Abstract

PATIL, ONKAR. Efficient and Lightweigth Inter-process Collective Operations for Massive Multi-core Architectures. (Under the direction of Dr. Frank Mueller.)

Multi-core processors are evolving fast and have taken the form of massive multi-core System on Chips (SoC). SoCs are quickly approaching the range of hundreds of independent processing cores and may soon reach to the range of thousands of processing cores. In order to harness such considerable computation power, future operating systems should be designed with the ability of efficiently handling huge number of processes running independently or dependent on each other. Multiple processes running simultaneously will need to communicate at the pico-kernel level with each other. Such communication may become a bottleneck for the performance of massive multi-core systems. Communication utilizes the Network on Chip (NoC), a very high bandwidth mesh of buses and routers for point-to-point or collective operations. Currently available libraries utilize this NoC but do not extract maximum performance due to contention. The motivation of this work is this link and core contention that exists in most communication patterns. We contribute novel designs for collective operations to avoid both link and core contention in support of efficient communication between cores on the Intel Single Chip Cloud (SCC) with reduced latency and high throughput. The design scales well in terms of both core count and message sizes and gives 2x to 9x increases in improved throughput than prior designs. It outperforms the prior work by an order of magnitude in latency using basic message passing benchmarks for all collectives.

Efficient and Lightweigth Inter-process Collective Operations for Massive Multi-core Architectures

by
Onkar Patil

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 Science

Raleigh, North Carolina

2014

APPROVED BY:

Dr. William Enck	Dr. Blair Sullivan

Dr. Frank Mueller
Chair of Advisory Committee

Dedication

To my Parents and my elder brother.

Biography

The author hails from Mumbai, Maharashtra in India where he was born and brought up by his family. He completed his schooling from St. Anne's High School, Malad and Sathaye Junior College in Mumbai. In 2012, he completed his undergraduate degree from Fr. Conceicao Rodrigues College of Engineering, Mumbai in the field of Information Technology. He was admitted to North Carolina State University, Raleigh in USA in Fall 2012 to pursue his graduate studies in the field of Computer Science. He began working as a Graduate Research Assistant under the guidance of Dr. Frank Mueller in May 2013. He takes keen intrest in pursuing his hobbies in music, dance, sports and reading. After graduating, he will be joining NetApp Inc., RTP, North Carolina to work with the EPS team.

Acknowledgements

I would like to take this opportunity and thank everyone who has played a part in bringing success to me at this point in time.

I would thank my advisor Dr. Frank Mueller to have trust and faith in my capabilities and give me this golden chance to work on this research. His guidance and immense support has been the biggest factor in completion of this thesis. His encouragement and motivation to acheive the desired results has been inspiring. I appreciate his constant availability and help for his students.

I would like to thank Dr. William Enck and Dr. Blair Sullivan to take interest in my research and who agreed to be members of my advisory committee. I would also like to thank my lab mates and people of my research group for their help, support and encouragement whenever I needed it. It was a pleasure working with you all.

I want to thank my room mates Rohit and Srihari for helping me throughout the last two years and making my stay enjoyable. I would also like to thank my friends Shruti, Sanjana, Apurva, Namrata, Aditi, Georgy, Aparna and Smriti who were here in Raleigh and looked after me for anything I ever needed. They supported me through some of the toughest times I have faced. I thank all my friends back in India and the ones spread all around the world for their constant support and well wishes.

I want to thank my elder brother, Nishad, for all his love, care and motivation. His words of advice and guidance have been invaluable. Last but not the least, I want to thank my parents who trusted me the most, that I was capable of pursuing my dreams and ambitions and supported me with all that they have got. Their love and kindness have made difficult times seem very easy.

Table of Contents

LIST OF FIGURES

Chapter 1

Introduction

1.1 Introduction

Earliest computer designs consisted of single-core processors which would perform all the arithmatic and logic operations with the help of a off-chip DRAM memory. To enable fast processing and pipelining of data to the cores, fast SRAM caches were introduced. However, the maximum computational power came from the processing cores which improved in terms of their frequency so that they could do more computations every second. The field of computer architecture design has moved from single-core processors to multi-core processors and is swiftly moving toward massive multi-core processors [30]. The onus of this shift is to achieve higher performance in solving complex computational problems and to provide an efficient and resilient platform for future applications. The advanced technology that is used in building these architectures has ensured that the potential performance of the massive multi-core architectures will create a challenge to build highly complex and demanding but useful applications.
To achieve faster computations for complex problems, the concept of parallelism was introduced in computers. Parallelism means to simaltaneously perform multiple operations to acheive more computations in less time. Earlier, parallelism of tasks was seen as a feature of the applications. The operating systems of the past were not designed to support physical parallelism due to majority of single-core processors [22]. However, the concept of multi-processing and multi-threading were accommodated in the design, as the applications demanded to launch more and more tasks simultaneously. Multi-processing divides the data and instructions of a particular application into different processes. Multi-threading divides the data and instructions of a process into differnt threads. The processor with the help of a scheduling policy works on all the threads and processes on a time sharing basis. The concepts of single instruction multiple data (SIMD) style applications and symmetric multi-processing (SMP) style emerged from this modification to the traditional operating system design [21]. In SIMD and SMP, same instruction

set is provided to every core, but different data is passed to every core from a large dataset. Some the data from the dataset is shared between processors, e.g., boundary values of matrix for doing matrix multiplication. These values need to communicated between the cores to obtain the correct computation. One of the main challenges of parallelism is to support efficient communication between processes or threads as they share data. The processes and threads wait on this shared data to obtain exclusive or shared access, which may waste computational resources. This inter-task dependency has the potential to perform and to fall short of peak parallel performance.

Over time, as 2, 4, 8, 16-core processors were introduced [28] [29], the potential for parallelism increased but the challenge of inter-task communication was not handled by traditional operating system design. Parallelism was restricted to independent processes executing on separate cores while dependent tasks were continued to be executed on a single core due to lack of inter-task communication. This modification posed new challenges as it was difficult to keep all the cores busy, which lead to load imbalance and wastage of computational power. Cores could perform any computation even if the processes and threads on other cores were starving. Performance of traditional operating systems could not scale up to the increasing number of cores in multi-core architectures. The number of cores in a processor has now reached the massive multi-core realm, and the inability of operating systems to extract close to peak performance needs to be addressed.

Currently, there is a need for a mechanism in the modern operating systems to support efficient and scalable inter-process and inter-task communication that aid in dynamic task scheduling and load balancing for multi-core architectures. These mechanisms should not alter traditional operating system designs and rather adapt to the new features introduced in multi-core architectures to improve communication between the cores. We propose to support task communication at the pico-kernel level to be utilized by the task scheduling and load balancing activities at the micro-kernel level. The pico and micro-kernel will be transparent to the Operating System kernel which will execute on top of them. However, the communication inter-connect on the processor chips suffer from performance bottlenecks in the form of link and core contention [7] [16]. Contention happens when multiple entities contest for access of the same resource. It causes the computational time of a process to increase. Multi-core systems are used in real-time applications where quick response time is expected and strict deadlines are followed. A delay in execution due to inefficient inter-process communication can be critical to any application. Hence, it becomes imperative to address this issue.

The design of an inter-process communication method without performance bottlenecks is the central idea of this research. We will demonstrate the challenges mentioned above on the 48-core Intel SCC platform [18] and then propose our solution to completely avoid bottlenecks due to contention on the interconnect. The Intel SCC introduced the message passing buffer (MPB)

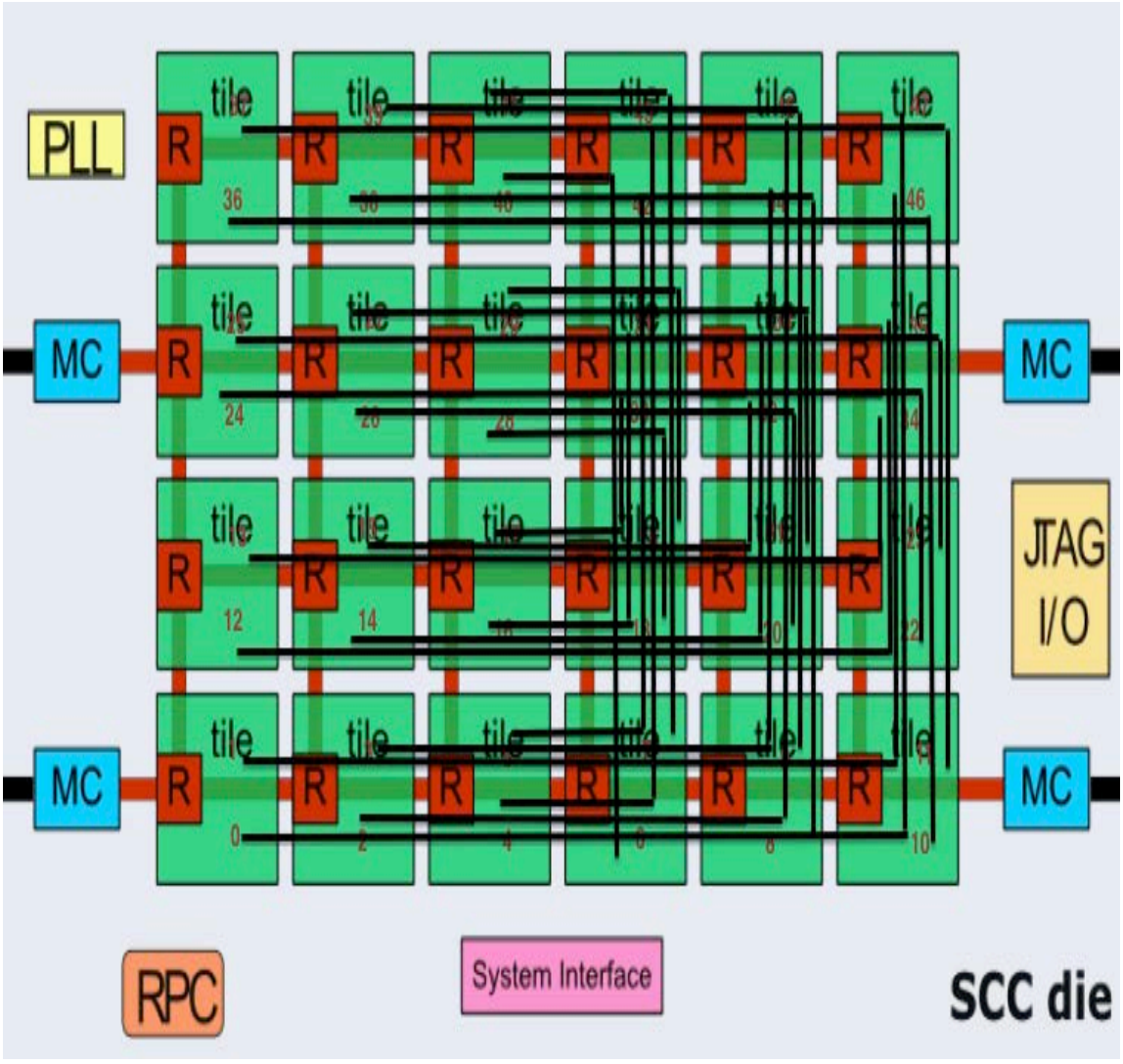
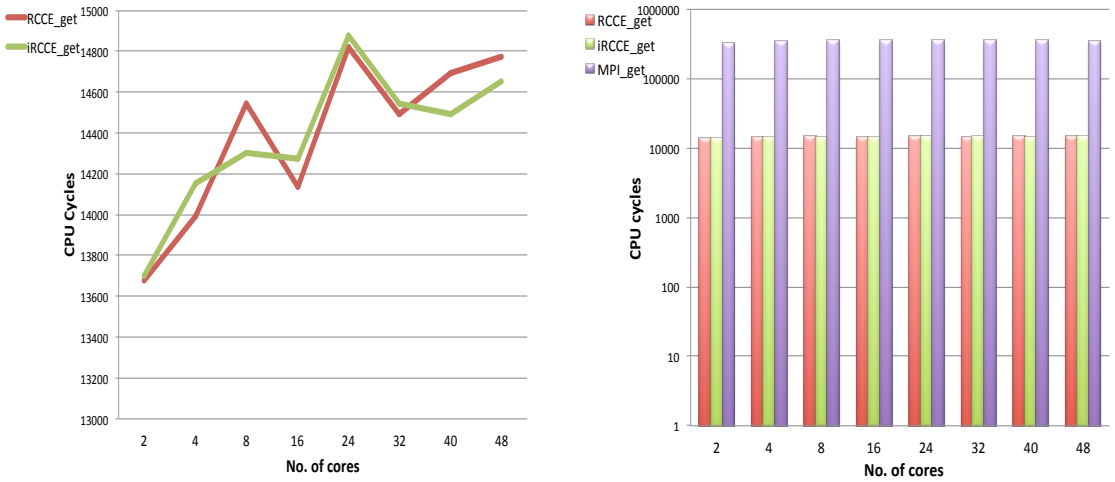


Figure 1.1: A extreme contention scenario on the SCC

to provide ultra-fast inter-process communication between the cores [4]. This is facilitated by a high-bandwidth on-chip mesh and fast SRAM technology. Communication using the MPB is 3 orders of magnitude faster than communication using off-chip shared memory. Using message passing over the MPB also removes the burden of synchronization of shared memory from the cores. But the MPB architecture presents a potential performance bottleneck.

The Intel SCC has a mesh network to enable inter-processor communication. The physical communication design of the 2D mesh using the MPB, on the SCC, is bound to have link and core contention. The cores are arranged in 24 tiles where every tile has 2 cores. The tiles are arranged in 6x4 matrix form. Every core communicates via a crossbar router that is present in

the tile they belong to. This router is connected to 2 or 4 links, depending on its position on the mesh and is also connected to the MPB on its tile. It can gain access to only one link at a time to maintain the atomicity of the communication operations. The messages are routed using the Manhattan routing where the messages can travel only along the X or Y axis. Due to pre-computed X-Y dimension order routing, multiple paths cross over a single link. Figure 1.1, depicts a scenario where every core communicates with every other core. The communication is one-sided using *put()* or *get()* primitives and all paths are inter-tile. Maximum contention is observed in the horizontal links that connect tiles in middle of the rows and right-most columns in the vertical links.



(a) Latency for RCCE, iRCCE get()

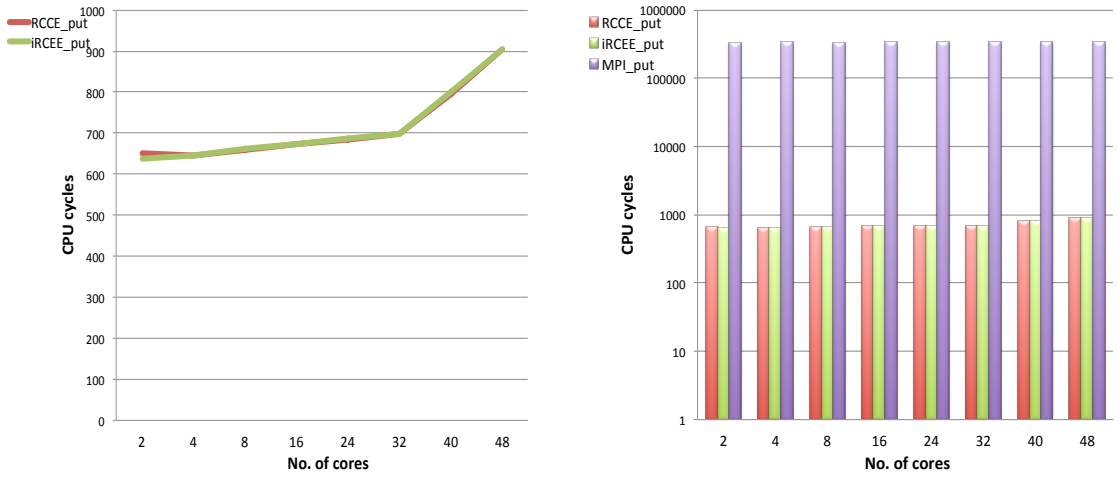
(b) Latency for RCCE, iRCCE, RCKMPI get()

Figure 1.2: Contention results for *get()* primitive of RCCE, iRCCE and RCKMPI

1.1.1 Motivation

At any given time, if multiple routers contest for the access to the link, access is given to a single router by arbitration, which is similar to the CSMA-Aloha protocol [5]. This allows only a single core to gain access to a link at any given time. Such exclusive router and link access is the root cause of contention and reduces the actual throughput of the mesh considerably, which in turn affects the performance of any application. To demonstrate this, we conduct two experiments on the SCC using the one-sided communication primitives of RCCE, iRCCE and RCKMPI libraries. In the first experiment, we make every core get a message from another remote cores MPB. This creates the exact link contention scenario as shown in figure 1.1. In the second experiment, all the cores put a message in the MPB of core 0. This causes core contention at core 0.

Figures 1.2(b) and 1.3(b) show the results for experiments comparing the performance of



(a) Latency for RCCE, iRCCE put()

(b) Latency for RCCE, iRCCE, RCKMPI put()

Figure 1.3: Contention results for *put()* primitive of RCCE, iRCCE and RCKMPI

RCCE, iRCCE and RCKMPI. The Y-axis shows the time for a process to complete in cycles and the X-axis shows the number of cores communicating. In Figures 1.2(a) and 1.2(b), we observe that the number of cycles consumed by *get()*, both blocking and non-blocking primitives, increases the number of cores. There is an increase of 1200 cycles per core when number of cores communicating increase from 2 to 48. This is caused by link contention. A similar behavior is observed for *put()* as seen in Figures 1.3(a) and 1.3(b) due to core contention. The number of cycles consumed increases from 650 to 900 as number of cores increases from 2 to 48.

Core contention is less significant than link contention as observed in Figures 1.2(a) and 1.3(a). Hence, intra-tile communications are not affected much. But link contention is more significant since many cores are competing for the access to the link. This drop in performance is amplified for collective operations, where minimum latency is a critical factor for achieving high performance for complex parallel problems. When scaled to larger number of cores, contention will adversely affect performance.

1.2 Hypothesis

To find a solution to the contention problem, we put forth a hypothesis which proposes to increase performance of modern day multi-core architectures.
If contention on mesh-based network-on-chip interconnects of multi-core architectures is avoided, performance of collective communication with their underlying point-to-point messages can be significantly improved.
Existing communication libraries do not take such contention into account for their design due to which their performance falls short of the potential of the SCC. Their algorithms suffer from contention resulting in reduced throughput. To avoid and remove the effects of contention, we have designed our algorithms in such a way that no two cores communicate over the same link at the same time. Accesses under contention are restricted to intra-tile, i.e., involving at most two cores at any given time. This improves the performance of the collective operations and contributes better performance for complex problems on the SCC. These algorithms can be scaled to any number of cores on a mesh architecture.
In the following, we provide background on multi-core architectures, the Intel SCC platform and communication libraries for the SCC in Chapter 2. In Chapter 3, we summarize past and current research related to this paper. Chapters 4 and 5 we specifically talk about the design of our proposed solution and its implementation respectively. Chapter 6 will assess the performance of our solution for the SCC architecture and Chapter 7 describes open problems followed by the conclusion in Chapter 8.

Chapter 2

Background

2.1 Architecture

The evolution of computers based on the von-Neuman architecture has constantly strived for higher performance by increasing floating point operations per second [23]. The evolution progressed in two directions. One direction was by increasing the clock ticks per second where with higher clock speeds, computers were able to execute large number of instructions per second. The other direction was due to gate width minimization and threshold voltages reduction, which in turn, reduced static delays and improved the performance of the processors. Both progressions, simultaneously, transformed processors into machines with higher performance, with every small progression following Moore's Law [33] in doubling transistor counts and integer floating point operations per second every two years.

However, both progressions were stalled due to excessive leakage power. Processors coped with slower clock speeds to avoid excessive power consumption. But the demand for higher performance was always on the rise with ever larger applications and data sets. The cost of the transistors, however, kept reducing due to advances in the CMOS production technology and the gate width kept decreasing. Hence, the evolution progressed into a new direction of multi-core processors to increase the performance of computers and to keep up with high performance demands.

Today, multi-core architectures have core counts up to 64 cores (Tilera TilePro64, Parallela Epiphany-IV) [24], [27], where the cores can be independent with their own private caches or orchestrate computation in conjunction with other processing cores. The core count for co-processor multi-core architectures reached 192 cores (Tegra K1) [25] for unified memory systems and 5760 processing cores (Nvidia Titan Z) [26] for discrete graphics processing units (GPUs). Co-processor architectures cater largely to multi-threaded applications using SMP-style memories whereas independent processor multi-core architectures support applications with indepen-

dent processes. Such multi cores are interconnected by a 2D mesh of high bandwidth links and crossbar routers. This mesh network is known as the NoC as it resides inside the multi-core chip. These chips have very low power consumption. Some of the latest chips consume less than 5 Watts (Parallela Epiphany-IV) [27]. Such massive multi-core processors are now able to achieve performance up to 384 Gflops (Tegra K1) [25] on the LINPACK benchmarks.

These new multi-core architectures are called SoCs because every core with its own private memory and the mesh network makes up a complete system on the chip. Multi-core architectures leverage the shared memory available on the chip to achieve maximum performance by use of very fast SRAM memory for synchronization and message passing as well.

2.2 Platform

The Intel SCC is a 48 core research chip developed by the Intel Labs [18]. As we see in Figure 2.1 it has 24 tiles with 2 cores per tile. The tiles are arranged in 6x4 grid die. Every tile is identified by *(x,y)* coordinates starting from (0,0) for the leftmost bottom tile and (5,3) for the rightmost upper tile. The cores inside each tile are Intel P54C processors that run at 800 MHz. Every core has its own private L1 cache. There is a separate 16KB L1 instruction cache and a 16KB L1 data cache. Every core has its own 256KB L2 cache as well. The L2 cache resides on the tile itself. Each core supports 32-bit addressing.

Every tile has a fast SRAM memory known as the Message Passing Buffer or MPB. The MPB is a 384KB SRAM memory shared equally between every core by default. The default 8KB per core is configurable. All cores can reference any address in the space of all other cores on the MPB. This MPB can be used for the inter-process communication on the SCC. It is the main feature of the architecture that, when leveraged effectively, can boost performance significantly. Any communication on the MPB is actually moving data from the L1 cache of the sending core to the MPB and then to the L1 cache of the receiving core. Intel has introduced a new datatype, message passing buffer type (MPBT) [4] for supplying message passing data.

MPBT typed data bypasses the L2 cache. SCC does not support cache coherence between cores. So cores might read stale MPBT data from the L1 cache. To avoid this, a new instruction CL1INVMB, invalidates the data typed as MPBT in the L1 cache. This is done every time a core uses the MPB for communication. Any subsequent access to the data becomes an L1 cache miss. Every core has its lookup table that translates a 32-bit core address to a 46-bit system address. The 46^{th} bit in the system address is the bypass bit. The next 8 most significant bits determine the tile and the core to be requested for access. The next 3 bits determine if it is a non-local MPB access, configuration register access, a memory controller interface or the system interface. The lower 34 bits are used by the memory controllers specified by the cores to access off-chip DRAM memory.

The SCC has 4 on-die memory controllers, which can be accessed by any core. Each controller can address up to 16 GB using the lower 34 bits of the system address and up to 64 GB of total DRAM memory off-chip. The SCC also has an on-die mesh network, which is the backbone of communication [5]. It carries all the traffic on the chip (I/O, memory, message passing). It has a bisection bandwidth of 2TB/s at 2 GHz and 1.1V. The latency is 4 cycles, which includes 1 cycle each for input arbitration, switch arbitration, virtual channel allocation and data transmission. The link width is 16 bytes. Each link has a bandwidth of 64 GB/s and is divided equally between 8 virtual channels over 2 message classes. It is a very low power network with a consumption of just 500mW at 50 °C.

Every tile has a low power 5 port crossbar router shared by both cores on a tile. The mesh

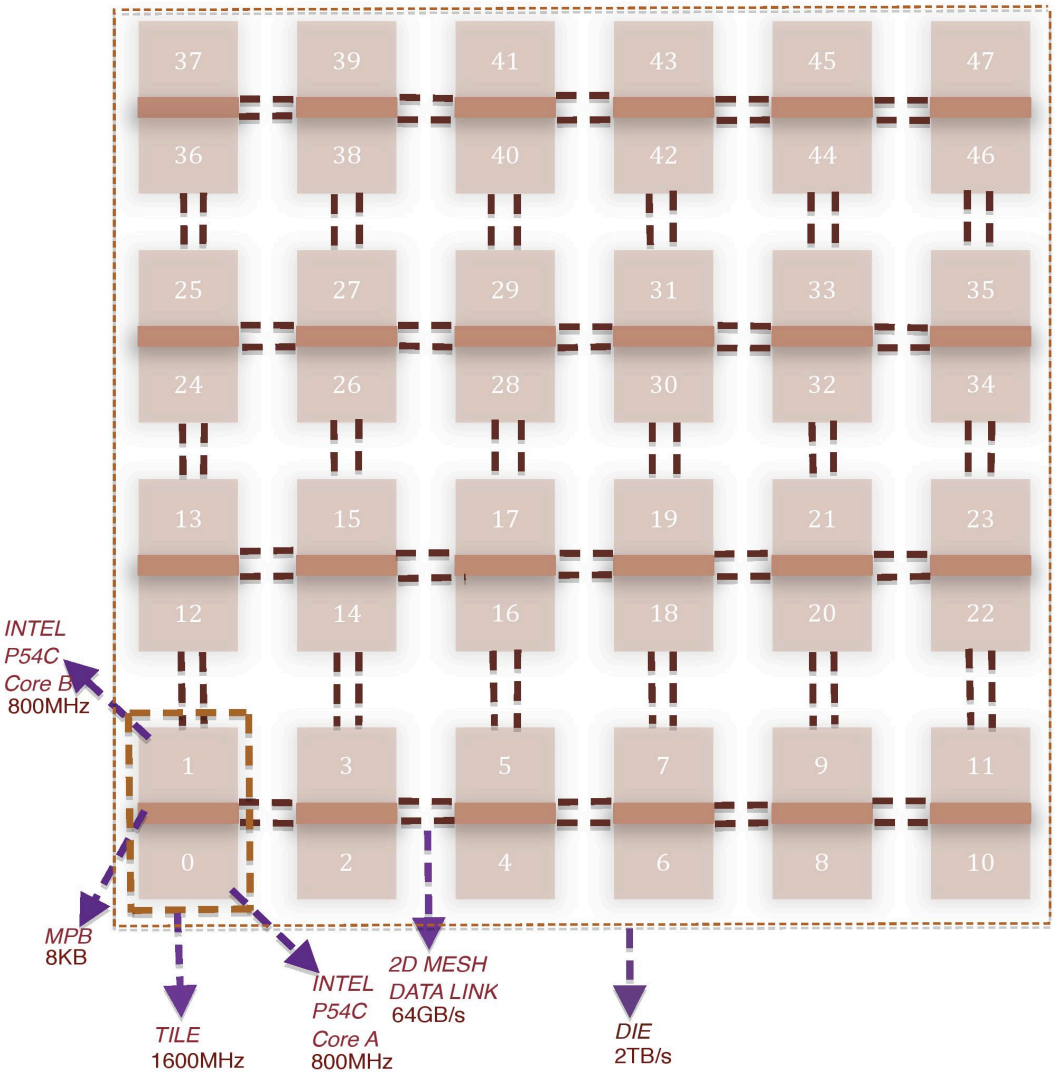


Figure 2.1: Intel SCC architecture schematic

network uses a pre-computed X-Y dimension-ordered routing. This means that a path from the sending core to the receiving core is not the same when a message is returned from the receiving core to the sending core. Each core executes a separate image of Linux and is independent of other cores. Each have their own IP addresses and can also communicate with each other via TCP/IP.

2.3 Communication Libraries

To take complete advantage of the SCC architecture and enable inter-process communication over the on-chip 2D mesh network, many libraries have been developed. Programmers can utilize them and build their own applications on top of them. These libraries hide the *bare metal* details of the SCC and provide clean and easy-to-use interfaces for developers. These libraries mostly follow the MPI-2 communication paradigm and provide many of the functionalities defined in the standard. The libraries choose their own protocols and networks in order to enable communication between the cores. They provide point-to-point interfaces as well as collective interfaces.

For the SCC, RCCE is a simplified collective communication library developed by Intel [9]. It was built with the aim of achieving high performance for parallel dense matrix computations on the SCC architecture by providing collective communication. It communicates using the MPB only. It can run on a *bare metal* machine, a function emulator on top of OpenMP or on a Linux image on SCC. RCCE provides only blocking point-to-point interfaces to send and receive, point to point messages, put and get for one-sided operations and collective interfaces for broadcasting, reductions and barriers.

The iRCCE library was later introduced to enable non-blocking point-to-point communication on the SCC on top of RCCE [8]. It introduced non-blocking point-to-point interfaces for two-sided (isend, irecv) and one-sided communication (iput, iget).

RCKMPI was designed to support complete MPI-2 standard on the Intel SCC [10] [11]. It is a modified version of MPICH-2 [12]. It takes into account the memory architecture of the SCC and the presence of the MPB. It provides all point-to-point interfaces and collective operations in the MPI-2 standard and, hence, facilitates parts of MPI applications to the SCC. It can be configured to communicate over the TCP/IP stack, MPB, shared memory to use MPB and shared memory together.

Chapter 3

Related Work

This work contributes to the PICASO project [2] [3] which proposed a solution to the problem of link contention for the Tilera processor platform. Collectives were designed to avoid link contention on the user dynamic network interconnects on the Tilera chip and to communicate in a row/column approach over the 2D mesh of the NoC [1]. This paper demonstrates platform independence of the solution but also caters to specific features of the SCC, to leverage the hardware underneath. van der Wijngaart et al. have provided blocking communication for SCC via RCCE [9] [14]. Clauss et al. developed the non-blocking version for point-to-point interfaces via iRCCE [8]. These libraries provide the primitives to interface between cores by exploiting the underlying hardware features. They communicate using the MPB and build their collectives on top of two-way communication interfaces. They have collectives for *bcast(), reduce()* and *allreduce()*.

Urena et al. developed RCKMPI library to enable execution of MPI applications on the SCC [10]. RCKMPI can be configured to run with the MPB or shared memory and provides collectives following the MPI-2 standard. They have all the collectives defined in the MPI-2 standard. However, none of the implementations address the performance bottlenecks caused by contention. Furst et al. analyzed the performance and power consumption of RCCE and showed that contention existed in the design of the collectives of RCCE [6]. They also analyzed its impact on performance of applications.

Petrovic et al. also analyzed the performance of the RCCE, iRCCE and RCKMPI and addressed the issue of core contention in collective operations [7]. They proposed a pipelined k-ary tree-based solution for *bcast()* that avoids core contention as opposed to the scatter-allgather approach. They have designed their algorithm for *bcast()* based on the LogP model.

Chandramowlisharan et al. also analyzed the contention on the SCC for one-sided communication interfaces and proposed tree-based communication patterns for collectives like *bcast()* and *reduce()* [15]. Their argument was that network contention on the SCC was negligible and does

not have a significant impact on performance of collective operations.
Rotta and Nolte figured out the best approach for collective operations out of shared memory and message passing [34]. They demonstrated the advantages of using message passing over shared memory for inter-process communication. Kohler et al. optimized the MPI collectives for SCC by reducing the overheads incurred in message passing between cores [16] [17]. Their assumption was that the overheads when using MPI collectives on external networks are known and can be optimized when MPI is used on NoCs with static routing. These are few of the many works that have been done related to SCC and massive multi-core architecture. The ongoing and future work will be discussed in Chapter 7 which talks about the future work.

Chapter 4

Design

The Intel SCC features a *X-Y* dimension-ordered routing to communicate over the NoC. This introduces the drawbacks of link and core contention on the NoC when multiple pairs of cores communicate over same link. The main idea behind designing the following algorithms is to avoid core contention and link contention completely. This implies two key requirements for every collective operation. 1) Only one core should be communicating on a tile at any point of time to avoid core contention. 2) Only one core should communicate over any given link in the NoC at any point of time to avoid link contention.

We can regard the NoC as a NxM matrix graph of links and routers (see Figure 2.1). The routers are nodes and the links are edges. For the SCC, we have a 6x4 matrix, where a router along with 2 cores make up a node and the links are the edges. The design for collectives can be modeled as the problem of traversing a graph (mesh) in a specific way as required by a collective. To satisfy both the requirements, we consider two parts of each algorithm. In the first part, only one core communicates with another core to remove core contention. In the second part, every core communicates over only one link to remove link contention. This is the design basis for every collective. The traversal can have a single start point and multiple end points or multiple start points and single end point. We have designed six collective operations namely *bcast()*, *reduce()*, *barrier()*, *allreduce()*, *alltoall()* and *alltoallv()*. We selected these collectives from the vast array of collectives of MPI because all others are variations or combinations of one of the collectives above.

To meet the stated requirements of MPI, we used a combination of source routing and directional flooding algorithms. Every core will make its own decisions for routing a message to the destination. The message propagates through the mesh in a flooding manner but is restricted to propagate only in certain directions based on the position of the source/destination cores in the mesh. Messages propagate in row and column-wise manner. Every collective is divided in two stages: (1) intra-tile communication and (2) inter-tile communication. This two-staged

communication ensures that no contention exists between any cores participating in the collective operation. All collectives are derived from the pattern for broadcast or reduction.

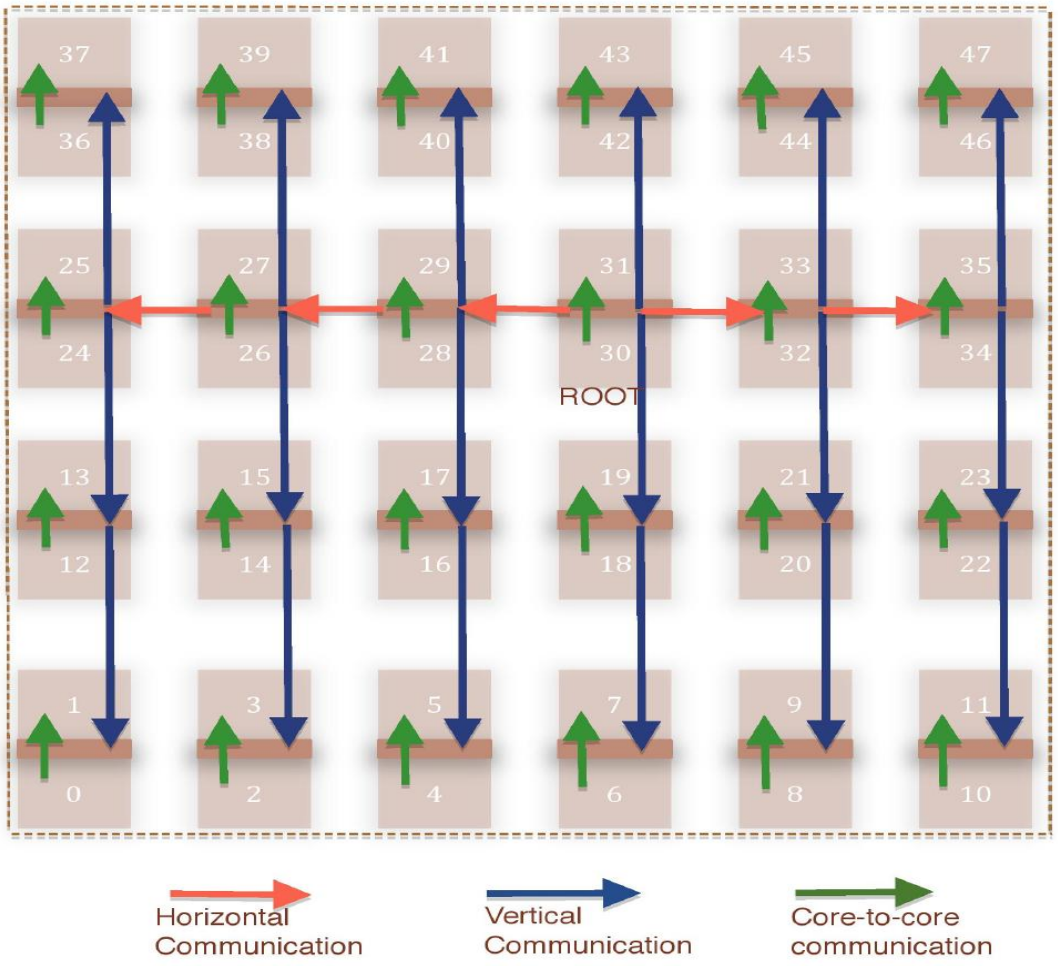


Figure 4.1: Communication pattern of Broadcast on Intel SCC

4.1 Broadcast

Let us consider the broadcast pattern. Broadcast is used to communicate a message to all the cores on the NoC. The broadcast pattern has a single start point and multiple end points(see Figure 4.1). It can be viewed as a graph traversal where, given a start point, we traverse each core once. Starting from the root core, the first stage of the broadcast involves inter-tile communication. The root core propagates the message within its row, i.e., to the cores of the neighboring tiles on the left and right. If the root is adjacent to a boundary, it will propagate the message to only the left or right neighbor. The neighbors propagate the message away from

the root, i.e., in opposite direction from which they received the message. Once every tile in the row has received the message, all of them propagate it to the neighboring tiles in their column up and down or, at the boundary, just up or down. The tiles in the neighboring rows of the root propagate the message in the opposite direction from where it was received. This continues until all tiles have received the message. Then, in intra-tile communication, every core that received the message sends the message to the other core in its respective tile.

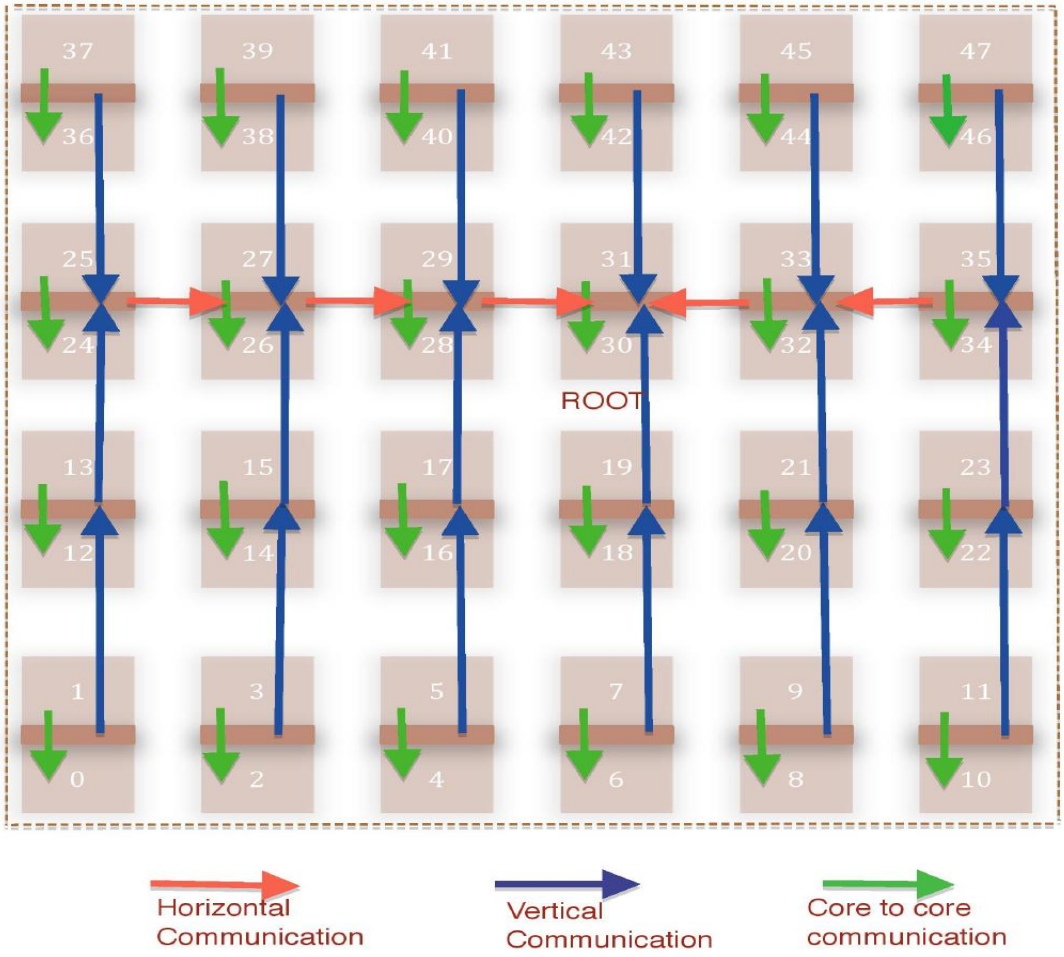


Figure 4.2: Communication pattern of Reduce on Intel SCC

4.2 Reduce, Allreduce and Barrier

Reduction is a common operation for composing values via basic arithmetic operations across cores on the NoC. Reductions have multiple start points and a single end point(see Figure 4.2). It is equivalent to shortest path traversal to reach the root. This pattern operates in reverse order of the broadcast algorithm. It starts with intra-tile communication, where one core on every tile sends a message to the other core on the tile. In the inter-tile communication phase, the receiving cores then communicate with the neighboring cores in their rows either above or below depending on their position respective to the root core. The cores receive in one direc-

tion and send in the opposite direction. The cores in the row of the root receive from both the neighbors that are above and below them. Then, they propagate the message toward the root. Cores from the edge propagate to their neighbors in the direction of the root. Finally, the root receives messages from both the neighbors if in the middle of the row, and only from one neighbor at the boundary. In this way, all cores propagate the message to the root core without any link contention.

All other collectives are a combination or a variation of the above two patterns. The barrier is an empty message reduction. It is used to ensure synchronization of all cores with respect to a certain execution point. The barrier communicates in the same way as a reduction but its message payload is empty. The allreduction collective communicates the reduced message over all cores back to every core. It works in 2 stages, where the first stage is a reduction from all cores, and the second stage is a broadcast of the reduced value to all cores.

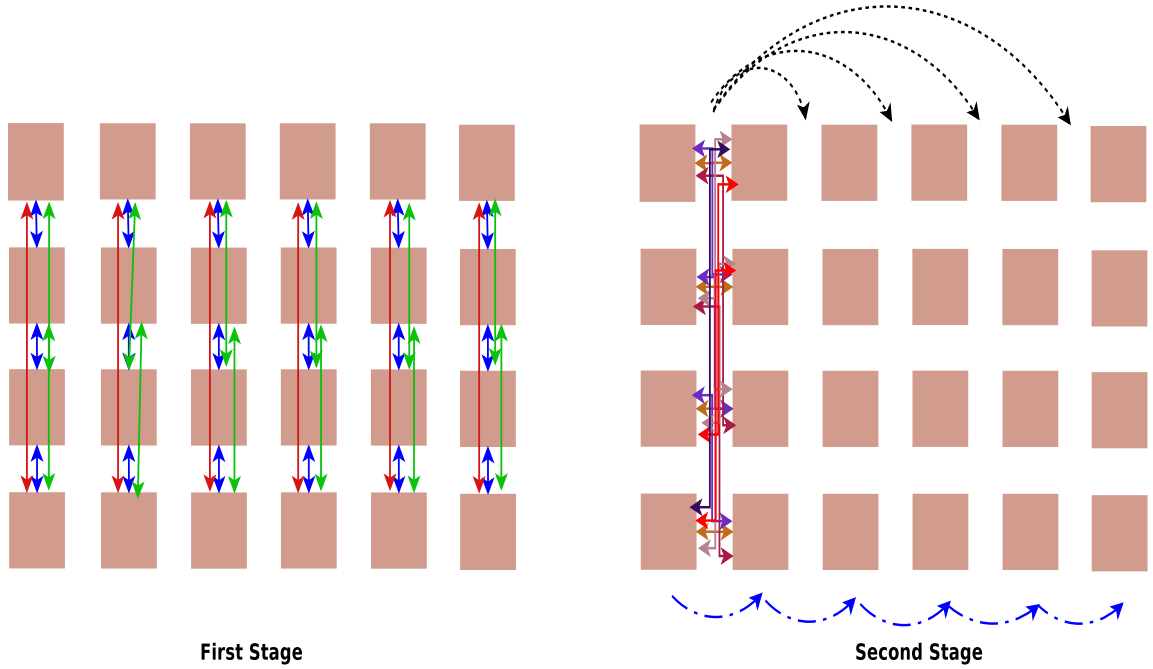


Figure 4.3: Communication pattern of Alltoall and Alltoallv on Intel SCC

4.3 Alltoall and Alltoallv

The alltoall collective is used to communicate to all the cores, fixed sized messages, each with a different content per source/destination pairs. It is a variation of the reduction pattern. Every core exchanges a fixed size message with every other core. The messages to be sent and received are stored in the buffers in the order of the core ranks. The communication is performed column-wise first and then row-wise as opposed to the reduction primitive. This facilitates the construction of the complete message and also ensures absence of contention.

The alltoallv primitive is a variant of the alltoall collective, where every core sends a different sized message to every other core. It propagates the messages using the same pattern as the alltoall collective but messages originate at different offsets of the array and are of varying sizes. This collective involves a large number of discontinuous memory references that can affect the performance of the processes.

Chapter 5

Implementation

The implementation of our collectives on the SCC, has been released using the RCCE library licensed under Apache Version 2.0 as the base library. We made this choice based on the analysis of performance difference derived from figures 1.2(b) and 1.3(b). The RCCE and iRCCE perform similarly, but we chose the RCCE library due to its larger chunk size and blocking characteristic. We refer to the implementations as NoC_Msg. All collectives of NoC_Msg use one-sided communication interfaces, *put()* and *get()*, to transfer messages from one MPB to the other. We use one-sided communication interfaces for the collectives as opposed to two-sided communication interfaces (*send()* and *recv()*) because the former are two orders of magnitude faster than the latter [9]. The *RCCE_put()* stores a message in the core's own MPB, and the *RCCE_get()* retrieves a message from a remote core's MPB. This makes *RCCE_get()* slightly more expensive than *RCCE_put()*.

The collectives based on broadcast are implemented in a pipelined manner, where a message is communicated as sequence of flits rather than as a whole over the NoC, i.e., cores receive a flit and forward the flit to the next core rather than waiting for all the flits to be received. The pipelining reduces the overall latency and increases the throughput for all collectives as the cores do not wait for flits after receiving the first flit. The collectives based on reduction are partially pipelined. The chunk size used is the default chunk size of RCCE, i.e., 8092 bytes. It is actually the size of the MPB, minus the 100 bytes reserved for the header overhead. If the message is smaller than the chunk size, we send the entire message as a flit. The API for all collectives are similar to the MPI API. Along with the collectives, blocking point-to-point functions, similar to MPI, are also implemented for NoC_Msg to provide basic functionality. NoC_Msg interfaces for *init()* and *finalize()* call the *RCCE_init()* and *RCCE_finalize()*, respectively.

5.1 Initialization

Algorithm 1 Initialize

```
1: procedure INIT
2:     root ← rank of the root core
3:     rank ← own core rank on the mesh
4:     chunksize ← size of message chunk
5:     chunk ← message chunks
6:     noofprocs ← number of processes
7:     row_cores ← number of cores in row
8:     col_cores ← number of cores in column
9:     total_rows ← number of rows
10:    total_cols ← number of columns
11:    my_row ← row of the core
12:    my_col ← column of the core
```

The *init()* initializes the global variables required by every collective which define the total number of processors, number of processors in every row and column, processor rank, row number and column number. Any programs to be executed on the SCC with NoC_Msg use the *rccerun* script(similar to *mpirun*) provided by RCCE. We describe the algorithms for our collectives below.

Every core after entering the *main()*, calls the *init()* function to initialize its global variables and also to process any command line arguments. These global variables are used by all collectives and point to point communication functions.

5.2 Broadcast

Algorithm 2 Broadcast

```
1: procedure BCAST
2:     buf ← the buffer for broadcast
3:     sendsize ← size to be sent
4:     root ← rank of the root core
5:     while chunk < sendsize do
6:         if rank%2 = root%2 then
7:             if rank/row_cores = root/row_cores then
8:                 if rank < root then
9:                     Recv chunksize from right tile
10:                    Send chunksize to left tile
11:                else if rank > root then
12:                    Recv chunksize from left tile
13:                    Send chunksize to right tile
14:                else if rank = root then
15:                    Send chunksize to left tile
16:                    Send chunksize to right tile
17:            else if rank/row_cores > root/row_cores then
18:                Recv chunksize from tile below
19:                if rank + 12 < noofprocs then
20:                    Send chunksize to tile above
21:            else if rank/row_cores < root/row_cores then
22:                Recv chunksize from tile above
23:                if rank − row_cores ≥ 0 then
24:                    Send chunksize to tile below
25:        else
26:            Recv chunksize from other core in the tile
27:        chunk = chunk + chunksize
```

The *bcast()* interface accepts a buffer for sending and receiving, the number of elements to be broadcasted and the datatype of the elements. The rank of the root core is also passed as a parameter. Based on the datatype and the number of elements, the number of bytes to be sent and received is calculated. The root core sends the message from the buffer whereas all other cores participating in the broadcast receive the message in the buffer. The root core sends the message to its neighboring cores, which propagate the chunk to their neighboring cores, eventually sending the message to every core. The broadcast algorithm works in a pipelined manner, i.e., the cores receive and send parts of message one after the other rather than whole

message at once. This increases the throughput with same latency.

5.3 Reduction

In a reduction, one buffer is used for sending and another buffer is used for receiving. All cores in the reduction send flits based on the number of elements to communicate and the datatype of the elements. All values are reduced at every stage of the algorithm by an arithmetic operation specified as a parameter. These are basic arithmetic operations like SUM, AVG, MIN, MAX and PROD. In *reduce()*, the root receives the final reduced values in the receive buffer. The receive buffer of other cores is used as an auxiliary buffer for the reduction of elements.

Algorithm 3 Reduce

```
1: procedure REDUCE
2:     buf ← the buffer for reduction
3:     sendsize ← size to be sent
4:     op ← arithmetic operation SUM, MAX, MIN, etc.
5:     while chunk < sendsize do
6:         if rank%2 = root%2 then
7:             Recv chunksize from other core in the tile
8:         else
9:             Send chunksize to other core in the tile
10:        chunk = chunk + chunksize
11:    if rank%2 = root%2 then
12:        Perform op on buf
13:        if rank/row_cores > root/row_cores then
14:            if rank + row_cores < noofprocs then
15:                Recv buf from tile above
16:            Perform op on buf
17:            Send chunksize to tile above
18:        else if rank/row_cores < root/row_cores then
19:            if rank − row_cores ≥ 0 then
20:                Recv buf from tile below
21:            Perform op on buf
22:            Send buf to tile above
23:        else if rank/row_cores = root/row_cores then
24:            if rank + row_cores < noofprocs then
25:                Recv buf from tile above
26:            if rank − row_cores ≥ 0 then
27:                Recv buf from tile below
28:            Perform op on buf
29:            if rank < root then
30:                Recv buf from left tile
31:                Perform op on buf
32:                Send buf to right tile
33:            else if rank > root then
34:                Recv buf from right tile
35:                Perform op on buf
36:                Send buf to left tile
37:            else if rank = root then
38:                Recv buf from left tile
39:                Perform op on buf
40:                Recv buf from right tile
41:                Perform op on buf
```

5.4 Allreduction

Algorithm 4 Allreduce

```
1: procedure ALLREDUCE
2:     buf ← the buffer for reduction
3:     sendsize ← size to be sent
4:     root ← rank of the root core
5:     op ← arithmetic operation SUM, MAX, MIN, etc.
6:     Reduce the buf to a root reduce()
7:     Broadcast the buf from the root bcast()
```

In *allreduce()*, the final reduced values are broadcasted to all the cores from the root and every core obtains the values in the receiving buffer. The *reduce()* function is partially pipelined, so a marginal increase in throughput is observed but the latency for reduction is shortened. For *allreduce()*, the first stage of reduction is partially pipelined and the second stage of broadcast is completely pipelined. This again provides an increase in the throughput and the latency decreases slightly. However, the internal function call adds a small overhead to the performance.

5.5 Barrier

Algorithm 5 Barrier

```
1: procedure BARRIER
2:     flag ← data structure for signalling arrival
3:     if rank%2 = root%2 then
4:         Recv flag from other core in the tile
5:         if rank/row_cores > root/row_cores then
6:             if rank + row_cores < noofprocs then
7:                 Recv flag from tile above
8:                 Send flag to tile above
9:         else if rank/row_cores < root/row_cores then
10:            if rank − row_cores ≥ 0 then
11:                Recv flag from tile below
12:                Send flag to tile above
13:        else if rank/row_cores = root/row_cores then
14:            if rank + row_cores < noofprocs then
15:                Recv flag from tile above
16:            if rank − row_cores ≥ 0 then
17:                Recv flag from tile below
18:            if rank < root then
19:                Recv flag from left tile
20:                Send flag to right tile
21:            else if rank > root then
22:                Recv flag from right tile
23:                Send flag to left tile
24:            else if rank = root then
25:                Recv flag from left tile
26:                Recv flag from right tile
27:    else
28:        Send flag to other core in the tile
```

In *barrier()*, cores synchronize at a common starting point in execution. It is a useful method for parallel processing applications. First, all the odd-ranked cores signal the even-ranked cores. Then the even-ranked cores signal their neighboring even-ranked core in the tile above or below based on the relative position from the root core. The even-ranked cores in the row of the root receive a message from the even-ranked cores in the neighboring tiles, that are located above and below them. These cores then send a message to the neighboring left or right tiles based

on the relative position from the root. The root, being an even-ranked core receives the final messages from its neighboring tiles in its row. The cores leave the barrier as soon as they send a message to their neighbors. The messages are light-weight messages due to a zero payload. Thus, the barrier is fast and consumes minimal computation time.

5.6 Alltoall

Algorithm 6 Alltoall

```
1:  procedure ALLTOALL
2:      sendbuf ← the buffer for sending
3:      recvbuf ← the buffer for receiving
4:      sendsize ← size to be sent
5:      recvsize ← size to be received
6:      while chunk < (sendsize & recvsize) do
7:          Copy chunksize from sendbuf to recvbuf
8:          Send from sendbuf to neighboring core in tile
9:          Recv in recvbuf from neighboring core in tile
10:         i=1
11:         while i < total_rows do
12:             Send from sendbuf to tiles above and below
13:             Recv in recvbuf from tiles above and below
14:             i++
15:         i=1
16:         while i < total_cols do
17:             if i = my_col then
18:                 Send from sendbuf to tiles in my_col + i
19:                 Recv in recvbuf from tiles in my_col + i
20:             i++
21:         chunk = chunk + chunksize
```

The *alltoall()* primitive uses a buffer each for sending and receiving. Every core sends an equally sized message determined by the number of elements and their datatypes. The algorithm works in two stages, where in the first stage all the cores in every column exchange messages from their respective buffers. In the second stage, cores from every column exchange messages with the cores from every other column. The second stage starts the exchange from the leftmost column. It communicates with columns on the right in a row-wise manner, where it exchanges messages with the rows below a tile first and then with the rows above it. Both the stages are completely pipelined, which increases the throughput and reduce the latency.

5.7 Alltoallv

Algorithm 7 Alltoallv

```
1: procedure ALLTOALLV
2:     sendbuf ← the buffer for sending
3:     recvbuf ← the buffer for receiving
4:     sendsize ← array of sizes to be sent
5:     recvsize ← array of sizes to be received
6:     sdispls ← array of positions in sendbuf
7:     rdispls ← array of positions in recvbuf
8:     while chunk < (sendsize & recvsize) do
9:         Copy chunksize from sendbuf to recvbuf
10:        Send from sendbuf to neighboring core in tile
11:        Recv in recvbuf from neighboring core in tile
12:        i=1
13:        while i < total_rows do
14:            Send from sendbuf to tiles above and below
15:            Recv in recvbuf from tiles above and below
16:            i++
17:        i=1
18:        while i < total_cols do
19:            if i = my_col then
20:                Send from sendbuf to tiles in my_col +i
21:                Recv in recvbuf from tiles in my_col + i
22:            i++
23:        chunk = chunk + chunksize
```

The *alltoallv()* is similar to *alltoall()* but has the capability of sending and receiving varying message sizes between pairs of cores. The number of bytes to send and receive from each core are specified in two different arrays whose size is equal to the number of cores. Thus, one may send bytes from any point in the buffer and also receive at any point in the buffer. The *alltoallv()* primitive also exchanges messages with cores within its columns first, then exchanges messages with cores in the remaining columns. The displacements for sending and receiving are also given in two different arrays. The final message is received by all the cores in the recvbuf, which is big enough to accommodate all the varied size messages from every core.

Chapter 6

Experimental Evaluation

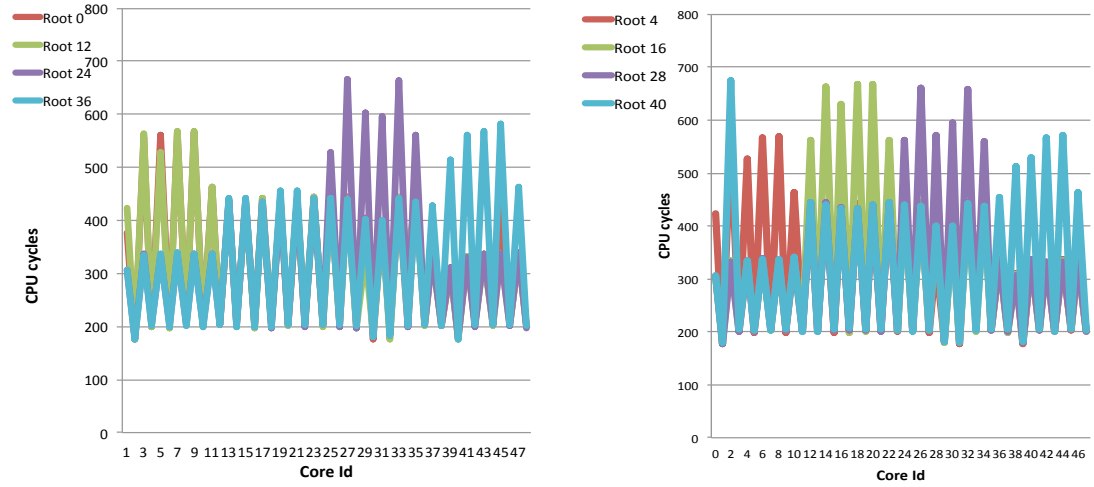
The collective implementations have been tested and benchmarked for the Intel SCC platform. The benchmarks were created to compare the performance of *NoC_Msg* collectives, with collectives provided by other libraries such as RCKMPI and RCCE. We have assessed the performance of our benchmarks on different configurations for the SCC. The benchmarks were created to evaluate the performance of the collectives based on the metrics latency, throughput and scaling.

6.1 Experimental Setup

All experiments were performed on the Intel SCC 48-core platform. The SCC hardware was provided by the Intel MARC group [36]. The SCC was configured to run with a core frequency of 800MHz, a tile frequency of 1600MHz, a mesh frequency of 1066MHz and a global clock that oscillates at 1600MHz. The SCC was configured to run at mesh frequency of 533MHz and other with the same frequencies to compare the *bcast()* with the oc-bcast implementation [7].
For latency and throughput experiments, we benchmarked the collectives by timing every run of multiple message exchanges using the collectives of varying sizes of all 48 cores. The memory for the messages was pre-allocated from a large array and different offsets within the same array were sent for each run to avoid the effects of caching on the latency and throughput. However, the effects of L2 cache size can be observed as some of the memory locations of the previous runs remain in L2 cache. The message sizes varied from 4 bytes to 1MB per core. For *alltoall()* and *alltoallv()*, we use message sizes from 192 bytes to 12MB per core. For scaling experiments, we timed all the collectives for multiple runs with different number of cores by passing a message size of 256KB per core, except for *alltoall()* and *alltoallv()*, where we used a message size of 192KB per core. For *barrier()*, the payload is zero. The range of the message sizes were large enough to cover all possible sizes. Every collective was timed for 10,000 different runs in a loop using the time stamp counter of the SCC. The time was measured in cycles for each run separately. The timing of 10,000 runs were averaged and plotted to evaluate the performance of collectives based on latency and throughput.

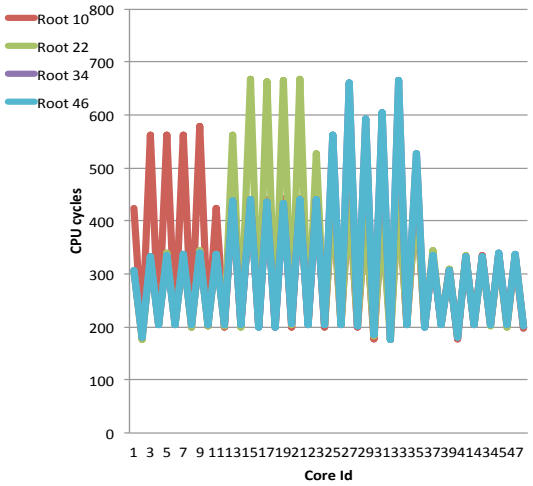
6.2 Experimental Results and Observations

6.2.1 Calibration Experiments

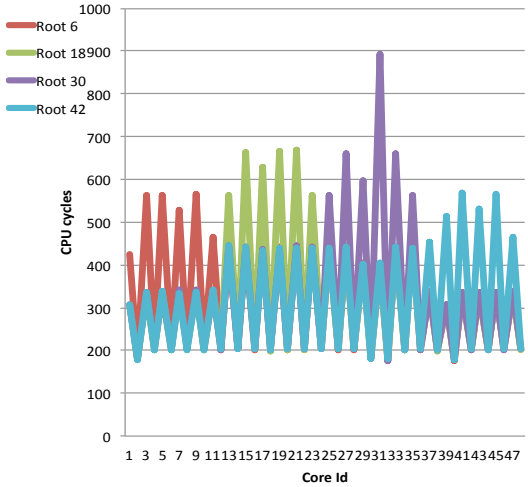


(a) Root at beginning of the row

(b) Root at end of the row



(c) Root at left middle of the row



(d) Root at right middle of the row

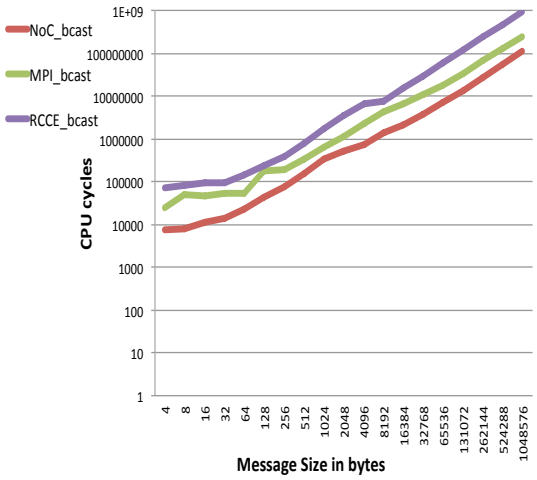
Figure 6.1: Analysis for the position for the root of collective operations using *barrier()* of NoC_Msg

All results were obtained after calibration experiments to achieve maximum throughput in a two-way communication between any pair of cores, to decide between pipelined or complete

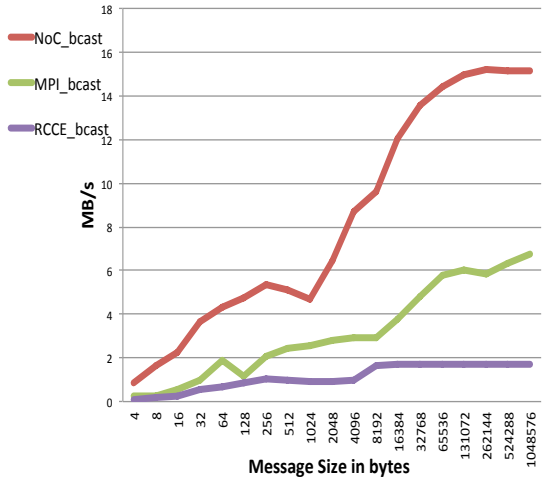
message transmission and to determine the placement of the root for all collectives and different number of cores. After optimizing and fine-tuning, results provided maximum fidelity for the performance experiments on the collectives reported here. Figures 6.1(a), 6.1(b), 6.1(c) and 6.1(d) show that the best results for the collective operations of NoC would be observed when the root core is in the right middle of the second or third row of the SCC. This will give a uniform performance to all the cores in the SCC.

In the latency graphs below, the y-axis shows the cycles consumed and the x-axis shows the message size in bytes. The y-axis of most of the latency graphs are logarithmic in nature, in order to accommodate all the data for all three libraries. The y-axis for the graph that describes the scaling of the throughput of *bcast()* for *NoC_Msg* is given in microseconds on a linear scale. For throughput graphs, the y-axis shows throughput in MB/s where as the x-axis again shows the message size. For scaling graphs, the y-axis shows cycles consumed where as the x-axis shows the number of cores.

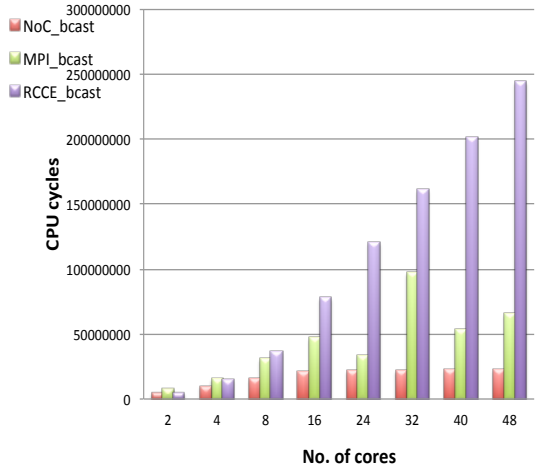
6.2.2 Broadcast results



(a) Latency for 48 cores



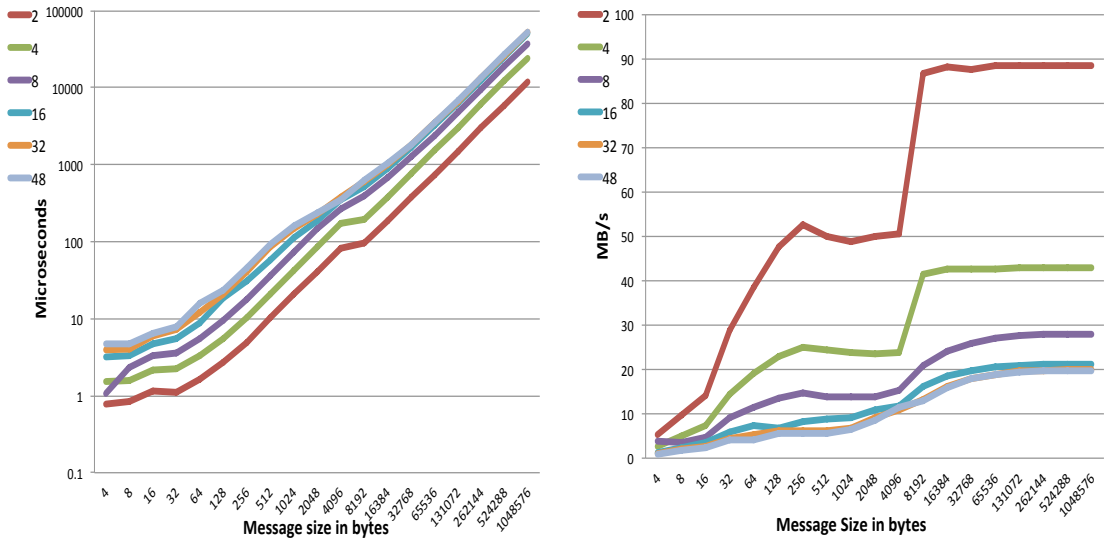
(b) Throughput for 48 cores



(c) Scaling for 256KB message

Figure 6.2: Performance analysis for *bcast()* of NoC_Msg, RCKMPI and RCCE

The performance for *bcast()* is depicted in Figure 6.2. Figure 6.2(a) depicts that the latency for *bcast()* for NoC_Msg is one order of magnitude faster than RCCE and half an order of magnitude faster than RCKMPI. The throughput improves by almost 150% (see Figure 6.2(b)). The throughput increases due to higher amount of data being communicated with less increase in latency. This is due to the simultaneous transmission of messages on the NoC without any



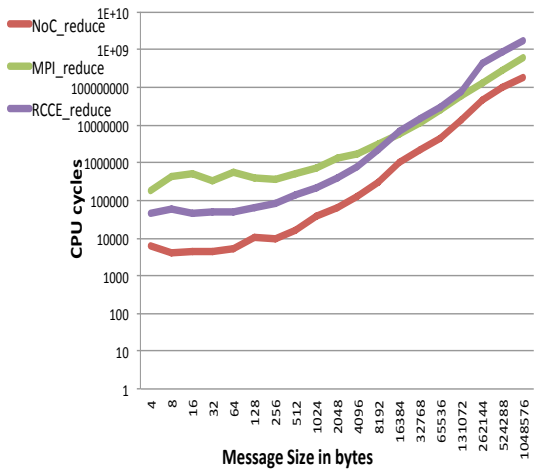
(a) Latency scaling for NoC_Msg

(b) Throughput scaling for NoC_Msg

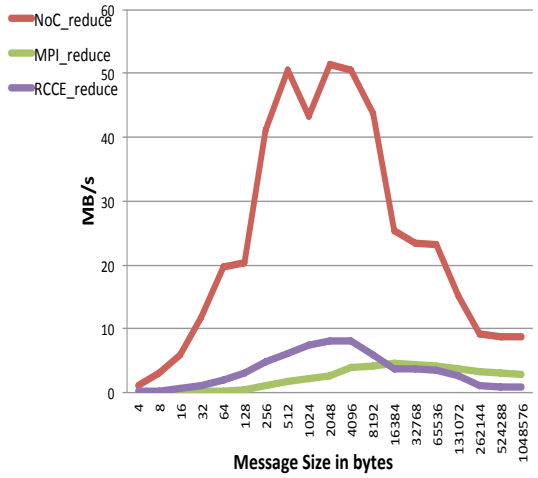
Figure 6.3: Scaling analysis for *bcast()* of NoC_Msg

contention, hence, reduced overhead. *bcast()* also scales well as the timing remains almost constant for any number of cores (see in Figure 6.2(c)). The scaling is observed both in terms of message sizes and number of cores. An outlier is seen in the graphs for a message size of 1KB, which is due to the chunk size we use to transmit a message. It causes an extra flit to be sent, due to protocol headers which causes a sudden increase in latency. The latency and throughput improvement of the *bcast()* is quite high when number of cores is up to 16 (see Figure 6.3(a) & 6.3(b)). Beyond that, the throughput remains constant for any number of cores. The performance compared to oc-bcast is better for smaller messages and cores less than 32. However, for higher number of cores and larger message sizes the throughput is slightly less than oc-bcast.

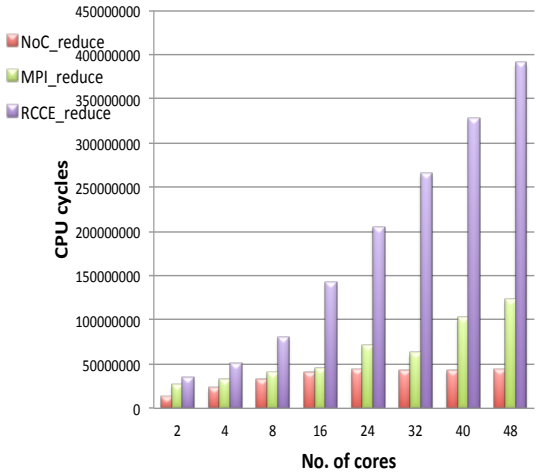
6.2.3 Reduction results



(a) Latency for 48 cores



(b) Throughput for 48 cores

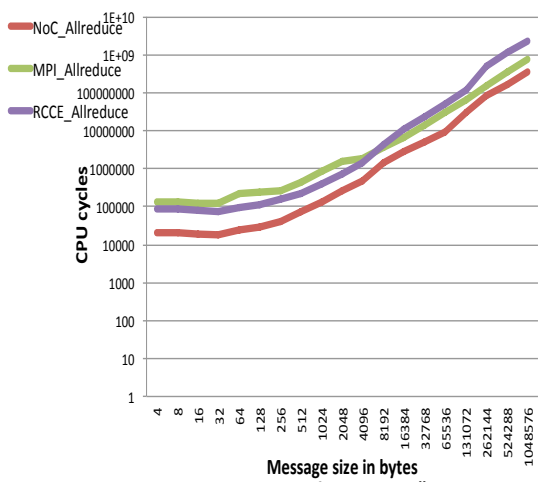


(c) Scaling for 256KB message

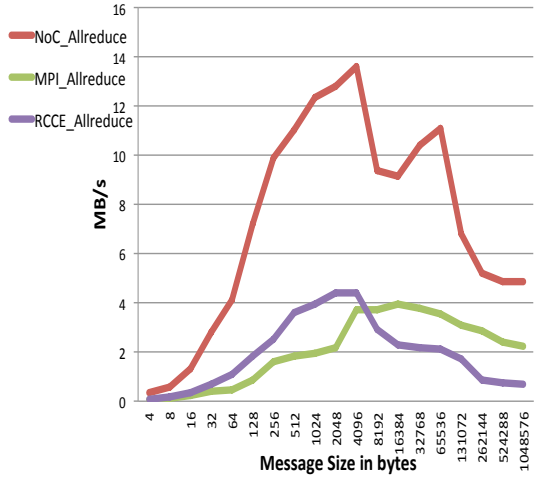
Figure 6.4: Performance analysis for *reduce()* of NoC_Msg, RCKMPI and RCCE

The *reduce()* primitive was assessed for performance using the SUM operator over the whole buffer passed by every core. In Figure 6.4(a), the y-axis shows the CPU cycles consumed on average for a particular message size to be reduced. Figure 6.4(b) depicts the throughput of *reduce()* for different message sizes. The performance for *reduce()* for NoC_Msg is also improved by almost 400% in comparison to RCCE and RCKMPI. The throughput drops at higher mes-

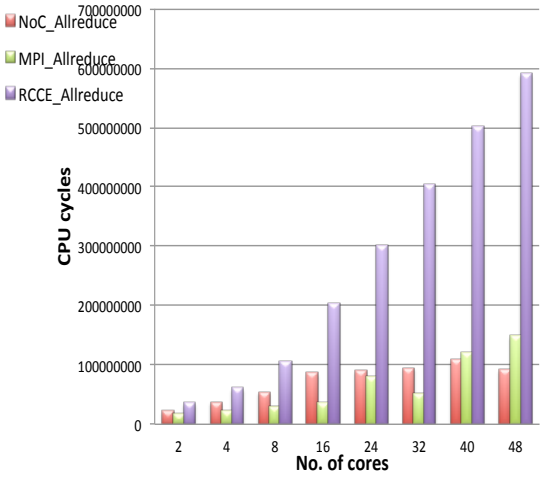
sage sizes due to lack of L2 caching after exceeding 256KB. However, the throughput at its peak is five times better than that of any other libraries. The increase in throughput is due to the partially pipelined behavior and the effect of L1 instruction cache. Due to the instruction cache, the arithmetic operations are executed quickly and for every operation, fewer cycles are consumed. Figure 6.4(c) shows that *reduce()* of NoC_Msg scales well for any number of cores consuming only 12% of the cycles required by RCCE.



(a) Latency for 48 cores



(b) Throughput for 48 cores

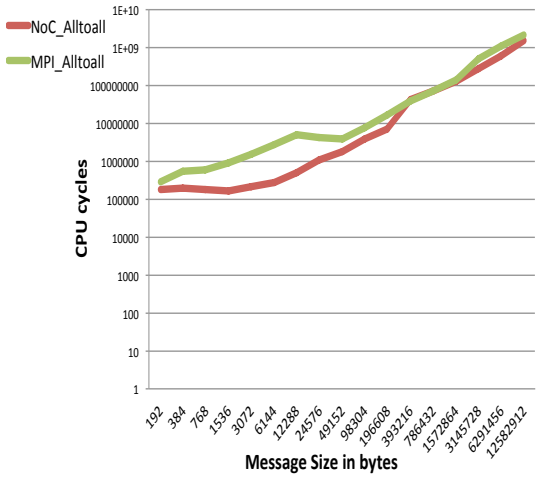


(c) Scaling for 256KB message

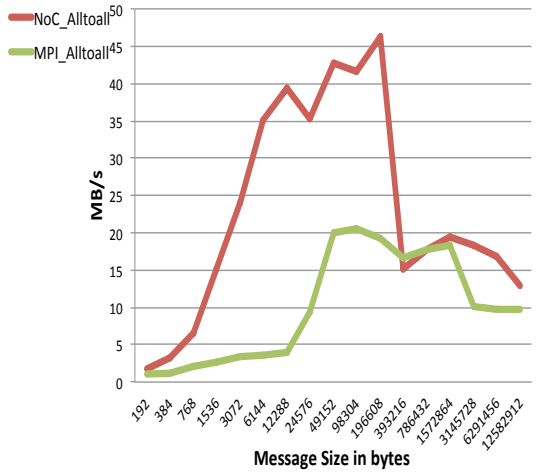
Figure 6.5: Performance analysis for *allreduce()* of NoC_Msg, RCKMPI and RCCE

6.2.4 Allreduction results

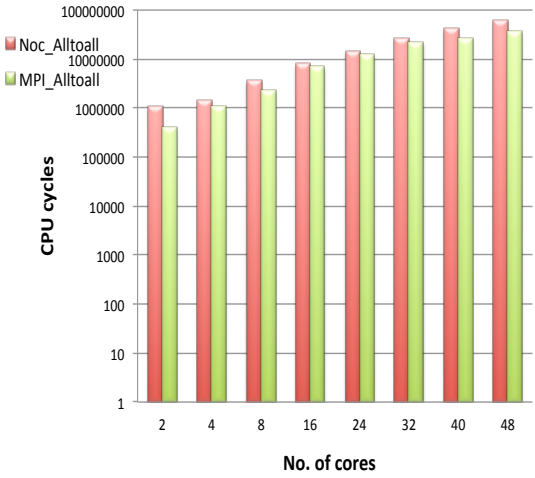
The combined effect of the improvement in performance of *bcast()* and *reduce()* can be seen in *allreduce()* for NoC_Msg (see Figures 6.5(a) and 6.5(b)). The throughput is again better than RCCE and RCKMPI for *allreduce()* with a maximum increase of 200% for a message size of 4KB, and latency is one order of magnitude lower than that of the others. However, the internal function call to *bcast()* results in a few extra cycles for *allreduce()*. This limits the increase in throughput to a certain extent.



(a) Latency for 48 cores



(b) Throughput for 48 cores



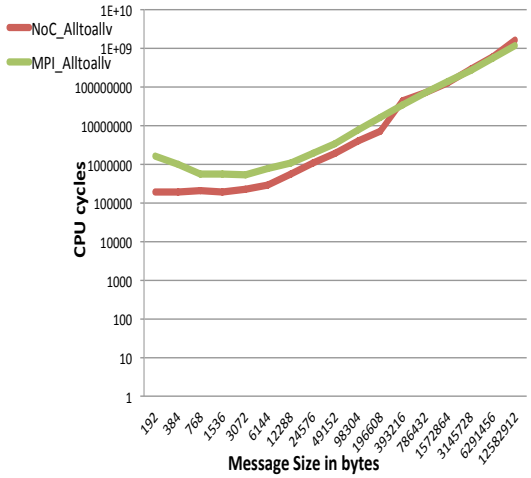
(c) Scaling for 192KB message

Figure 6.6: Performance analysis for *alltoall()* of NoC_Msg, RCKMPI and RCCE

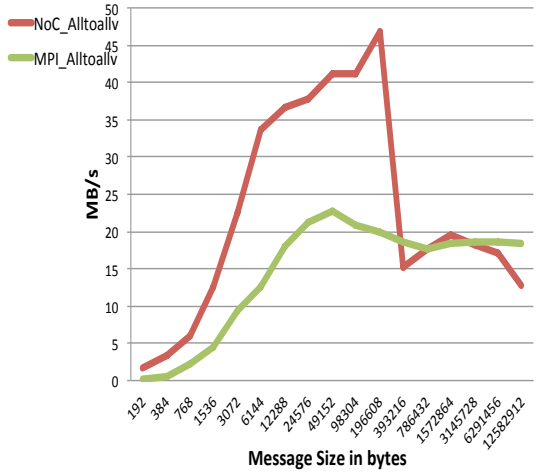
6.2.5 Alltoall and Alltoallv results

The performance of NoC_Alltoall and NoC_Alltoallv is compared with RCKMPI. Figures 6.6(a) and 6.7(a) show the latency graphs for both the collectives. The performance of *alltoall()* and *alltoallv()* is similar and shows 50% reduction in the cycles consumed compared to RCKMPI for smaller messages below 8KB. Both collectives achieve 2.5 times the maximum throughput of RCKMPI as seen in Figures 6.6(b) and 6.7(b). The implementations for *alltoall()* and *alltoallv()* have a potential for contention only on a single occasion throughout the algorithm, i.e., when cores in the same column exchange messages with cores, which are at a distance of 2 tiles

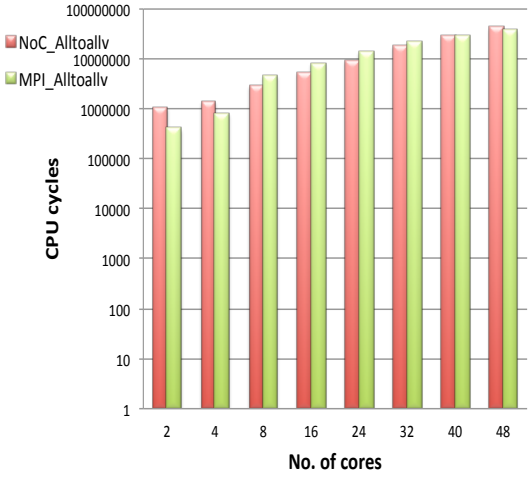
below or above in the Y-direction (see Figure 4.3). However, the cores communicate in the same direction and the chance for contention is minimal as the cores have one link to cover before communication over the link with contention. The effect is neither observed in Figure 6.6(b) nor in Figure 6.7(b), which show the throughput in MB/s for different message sizes. The throughput increases around 100% for message sizes for 192KB. The L2 cache effect is observed here as well: once the message size exceeds L2 cache size, both operations are better or match the equivalent ones of RCKMPI. The scaling performance of these two collectives is similar to RCKMPI because once the message reaches the size of the L2 cache, both implementations have the same latency(see Figures 6.6(c) and 6.7(c)).



(a) Latency for 48 cores



(b) Throughput for 48 cores



(c) Scaling for 192KB message

Figure 6.7: Performance analysis for *alltoallv()* of NoC_Msg, RCKMPI and RCCE

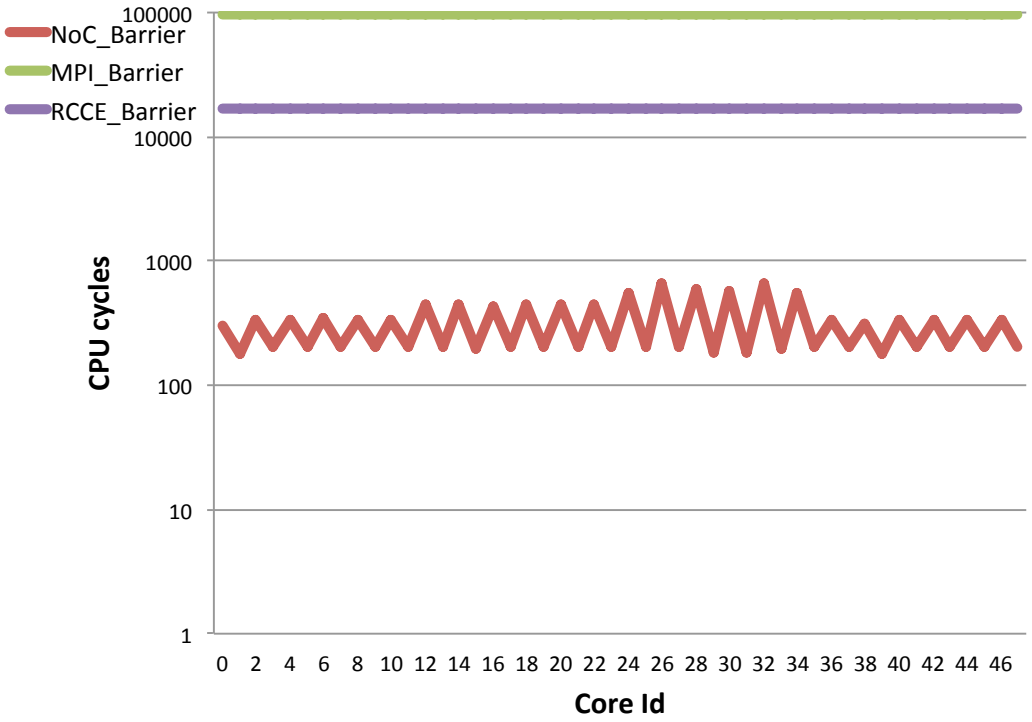


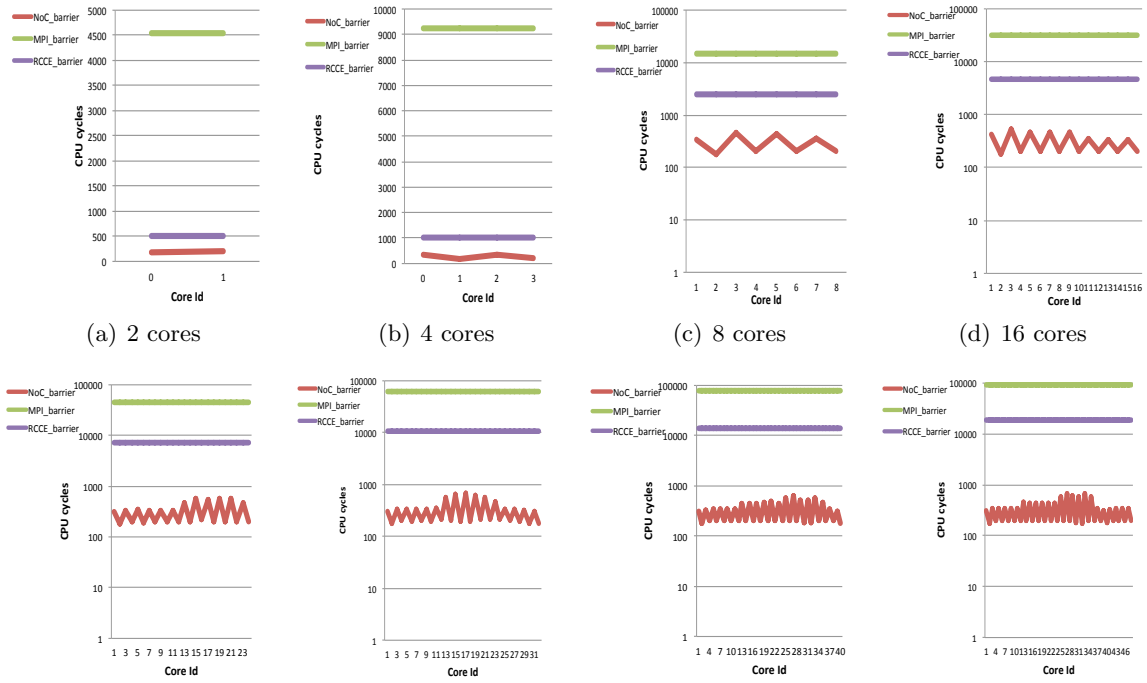
Figure 6.8: Comparison of *barrier()* for NoC_Msg, RCCE and RCKMPI

6.2.6 Barrier results

barrier() for NoC_Msg shows a significant improvement over RCKMPI and RCCE as seen in Figure 6.8. The y-axis denotes the CPU cycles consumed and the x-axis shows the core id. It is two orders of magnitude faster than RCCE and almost three orders of magnitude faster than RCKMPI. It follows a saw-tooth shape because of differences between even ranked core and odd ranked core communication.

As described in Chapter 5, the odd ranked cores communicate with the even ranked cores in their tile. Hence, they leave the barrier sooner than the even ranked cores. The even ranked cores then leave the barrier based on their position from the root. Even if the cores leave the barrier at slightly different times, the difference is a few hundred cycles which is insignificant. The performance of *barrier()* for NoC is consistently better than RCCE and RCKMPI when scaled from 2 to 48 cores (see Figure 6.9).

All the performance improvements observed in the above graphs, are the result of only avoiding core and link contention. The performance of RCKMPI and RCCE reflect the degrading effect of contention as they achieve only around 10% of maximum throughput that is available on the NoC of the SCC. The *NoC_Msg* achieves twice the performance of RCCE and RCKMPI with its contention free approach. All the collectives show tremendous improvement in throughput



(a) 2 cores (b) 4 cores (c) 8 cores (d) 16 cores

(e) 24 cores (f) 32 cores (g) 40 cores (h) 48 cores

Figure 6.9: Scaling performance for *barrier()* of NoC_Msg, RCKMPI and RCCE

and latency when compared to the two default libraries on the SCC. This improvement is due to absence of link and core contention in the communication patterns of the the collectives. The collectives also perform well when scaled.

Chapter 7

Future Work

The area of inter-process communication on NoCs offer ample potential for improvement. Many collectives can be implemented based on the MPI-2 standard in a contention-free manner. This would provide efficient communication means for the massive multi-core architectures.
The inter-process communication methods described above can be used at the operating system kernel level to achieve many critical tasks [19]. Dedicated cores can utilize the collectives to understand and direct the load balancing and sharing between a group of cores. They can also be used for task allocation and process scheduling over massive multi-cores. The collectives can be used to effectively use all the resources on the SoCs and keep every core as busy as possible. They can be used to effectively power down and power up cores according to task requirements [35]. This will help reduce the power consumption of the multi-core architectures and increase its use in more power constraint applications. The analysis of the power consumption of the SoCs, while using the collective methods on the NoCs, is an area of further studies.
Apart from contention, the SCC provides many opportunities where communication performance can be increased. The basic assumption of how the MPB works in RCCE can be modified and more efficient usage of the shared memory architecture can be realized. Direct insertion of data into any core's memory space on the SCC in a contention free manner can result in lightweight communication [32]. This will reduce copy overheads on MPB and improve the performance of one-sided communications on the SCC. Timing analysis of data insertion into a remote core's memory space is also an interesting field to work.
Highly efficient collective operations can be used for process synchronization using extremely light payloads. This can eliminate the usage of difficult techniques with high overhead that achieve task synchronization, e.g., shared memory signaling [20]. SoCs like the SCC, where each core can run a different operating system, can use the NoC as high-speed interconnect to service inter-server requests and eliminate the usage of the TCP/IP protocol stack. Collective libraries can also be utilized at the user level within an application to achieve high performance.

Chapter 8

Conclusion

As massive multi-core processors are growing in size, it becomes imperial to have efficient communication means for processes. The operating systems should adapt to these methods to achieve maximum performance possible. Link and core contention exist on SoC's like Intel SCC. This contention may degrade the performance of multi-core architectures. This reduction in performance increases manifold when collective communication takes place based on methods that contain contention. The results indicate that contention on the NoC can be a big hurdle for massive multi-core architectures to obtain efficient and high performance inter-process communication. When both link and core contention are avoided, the performance increases by a factor of at least two. A 2x to 9x improvement in throughput is achieved for collective operations when contention is avoided. We believe that contention-free inter-process communication techniques need to be included in future operating system and runtime designs. This will ensure low latency and high throughput of communication between processes on massive multi-core architectures. The actual performance can effectively approach theoretical peak performance if efficient and nearly contention-free communication is provided for the processes, even as the number of cores is scaled up.

With such promising results, we effectively say that our hypothesis which says that "If contention on mesh-based network-on-chip interconnects of multi-core architectures is avoided, performance of collective communication with their underlying point-to-point messages can be significantly improved", is proved.

References

[1] Yagna, Karthik. "Efficient Collective Communication for Multi-core NOC Interconnects." (2013).

[2] Zimmer, Christopher John. "Bringing Efficiency and Predictability to Massive Multi-core NoC Architectures." (2012).

[3] C. Zimmer and F. Mueller, "Nocmsg: Scalable noc-based message passing", in International Symposium on Cluster, Cloud and Grid Computing, 2014.

[4] Mattson, Timothy G., et al. "The 48-core SCC processor: the programmer's view." Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society, 2010.

[5] Salihundam, Praveen, et al. "A 2 tb/s 6 4 mesh network for a single-chip cloud computer with dvfs in 45 nm cmos." Solid-State Circuits, IEEE Journal of 46.4 (2011): 757-766.

[6] Furst, John-Nicholas, and Ayse K. Coskun. "Performance and power analysis of RCCE message passing on the intel single-chip cloud computer." 4th Many-core Applications Research Community (MARC) Symposium. 2011.

[7] Petrovi, Darko, et al. "Asynchronous broadcast on the intel scc using interrupts." Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium. 2012.

[8] Clauss, Carsten, et al. "iRCCE: a non-blocking communication extension to the RCCE communication library for the Intel Single-chip Cloud Computer." Chair for Operating Systems, RWTH Aachen University (December 17, 2010) (2011).

[9] Mattson, Tim, and Rob van der Wijngaart. "RCCE: a small library for many-core communication." Intel Corporation, May (2010).

[10] Urena, Isaias A. Compres, Michael Rieper, and Michael Konow. "RCKMPI-lightweight MPI impentation for Intel's Single Chip Cloud Computer(SCC)." Recent Advances in the Message Passing Interface. Springer Berlin Heidelberg, 2011. 208-217.

[11] Geist, Al, et al. "MPI-2: Extending the message-passing interface." Euro-Par'96 Parallel Processing. Springer Berlin Heidelberg, 1996.

[12] Gropp, William. "MPICH2: A new start for MPI implementations." Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer Berlin Heidelberg, 2002. 7-7.

[13] Clauss, Carsten, Stefan Lankes, and Thomas Bemmerl. "Performance Tuning of SCC-MPICH by means of the Proposed MPI-3.0 Tool Interface." Recent Advances in the Message Passing Interface. Springer Berlin Heidelberg, 2011. 318-320.

[14] Chan, Ernie. "RCCE comm: A collective communication library for the Intel Single-chip Cloud Computer." online http://communities. intel. com/servlet/JiveServlet/previewBody/5629-102-1-8718/RCCE comm. pdf (2010).

[15] Aparna Chandramowlishwaran, Kamesh Madduri, and Richard Vuduc. Diagnosis, tuning, and redesign for multicore performance: A case study of the Fast Multipole Method. In Proc. ACM/IEEE Conf. Supercomputing (SC), New Orleans, LA, USA, November 2010.

[16] Kohler, Adan, et al. "Low-Latency Collectives for the Intel SCC." Cluster Computing (CLUSTER), 2012 IEEE International Conference on. IEEE, 2012.

[17] Kohler, Adan, and Martin Radetzki. "Latency-optimized Collectives for High Performance on Intel's Single-chip Cloud Computer."

[18] Howard, Jason, et al. "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS." Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International. IEEE, 2010.

[19] Zimmer, Christopher, and Frank Mueller. "Low contention mapping of real-time tasks onto tilepro 64 core processors." Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th. IEEE, 2012.

[20] Perez, Josep M., Rosa M. Badia, and Jesus Labarta. "A dependency-aware task-based programming environment for multi-core architectures." Cluster Computing, 2008 IEEE International Conference on. IEEE, 2008.

[21] Rauber, Thomas, and Gudula Runger. Parallel programming: For multicore and cluster systems. Springer Science & Business, 2013.

[22] Stallings, William. Operating systems: internals and design principles. Test, 2008.

[23] Patterson, David A., and John L. Hennessy. Computer organization and design: the hardware/software interface. Newnes, 2013.

[24] Tilera TilePro64. "www.tilera.com/sites/default/files/productbriefs/ TILEPro64_Processor_PB019_v4.pdf".

[25] Nvidia Tegra K1. "www.nvidia.com/object/tegra-k1-processor.html".

[26] Nvidia Titan Z. "www.nvidia.com/gtx-700-graphics-cards/gtx-titan-z".

[27] Adapteva Epiphany-IV "www.adapteva.com/docs/ e64g401_datasheet.pdf".

[28] Intel legacy processors "ark.intel.com/products/family/79667/ Legacy-Intel-Core2-Processor#@All"

[29] AMD server processors "www.amd.com/en-us/ products/server/6000/6300#"

[30] Olofsson, Andreas, et al. "A 1024-core 70 GFLOP/W floating point manycore microprocessor." Poster on 15th Workshop on High Performance Embedded Computing HPEC2011. 2011.

[31] Mattson, Tim. "The Future of Many Core Computing: A tale of two processors." Intel Labs Report (2010).

[32] van Tol, Michiel W., et al. "Efficient memory copy operations on the 48-core intel scc processor." 3rd Many-core Applications Research Community (MARC) Symposium. Vol. 7598. KIT Scientific Publishing, 2011.

[33] Moore, Gordon E. "Cramming more components onto integrated circuits." (1965).

[34] Rotta, Randolf, et al. "In-memory communication mechanisms for many-cores-experiences with the Intel SCC." TACC-Intel Highly Parallel Computing Symposium (TI-HPCS). 2012.

[35] Chaparro-Baquero, Gustavo A., et al. "Power-Efficient Schemes via Workload Characterization on the Intel's Single-Chip Cloud Computer." Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International. IEEE, 2012.

[36] Intel Many-core Application Research Community. "communities.intel.com/community/marc"