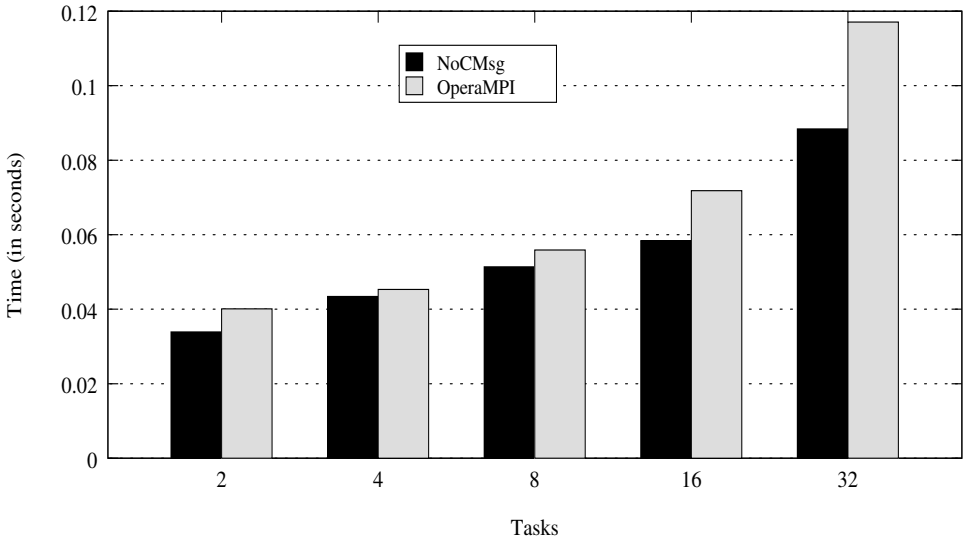

Figure 4.12: NPB IS : Weak Scaling

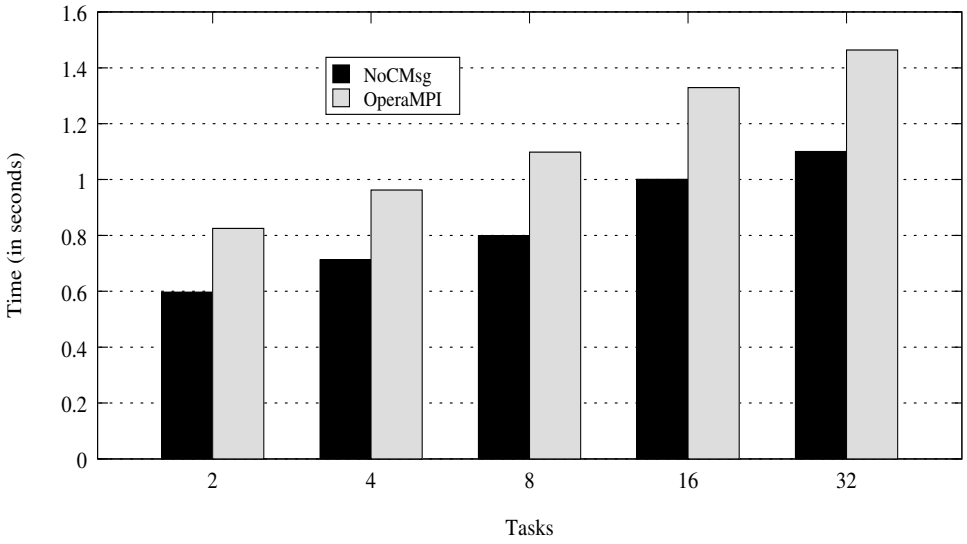

Figure 4.13: NPB CG : Weak Scaling

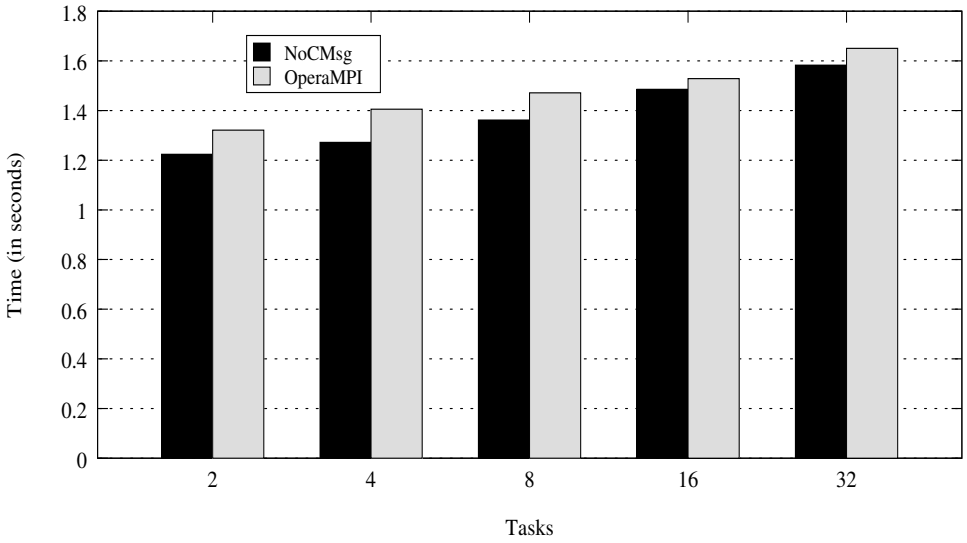

Figure 4.14: NPB FT : Weak Scaling

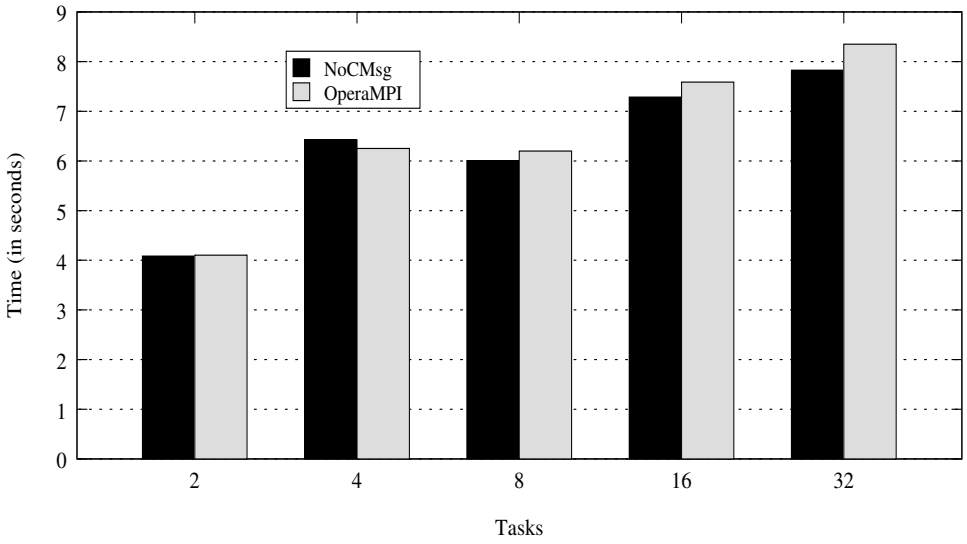

Figure 4.15: NPB LU : Weak Scaling

Chapter 5

Related work

Communication patterns and communication trees as a means to implement collective operations have been well studied [24]. Different approaches have been proposed for different collectives.

One interesting algorithm for implementing broadcast was proposed in [7]. In the algorithm, the source node sends the message halfway across the linear array, partitioning the network into two sub-networks. In subsequent steps, each node holding a copy of the message forwards it to a node in its partition that has not yet received the message. This continues until all the nodes are covered. This approach is loosely based on spanning binomial trees. Another tree based multicast scheme is proposed in [39]. The scheme constructs a quad-branch multicast (QBM) tree for transmitting multicast messages. A QBM tree is a logic tree rooted at the source node of a multicast and has four subtrees. These subtrees are used to distribute the multicast message to a subset of the destinations through a particular virtual network. The destination nodes are partitioned according to their positions relative to the source node in the 2D mesh. Our implementation of Broadcast uses a communication tree rooted at the task performing the broadcast. We also make use of relative position of nodes to the root for building the broadcast tree. Tree branches are mapped onto the NoC in a contention-free manner and used to send the message from the root to all the children. But unlike the QBM implementation, our approach does not require special registers in the routers, support of double-XY routing and changes to message headers.

Several approaches apply graph theory concepts to build efficient trees. One such approach extends the concept of dominating sets from graph theory to build a broadcast tree structure that is composed of multiple levels of extended dominating nodes (EDN) [36]. This approach requires an all-port communication architecture to be efficient, where as, typical NoC based platforms use a single-port communication architecture. Our implementation does not have any such special requirements and can perform well on both single-port and all-port communication

architecture. Our implementation of Reduce and Barrier also use the same approach.

Barrier is the most commonly used collective and needs to be highly efficient. Numerous efforts have been devoted to developing an efficient implementation of barrier synchronization, both in software and hardware. Hardware barriers are typically faster than software barriers [30], but are not scalable. For this reason, a number of methods have been proposed based on the idea of multidestination mechanism, which combines message-passing with hardware support in the routers [23], [29]. A multidestination worm is a message that carries multiple destination addresses so that it can be sent to multiple nodes with a single start up delay. At each intermediate destination, the associated router replicates the message, sends one copy to the local processor, and forwards the other to the next destination. However, these approaches require long headers to carry the information of multiple destination and incur additional processing overhead at each node.

Another way to exploit message-passing is to implement tree-based barrier. One such implementation is the Collective Synchronization (CS) tree scheme proposed by Yang and King [40]. In this scheme every member node builds the CS tree in a distributed fashion by determining its parent-child relationship. The basic idea is to partition the 2D mesh into four overlapping quadrants using the chosen root node as the origin. Each node searches for its parent node among a set of member nodes within the same quadrant that are closer to the root node than itself. Once the CS tree is built special registers in the routers are set up to direct synchronization messages to appropriate output ports. Another similar approach is the Barrier Tree for Meshes (BTM), which is a 4-ary synchronization tree constructed in a recursive manner [26]. The algorithm starts by partitioning the 2D mesh into four disjoint submeshes around the chosen root node. Then, for each quadrant, a local root node is chosen and the quadrant is partitioned again into four submeshes around the local root node. The recursive partitioning continues until there remains only one node in each submesh. These chosen nodes and leaf nodes together form the BTM tree. Our implementation of barrier also uses a tree rooted at a chosen root node. But unlike the other approaches, ours does not require dividing the 2D mesh into submeshes and does not need special registers for building the tree. Our tree-based implementations relies on the relative position of each node from the root and takes advantage of 2D mesh topology to build and map the tree in a contention-free manner.

Our implementation of Alltoall exploits pattern-based communication to concurrently exchange messages between partners. On the surface, this approach shares design strategies with the "direct algorithm" of [35]. The direct algorithm assigns nodes of the mesh the ordinal numbers 0 through N-1 in a row-major fashion. During step k, for $k = 1, 2, ..., N-1$, the node with ordinal number i sends a message to the node whose ordinal number is an exclusive or (XOR) of i and k. This results in a communication pattern similar to ours under dimension order routing. However, unlike our approach, the direct algorithm suffers from link contention.

Other approaches to implement all-to-all require splitting up the tasks into distinct communication groups. Message combining algorithms referred to as binary exchange and quadrant exchange were proposed in [11]. In the binary exchange, the mesh is recursively halved and nodes symmetrically located with respect to each cut exchange block. The quadrant exchange treats the mesh as groups of 2x2 submeshes and exchanges blocks among the nodes in each submesh. Successive groups of 2x2 submeshes are interleaved until all blocks are exchanged. Another algorithm called cyclic exchange proposed for power-of-two 2D meshes [34] makes use of multiple communication phases. In each phase of the cyclic exchange, every node communicates in two steps with two other nodes, one in the same row and one in the same column. In a step of a phase, some pairs of nodes perform the horizontal exchange first, while others perform the vertical exchange first. Subsequent steps reverse the order.

Another message combining algorithm proposed for multidimensional torus and mesh networks splits the mesh into 4x4 block groups [32]. Message exchange is divided into phases. In Phases 1 and 2, nodes in the same group perform all-to-all personalized communication among them. In the next two phases (Phases 3 and 4), message transmissions are performed among nodes in distinct groups and in the same sub mesh. Suh and Yalamanchili [33] introduce bottom-up algorithms for all-to-all communication where communication proceeds from smaller submeshes of the NoC to larger ones. Our implementation also uses a bottom-up approach, but it neither requires division of the grid into smaller submeshes nor does it result in network contention.

More recent approaches focus on building static schedules for all-to-all communication [12]. Some approaches perform path selection, core mapping and time-slot allocation intelligently to resolve conflicts on shared networks [17]. Others exploit Time-Division-Multiplexing based NoC platforms and try to solve the slot and path selection problem to provide contention free communication [31]. Unlike these approaches, our implementation neither requires dynamic route calculations nor offline pre-calculations nor storage of large routing tables. This keeps our implementation simple, generic and scalable with minimum overhead.

Chapter 6

Conclusion

We have designed a set of efficient and predictable group communication primitives using message passing utilizing NoC architectures. The primitives employ highly efficient algorithms to provide contention-free communication and utilize advanced NoC hardware features. These primitives improve performance and reduce imbalance for HPC applications while providing higher timing predictability for high-confidence real-time systems.

Our implementation of the most commonly used collectives reduces the communication time over a reference MPI implementation by up to 95% for single packet messages and up to 98% for larger messages. NoCMsg has superior performance over OperaMPI irrespective of message size for all but one collective: For Alltoall, NoCMsg performs better for message sizes up to 256 Bytes while OperaMPI performs better for larger messages. Evaluation using NPB also shows that NoCMsg outperforms OperaMPI. NoCMsg thus nicely complements prior work that is efficient at larger (yet less common) message sizes for this case. Additionally, the variance of execution times for our implementation is several orders of magnitude lower than that of the reference MPI implementation, making our implementation ideal for balanced HPC as well as hard real-time applications. And instead of assuming ideal NoC symmetry with wrap-around links on the 2D boundaries, our work addresses realistic 2D meshes without wrap-around, such as present in contemporary NoC hardware designs.

Therefore, we can conclude that efficient, predictable and scalable contention-free collective communication can be easily implemented on massive multi-core NoC platforms. Such an implementation can provide balanced parallel execution resulting in improved performance and low timing variability, as postulated in the thesis statement.

REFERENCES

[1] Adapteva processor family. www.adapteva.com/products/silicon-devices/e16g301/.

[2] Nas parallel benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[3] Single-chip cloud computer. blogs.intel.com/research/2009/12/sccloudcomp.php.

[4] Tera-scale research prototype: Connecting 80 simple cores on a single test chip. ftp://download.intel.com/research/platform/terascale/terascaleresearchprototypebackgrounder.pdf.

[5] Tilera processor family. www.tilera.com/products/-processors.php.

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. 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. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[7] Michael Barnett, David G. Payne, and Robert A. van de Geijn. Optimal broadcasting in mesh-connected architectures. Technical report, Austin, TX, USA, 1991.

[8] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598, 2008.

[9] D. Bertozzi and L. Benini. Xpipes: a network-on-chip architecture for gigascale systems-on-chip. *Circuits and Systems Magazine, IEEE*, 4(2):18–31, 2004.

[10] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Comput. Surv.*, 38(1), June 2006.

[11] S.H. Bokhari and H. Berryman. Complete exchange on a circuit switched mesh. In *Scalable High Performance Computing Conference, 1992. SHPCC-92, Proceedings.*, pages 300–306, 1992.

[12] Florian Brandner and Martin Schoeberl. Static routing in symmetric real-time network-on-chips. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS '12, pages 61–70, New York, NY, USA, 2012. ACM.

[13] William J. Dally and Brian Towles. Route packets, not wires: on-chip inteconnection networks. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 684–689, New York, NY, USA, 2001. ACM.

[14] Michael J. Flynn and Patrick Hung. Microprocessor design issues: Thoughts on the road ahead. *IEEE Micro*, 25(3):16–31, May 2005.

[15] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European PVM/MPI Users' Group Meeting*, pages 97–104, September 2004.

[16] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[17] Andreas Hansson, Kees Goossens, and Andrei Rădulescu. A unified approach to constrained mapping and routing on network-on-chip architectures. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '05, pages 75–80, New York, NY, USA, 2005. ACM.

[18] Jörg Henkel, Wayne Wolf, and Srimat Chakradhar. On-chip networks: A scalable, communication-centric embedded system design paradigm. In *Proceedings of the 17th International Conference on VLSI Design*, VLSID '04, pages 845–, Washington, DC, USA, 2004. IEEE Computer Society.

[19] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, July 2005.

[20] M. Kang, E. Park, M. Cho, J. Suh, D.-I. Kang, and S. P. Crago. Mpi performance analysis and optimization on tile64/maestro. In *Workshop on Multi-core Processors for Space — Opportunities and Challenges*, July 2009.

[21] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[22] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[23] X. Lin, P. K. McKinley, and L. M. Ni. Deadlock-free multicast wormhole routing in 2-d mesh multicomputers. *IEEE Trans. Parallel Distrib. Syst.*, 5(8):793–804, August 1994.

[24] Philip K. McKinley, Yih jia Tsai, and David F. Robinson. A survey of collective communication in wormhole-routed massively parallel computers. *IEEE COMPUTER*, 28:39–50, 1994.

[25] Giovanni De Micheli and Luca Benini. *On-Chip Communication Architectures: System on Chip Interconnect*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[26] Sangman Moh, Chansu Yu, Ben Lee, Hee Young Youn, Dongsoo Han, and Dongman Lee. Four-ary tree-based barrier synchronization for 2d meshes without nonmember involvement. *IEEE Trans. Comput.*, 50(8):811–823, August 2001.

[27] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, February 1993.

[28] John D. Owens, William J. Dally, Ron Ho, D.N. (Jay) Jayasimha, Stephen W. Keckler, and Li-Shiuan Peh. Research challenges for on-chip interconnection networks. *IEEE Micro*, 27(5):96–108, 2007.

[29] D. K. Panda. Fast barrier synchronization in wormhole k-ary n-cube networks with multidestination worms. In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, HPCA '95, pages 200–, Washington, DC, USA, 1995. IEEE Computer Society.

[30] Vara Ramakrishnan and Isaac D. Scherson. Efficient techniques for nested and disjoint barrier synchronization. *J. Parallel Distrib. Comput.*, 58(2):333–356, August 1999.

[31] Radu Stefan and Kees Goossens. An improved algorithm for slot selection in the thereal network-on-chip. In *Proceedings of the Fifth International Workshop on Interconnection Network Architecture: On-Chip, Multi-Chip*, INA-OCMC '11, pages 7–10, New York, NY, USA, 2011. ACM.

[32] Young-Joo Suh and Kang G. Shin. All-to-all personalized communication in multidimensional torus and mesh networks. *IEEE Trans. Parallel Distrib. Syst.*, 12(1):38–59, January 2001.

[33] Young-Joo Suh and Sudhakar Yalamanchili. All-to-all communication with minimum start-up costs in 2d/3d tori and meshes. *IEEE Trans. Parallel Distrib. Syst.*, 9(5):442–458, May 1998.

[34] N. S. Sundar, D. N. Jayasimha, D.K. Panda, and P. Sadayappan. Complete exchange in 2d meshes. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 406–413, 1994.

[35] Rajeev Thakur and Alok Choudhary. All-to-all communication on meshes with wormhole routing. In *In Proceedings of the 8 th International Parallel Processing Symposium*, pages 561–565, 1994.

[36] Yih-jia Tsai and Philip K. McKinley. Broadcast in all-port wormhole-routed 3d mesh networks using extended dominating sets. In *Proceedings of the 1994 International Conference on Parallel and Distributed Systems*, pages 120–127, Washington, DC, USA, 1994. IEEE Computer Society.

[37] J. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In *International Parallel and Distributed Processing Symposium*, April 2002.

[38] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27:15–31, 2007.

[39] Jenq-Shyan Yang and Chung-Ta King. Efficient tree-based multicast in wormhole-routed 2d meshes. In *Proceedings of the 1997 International Symposium on Parallel Architectures,*

Algorithms and Networks, ISPAN '97, pages 494–, Washington, DC, USA, 1997. IEEE Computer Society.

[40] Jenq-Shyan Yang and Chung-Ta King. Designing tree-based barrier synchronization on 2d mesh networks. *IEEE Trans. Parallel Distrib. Syst.*, 9(6):526–534, June 1998.