Jørgen Bele Reinfjell

Automatic Detection of Interconnect Topologies and Optimization of the Broadcast Operation for MPI-based systems

Master's thesis in MTDT
Supervisor: Jan Christian Meyer
July 2023

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

HPC LAB

Jørgen Bele Reinfjell

Automatic Detection of Interconnect Topologies and Optimization of the Broadcast Operation for MPI-based systems

Master's thesis in MTDT
Supervisor: Jan Christian Meyer
July 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Dedication

I want to express my appreciation to my supervisor, Jan Christian Meyer, for providing valuable guidance and for being an outstanding mentor throughout the work on this thesis. I also want to thank the people at the HPC lab and Prof. Anne C. Elster for giving me access to the HPC lab. Finally, I want to thank my family and friends for their support throughout the work on this thesis.

Problem description

This thesis aims to employ empirical measurements for the automatic detection of interconnect topologies. Furthermore, it examines the utility of those topologies for automatic performance tuning of MPI collective operations.

Abstract

In this thesis, pair-wise measurements are used to automatically detect the topology of a network using MPI. The PLogP model is applied to model the performance of communication between two ranks. The PLogP parameters between two ranks are found by a micro-benchmark that was created using the LGate test harness. Clustering techniques are applied to the results of the micro-benchmark and used to extract topology information on several levels. Using an automatic parameter search, many different topologies can be extracted.

Results show that applying clustering algorithms to the PLogP micro-benchmark measurements is accurate at detecting both network topology and node-internal topologies. We run tests with different measurement counts. Analysis shows how that has an impact on the runtime of the micro-benchmark and the impact on the level of detail. They also show that increasing the number of repetitions used to get our measurements can be used to increase the detail in the distance matrices.

We implement a topology-aware `MPI_Bcast` implementation that can use the topologies that were automatically detected. A micro-benchmark is used to find the best broadcast implementation from a set of algorithms and topologies for a given message size. We generate a parameterized version of the broadcast operation that uses the results for a given message size. Our results show a relative speedup ≥ 1.5 for up to $P = 512$ for several clusters when compared to OpenMPI and Intel MPI. We apply the technique for the single-node system ARM1 with a currently unknown topology and achieve a speedup of 1.5 over OpenMPI.

Table of Contents

Problem description i

Abstract iii

Table of Contents vii

List of Tables ix

List of Figures xii

Abbreviations xiii

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Scope . . . 1
- 1.3 Structure . . . 2

2 Background and related work **3**
- 2.1 Performance models . . . 3
  - 2.1.1 Fundamental equation of modeling . . . 3
  - 2.1.2 Scaling . . . 3
  - 2.1.3 Speedup . . . 4
- 2.2 Communication models . . . 4
  - 2.2.1 Hockney . . . 4
  - 2.2.2 Bulk Synchronous Parallel . . . 4
  - 2.2.3 LogP model . . . 5
  - 2.2.4 LogGP and Parameterized LogP (PLogP) . . . 5
- 2.3 Message Passing Interface (MPI) . . . 5
  - 2.3.1 Point-to-point communication . . . 6
  - 2.3.2 Communication protocols . . . 7
  - 2.3.3 Process rank and communication groups . . . 7

2.3.4 Collective communication . . . 7
2.3.5 MPI Implementations . . . 8
2.4 High Performance Computing . . . 8
2.4.1 Basic Linear Algebra Subprograms (BLAS) . . . 8
2.4.2 Nodes, clusters and Network topology . . . 8
2.4.3 Topology aware communication . . . 9
2.5 Clustering methods . . . 10
2.5.1 Metric spaces . . . 10
2.5.2 Vector spaces . . . 10
2.5.3 Sparse Spatial Selection (SSS) . . . 10
2.5.4 K-means . . . 10
2.5.5 K-medoids . . . 11
2.5.6 Spectral clustering methods . . . 11
2.5.7 Hierarchical clustering . . . 11
2.5.8 SSSTree . . . 12
2.5.9 DBSCAN . . . 13
2.5.10 HDBSCAN . . . 13
2.6 Related work . . . 14
2.6.1 Measurements of the LogP parameters . . . 14
2.6.2 Optimization of broadcast and other collective operations in MPI 14

3 Methodology **15**
3.1 PLogP measurements using LGate . . . 15
3.1.1 LGate . . . 15
3.1.2 PLogP measurements using LGate . . . 17
3.2 Optimizing network topologies . . . 22
3.3 Automated clustering parameter search . . . 24
3.4 Experiments . . . 24
3.5 Clustering algorithm performance . . . 25

4 Experimental setup **27**
4.1 Compilation . . . 27
4.2 SLURM and MPI . . . 27
4.3 Analysis and processing . . . 27
4.4 `HPC Lab 06 computer - HPCLAB` . . . 27
4.5 `IDUN` . . . 28
4.6 `FRAM` . . . 28
4.7 `BETZY` . . . 29
4.8 `IDUN-ARM1` . . . 29

5 Results and discussion **31**
5.1 Finding internal node topologies for a single node . . . 31
5.1.1 Outliers for first two ranks . . . 32
5.1.2 Cause of diagonal bands . . . 34
5.2 Finding inter-node topologies between two nodes . . . 34
5.3 Finding node interconnection topologies . . . 35

5.4 Determining the cluster topology for a large number of nodes 35
5.5 Discovery of currently unknown topology for ARM1 37
5.6 Find topology for BETZY 39
5.7 Baseline 40
5.8 Scaling 42
5.9 Clustering algorithms 42

6 Case study: topology aware MPI broadcast 45
6.1 Motivation 45
6.2 Background 46
6.2.1 Broadcast in OpenMPI and Intel MPI 46
6.3 Topology based optimization 47
6.3.1 Intra-node and inter-node optimization 47
6.3.2 Fat-tree-based communication 47
6.3.3 MagPIe 47
6.4 Implementation 47
6.4.1 Limitations 49
6.4.2 Algorithms 49
6.5 Methodology 50
6.5.1 Statistical error 50
6.5.2 Predicting and algorithmic selection of best broadcast function . . 50
6.6 Experiments 50
6.7 Results and discussion 51
6.7.1 FRAM 51
6.7.2 BETZY 53
6.7.3 IDUN 54
6.7.4 ARM1 55
6.8 Summary 56

7 Conclusion 59
7.1 Further work 60

Bibliography 61

Appendix 67
7.1.1 A - Clustering accuracy 67
7.1.2 B - Latency matrices 67

List of Tables

2.1 Relationship between PLogP and LogP/LogGP models 6
2.2 Point-to-point based communication functions in MPI 6
2.3 Some collective communication functions available in MPI 7

3.1 Table of experiments 25

6.1 List of base broadcast algorithms used 49
6.2 List of top-level broadcast algorithms used 49
6.3 Speedup for P=128,256 for OpenMPI and Intel MPI @ FRAM 53
6.4 Best broadcast algorithm for message size and P, OpenMPI @ FRAM * - non-blocking, `m` - using magpie 53
6.5 Speedup for P=512,1024,2048 for OpenMPI and Intel MPI @ BETZY . . . 54
6.6 Best broadcast algorithm for message size and P, OpenMPI @ BETZY * - non-blocking, `m` - using magpie 54

List of Figures

2.1 LogP (left) vs PLogP (right) message transmission diagram [1] 6
2.2 Fat-Tree topology with two layers and two nodes for each stem. 9
2.3 Star topology 10
2.4 Example dendrogram from network measurements on ARM1 12
2.5 Tree construction algorithm for SSSTree 13

3.1 Subset barrier algorithm based on dissemination used in LGate 16
3.3 Algorithm for joining leaves in the cluster tree 16
3.2 Algorithm for running all pairwise test 17
3.4 LGate topology file and tree representation 17
3.5 LogP measurements 19
3.7 busy() function used to measure clock overhead, same as used in [1] . . . 20
3.6 Algorithm for joining leaves in the cluster tree 21

4.1 Python packages and versions 28
4.2 HPCLAB system properties 28
4.3 IDUN system properties 28
4.4 FRAM system properties 29
4.5 BETZY system properties 29
4.6 ARM1 system properties 30

5.2 Variance for values of R 31
5.1 Communication time prediction for 1 node @ FRAM. The clear alternating pattern at R=64 and R=128 is caused by inter-socket communication. 32
5.3 Communication time prediction for 1 node, $R = 256$ @ FRAM. Using –distribution block:block 32
5.4 Two different rank assignments and alternative subset_barrier, m=0, P=32, R=128 @ FRAM 33
5.5 $RTT(0)$ prediction for 2 nodes @ FRAM. Here the same pattern as for 1 node appears, but the divide between the two nodes is also visible. Outliers $\geq$ 95th percentile are removed to maintain a reasonable color scale. . . . 34

5.6 Latency for 4 nodes @ FRAM. Here intra-node latency is apparent. Differences between runs are due to different node assignments. Outliers $\geq$ 95th percentile are removed to maintain a reasonable color scale. 36
5.9 RTT(0) with 1 core per node, R=64 @ FRAM 36
5.7 Latency for 8 nodes, R=64 @ FRAM. Here intra-node latency is apparent. Differences between runs are due to different node assignments. Outliers $\geq$ 95th percentile are removed to maintain a reasonable color scale. . . . 37
5.8 Communication time prediction for 4 nodes @ FRAM. Here intra-node latency is apparent. Differences between runs are due to different node assignments. Outliers $\geq$ 95th percentile are removed to maintain a reasonable color scale. 38
5.10 Predicted RTT(1) and latency, R=8192 @ ARM1 38
5.11 Predicted RTT(1) and latency, R=256 with –map-by core @ ARM1 . . . 38
5.12 HDBSCAN Clustering dendrogram from measurements on ARM1 39
5.13 RTT prediction for m=0,R=8,P=512 @ BETZY 39
5.14 Zoom-in to top left corner of Figure 5.13 40
5.15 Predicted RTT(0) @ HPCLAB 41
5.16 PLogP Latency, R=8196 @ HPCLAB 41
5.17 Core to core latency [2] 41
5.18 Time to run PLogP measurements for different numbers of nodes and repetitions, using all 32 cores of each node @ FRAM, M=2MiB 42
5.19 Some clusters found for two-nodes @ FRAM 43

6.1 Fat tree with 12 processes in 3 clusters 47
6.2 Communication ordering using clustering and binary tree bcast 48
6.3 Communication ordering using binary tree bcast 48
6.4 Selection of best broadcast implementation 50
6.5 Broadcast results using OpenMPI @ FRAM 51
6.6 IMPI - 4 nodes, 128 cores total @ FRAM, 32MB, 128MB, and 256MB] . 52
6.7 IMPI vs optimized - 4 nodes, 128 cores total @ FRAM] 52
6.8 Broadcast benchmark results using Intel MPI @ BETZY 54
6.9 Broadcast benchmark results using OpenMPI @ BETZY 55
6.10 16 nodes with 28 cores comparison of OpenMPI Bcast @ IDUN 55
6.11 Performance and speedup curve of optimized BCast using 2 and 4 clusters compared to OpenMPI default on ARM1. 50 warmup-rounds and 1000 repetitions. 56
6.12 Optimized broadcast decision function for ARM1 with 4 cluster topology 56

7.1 Clustering accuracy for values of R, P=32 @ FRAM 68
7.2 Clustering accuracy for values of R, P=64 @ FRAM 69
7.3 Clustering accuracy for values of R, P=128 @ FRAM 70
7.4 Clustering accuracy for values of R, P=512 @ BETZY 71
7.5 Clustering accuracy for values of R, P=96 @ ARM1 72
7.6 Latency P=32 @ FRAM 73
7.7 Latency P=128 @ FRAM 73
7.8 Latency P=96 @ ARM1 74

Abbreviations

MPI	=	Message Passing Interface
SLURM	=	Simple Linux Utility for Resource Management
S	=	speedup
RTT	=	Round-trip time
t	=	Time
L	=	Latency
o_s	=	PLogP send overhead parameter
o_r	=	PLogP recieve overhead parameter
g	=	PLogP gap parameter
P	=	number of processes
R	=	number of repetitions
M	=	largest message size
m	=	message size

Chapter 1

Introduction

In this chapter, we present the motivation, scope, and structure of this thesis.

1.1 Motivation

An understanding of the network topology is required to reach optimal performance in HPC applications. Empirical measurements of the network have the possibility of finding specific properties of that exact network that cannot easily be deduced from the known facts of the network, or that depend on the state of a continuously changing system.

1.2 Scope

We use the LGate pairwise test-harness to measure PLogP parameters of the interconnection between processes in MPI applications. The main goal of these measurements is to find patterns that reveal useful properties of the system. The measurements are used to define distance functions and distance matrices. We look at using clustering techniques from data mining and machine learning, in an attempt to extract useful information from the measurements. Several clustering methods for extracting useful topology information from these distance functions are applied. The information is used to create a network topology, which we apply in a topology-aware implementation of the MPI broadcast operation. The main reason for looking at the broadcast operation is that it is well researched and optimized implementations exist. We are interested in determining if the topologies we find can lead to performance improvements in such operations. Therefore we apply the technique to a system with an unknown architecture to demonstrate that it can be used to find useful topologies.

1.3 Structure

In Chapter 2 we present background information that is useful for understanding the topics covered in this thesis. Chapter 3 introduces related work. Chapter 4 covers an introduction to LGate, the PLogP micro-benchmark, automated clustering parameter search, definitions of metrics, and lists the experiments used in this thesis. The experimental setup and descriptions of the computational resources used in this thesis are presented in Chapter 5. Our results and discussion of the PLogP micro-benchmark and clustering benchmarks are presented in Chapter 6. Chapter 7 contains our case study of a topology-aware MPI broadcast. Here we introduce the reader to MPI collective operations, and various algorithms used for broadcast. We also present the implementation, experimental setup, and results from benchmarks of our parameterized broadcast operation. Finally, Chapter 8 concludes with both our main results from the PLogP micro-benchmark and clustering algorithms and the case study, and proposes interesting directions for future research.

Chapter 2

Background and related work

This chapter covers the background and related work. We organize the presentation into several sub-chapters. Sections 2.1 and 2.2 present the performance and communication models used in this thesis. Section 2.3 covers details of MPI. Section 2.4 presents important concepts of High-Performance Computing. Section 2.5 introduces clustering, and presents the methods used in this work. The chapter ends with Section 2.6 that presents the related work.

2.1 Performance models

2.1.1 Fundamental equation of modeling

The total time for parallel computation is modeled by the fundamental equation of modeling [3]. The model is based on the property that parallel programs can be reduced into two parts: computation and communication. It introduces the overlap of computation and communication. By overlapping computation and communication the total time will be reduced according to the overlap. Equation 2.1 shows the fundamental equation.

$$T_{\text{total}} = T_{\text{comp}} + T_{\text{comm}} - T_{\text{overlap}} \tag{2.1}$$

2.1.2 Scaling

In this thesis, we define scaling as the capability a program has to utilize computational resources as a function of the computational resources. Understanding how a program scales as computational resources increase is useful to understand how the program will perform for new systems. For our purposes, the number of CPUs is the measure of computational resources.

2.1.3 Speedup

Speedup is a measure used to compare the performance of two program implementations. The speedup for a program a when compared to another program b is given by Equation 2.2.

$$\text{speedup} = \frac{T_a}{T_b} \tag{2.2}$$

2.2 Communication models

2.2.1 Hockney

The Hockney model estimates the time required to send a message of size m between two nodes. It was first described by R. Hockney [4] to compare communication performance between two types of computers. The model consists of two parameters α and β. α is the latency, and β is the bandwidth of the link between the two nodes. The model is defined by Equation 2.3.

$$t(m) = \alpha + \beta^{-1}m \tag{2.3}$$

The values α and β are found by repeatedly measuring the time required to send messages between the two nodes for selected message sizes, followed by using linear regression to find the slope and y-intercept.

2.2.2 Bulk Synchronous Parallel

The Bulk Synchronous Parallel (BSP) model is a computational model designed to merge the bridge between hardware and software [5]. To achieve this, BSP models a computer by its computational units, the network that connects the computational units, and the hardware that makes the communication between components possible. The computational units in the BSP computer are able to do local computation and local memory operations. In modern computation systems, these computational units are the CPUs and threads of the system with their own local memory. The BSP model describes parallel program execution as a series of global *supersteps*. Each *superstep* consists of the following three components:

- Concurrent computation: local computation and memory operations within each component
- Communication: where memory operations between components are possible
- Synchronization: a barrier where all processes in a group must arrive until any process can proceed

Note that the synchronization step in the BSP model can be implemented using the broadcast operation.

2.2.3 LogP model

The LogP model, as described in [6] by Culler *et al.*, is a communication model that includes parameters for parallel systems. It estimates the communication time by four parameters: the latency L, the overhead o, the gap between messages g, and the number of processors P.

- The latency L is defined as the time it takes a message from the first byte is sent from the sender until the first byte is received by the receiver. Because the LogP model assumes independently executing processors, the parameter L is the upper bound on the latency.
- The overhead o is a measure of the time the CPU has to spend on sending and receiving a message.
- The gap g is the lower bound on the time interval between two consecutive messages. This means that two messages that are sent directly after each other will at least have a gap of g between them.
- P is the number of processors in the system.

A finite capacity for communication is assumed, so no more than $\frac{L}{g}$ messages can be sent at any time.

2.2.4 LogGP and Parameterized LogP (PLogP)

LogGP is a variant of the LogP model with an additional parameter G. G is the gap between messages for large messages.

The parameterized LogP model extends the LogP model by making the parameters depend on message size. PLogP also differentiates between sending overhead $o_s(m)$ and receiving overhead $o_r(m)$, with the relation $o(m) = \frac{o_r(m)+o_s(m)}{2}$. PLogP is useful when we want to study performance for various message sizes and with finer granularity than the standard LogP model.

Table 2.1 shows the relationship between the parameters of the LogP model and the PLogP model. The main difference is that all LogP/LogGP parameters are expressed through more parameters than the LogP model.

Figure 2.1 shows the message transmission diagram for LogP and PLogP as it is presented by Kielmann *et al.* in their paper on fast measurements of LogP parameters. It shows what the PLogP parameters measure, and how the measurement compares to that of the LogP model. In the diagram, the terms *measure* and *mirror* are used in the PLogP model, because that is what the original paper by Kielmann *et al.* [1] used. In this paper the terms *sender* and *receiver* are used instead.

2.3 Message Passing Interface (MPI)

Message Passing Interface (MPI) is a standardized API for high-performance parallel programs [7]. MPI is commonly used for parallel computations on large clusters, such as

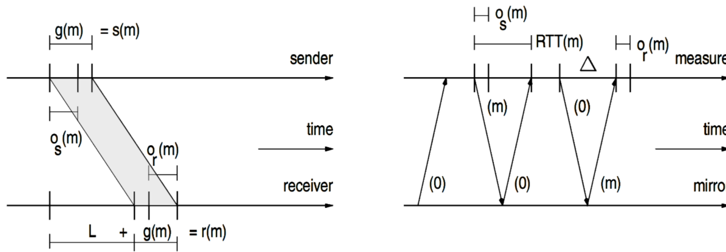


Figure 2.1: LogP (left) vs PLogP (right) message transmission diagram [1]

LogP/LogGP	PLogP
L	$L + g(1) - o_s(1) - o_r(1)$
o	$(o_s(1) + o_r(1))/2$
g	$g(1)$
G	$g(m)/m$
P	P

Table 2.1: Relationship between PLogP and LogP/LogGP models

supercomputers. It has support for both point-to-point and collective operations. The first version of MPI only had support for distributed memory, but later versions have support for shared memory [8].

2.3.1 Point-to-point communication

In MPI, processes can communicate with specific other processes by using point-to-point communication routines. Support for several communication modes is available. Normal, synchronized, and non-blocking point-to-point communication are some of these modes. In normal mode, the call will wait until the buffer is available to be reused. Synchronized mode forces a send call to wait until the corresponding receive call has started, but does not wait until it has completed. Non-blocking mode returns immediately but might send at a later time. It allows for overlap of communication and computation as modeled by Equation 2.1. For non-blocking mode the user has to check if the send or receive has been completed manually. Table 2.2 shows a subset of the point-to-point communication operations available in MPI. Two processes must both start their end of the operation for them to be able to communicate.

Operation	Synchronous	Blocking	Non-blocking
Send	MPI_SSend	MPI_Send	MPI_ISend
Recieve		MPI_Recv	MPI_Irecv
SendRecv		MPI_Sendrecv	MPI_Isendrecv

Table 2.2: Point-to-point based communication functions in MPI

2.3.2 Communication protocols

Most MPI implementations support two communication protocols: the eager, and the rendezvous protocol. The eager protocol allows for small messages to be sent directly to another process without synchronization [9]. It works by each process reserving memory for incoming eager messages. In the rendezvous protocol, each send requires to first exchange information like message size before starting the actual send. Therefore, an overhead of an extra round-trip is involved when switching to the rendezvous mode. Each implementation has a different message size threshold for switching from the eager protocol to the rendezvous protocol, which can lead to different performance for some message sizes.

2.3.3 Process rank and communication groups

Processes in MPI are identified by a numerical id known as the *rank*. The default communication group `MPI_COMM_WORLD` is created by MPI on program initialization, and contains all processes. Each process has a unique rank in a given communication group in the range $0 \ldots P-1$ that can be retrieved by using the function `MPI_Comm_rank`. Subcommunicators are communicators that only contain a subset of `MPI_COMM_WORLD`, and are used for communication on only a subset of all processes.

2.3.4 Collective communication

Collective operations in MPI are operations that work on communication groups. They require that all ranks in a communication group call the same operation, and those used in this thesis will block until all processes in the group are ready. Table 2.3 shows a list of some commonly used collective operations in MPI.

Several collective operations work with a *root* rank. The rank that is designated as root will do extra work such as coordination. For single-to-many or many-to-single operations the root rank will be the source and receiver of the data distributed to all other ranks.

MPI Function	Description
MPI_Barrier	Synchronization point
MPI_Broadcast	One-to-many send
MPI_Gather	Collect data from all ranks to root
MPI_Scatter	Root sends equal splits of data to all ranks
MPI_Allgather	All ranks collect data from all other ranks
MPI_Reduce	Reduce operation on data from all ranks

Table 2.3: Some collective communication functions available in MPI

We only go into further details about the Barrier and Broadcast operations, because they are most relevant for this thesis.

Barrier

The barrier operation is the simplest collective operation in MPI. It makes all processes in the communication group synchronize with each other. No processes are allowed to pass

the barrier until all processes in the group are synchronized.

Broadcast

The broadcast is a collective operation used to send data from a single root rank to all other ranks in the communication group.

MPI Process Affinity

Process affinity refers to the binding of a process to a specific processing unit or core. For MPI, the process affinity defines which cores should be mapped to which rank. One way to achieve this is to bind cores by a shared resource or property, such as l2cache, socket, or board. The process affinity is important for performance because different groups of cores have different shared resources and properties [10]. The `--map-by core` option to OpenMPI can be used to map all cores to ranks sequentially.

2.3.5 MPI Implementations

Intel MPI is a highly optimized version of the MPI specification for Intel processors [11]. The implementation is not open-source. OpenMPI is an open-source implementation that provides high performance on both homogenous and heterogenous systems, and with support for shared memory, Infiniband, and other communication protocols [12] [13]. MPICH2 is another implementation of MPI and was created in an attempt to provide an open-source and free, high-performance, and portable implementation of the MPI standard [14].

2.4 High Performance Computing

2.4.1 Basic Linear Algebra Subprograms (BLAS)

BLAS is an interface to commonly used fundamental linear algebra functions [15]. Functionality includes functions for vector operations like dot product and sum, and matrix operations like multiplication on both dense and sparse matrices. Highly optimized implementations of the BLAS interface include ATLAS [16], OpenBLAS [17], and Intel MLK [18].

2.4.2 Nodes, clusters and Network topology

Nodes are the entities that participate in a network and can be computers, servers, and other network devices. In the context of High-Performance Computing, a node is a self-contained system with CPUs and memory shared among these CPU cores. A cluster is a collection of many nodes that are interconnected in a network, such that each node can communicate with any other node in the cluster. Interconnect technologies include Ethernet and InfiniBand. InfiniBand is a high-speed interconnect technology that provides high bandwidth and low latency communication. Large HPC clusters contain 1000s of nodes with a total of tens of thousands of cores.

The network topology is defined by the way these nodes are interconnected. There are many different ways to interconnect nodes in clusters. The following are some commonly used network topologies in HPC clusters:

- **Fat-tree topology**: A fat-tree topology is a hierarchical network interconnect that is commonly used in large-scale HPC clusters. It consists of a series of layers, with each layer containing switches and routers that connect to the layer above and below it. This topology is highly scalable and can support a large number of nodes, but it can be expensive to implement. Figure 2.2 shows a fat tree with two layers, with a total of four nodes.

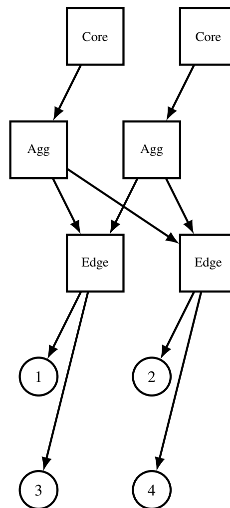


Figure 2.2: Fat-Tree topology with two layers and two nodes for each stem.

- **Star topology**: A star topology consists of a single switch connecting all network devices. Every device is connected to the switch with its own cable, and all communication goes from one node, through the switch, and out to the receiving node. This topology makes it easy to add or remove devices from the network and isolate problems to specific devices. When visualized with the switch in the center we get a star shape, as seen in Figure 2.3.

2.4.3 Topology aware communication

Information about the network topology can be to improve the performance of programs on clusters. Operations that utilize the network topology in communication are called *topology aware*. Topology-aware operations can achieve higher performance because they are able to better utilize system resources. Communication time is often a significant part of the total time in HPC applications for distributed systems, and a reduction in time spent on communication leads to higher performance of the application. Topology-aware communication is communication that utilizes the topology of the system it runs on to improve performance.

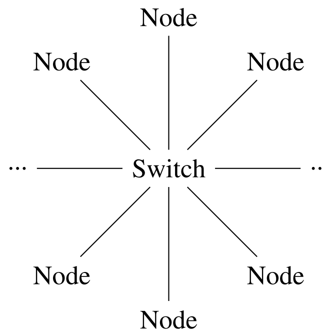


Figure 2.3: Star topology

2.5 Clustering methods

2.5.1 Metric spaces

A metric space $(\mathbb{X}, d)$ consists of objects $\mathbb{X}$ and a distance function $d : \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{R}^+$. The distance function d has to satisfy the following properties in metric spaces:

- **non-negativity**: $d(x, y) > 0 \; \forall (x, y) \in \mathbb{X} \times \mathbb{X}$, and $x = y$ if $d(x, y) = 0$
- **symmetry**: $d(x, y) = d(y, x)$
- **triangle equality**: $d(x, z) \leq d(x, y) + d(y, z)$

2.5.2 Vector spaces

A vector space $(\mathbb{V}, d)$ is a metric space where all the objects in $\mathbb{V}$ are k-dimensional vectors of real numbers. Distance functions for metric spaces include the Euclidian distance and the Manhattan distance.

2.5.3 Sparse Spatial Selection (SSS)

Sparse Spatial Selection is a technique used to dynamically select pivots when clustering [19]. To use SSS as a clustering technique, we start by choosing an object as the first cluster center. We iterate over every object and have a choice between adding it to the closest existing cluster c_{closest}, or creating a new cluster. c_{closest} is the cluster where the distance between the object and the cluster center is minimal. We let the maximum distance between objects in a cluster be M, and let α be a constant parameter. If the distance between the object and the cluster c_{closest} is greater than $M\alpha$, we create a new cluster with the object as the center. Otherwise, we add it to the closest existing cluster.

2.5.4 K-means

The K-means clustering algorithm [20] is one of the most commonly used clustering algorithms in research. The algorithm works on vector spaces of size n, and creates a partition

of the vector space into k sets, $k \leq n$, hence the name k-means. K-means is a static clustering algorithm because it takes the number of clusters as part of the input. The goal of K-means is to minimize the variance within each cluster. The centroid of a cluster is the mean value of the points in the cluster and does not need to be an object in the vector space. To find clusters, the K-means algorithm first selects k objects and makes each of them a new cluster.

Although the algorithm has many applications in research and data mining, it has many known limitations. Problems include the initial selection of centroids that lead to unexpected convergence, and the handling of various data types [21]. The optimal solution to the K-means problem is NP-hard, and therefore approximations are used. To mitigate the problems with convergence, many uses of K-means include running the algorithm for many iterations with different initial centroids, and using the best result. The big-O runtime complexity of K-means is $O(n^2)$, but heuristics have been used to achieve linear runtime complexity for a fixed number of dimensions and iterations.

2.5.5 K-medoids

K-medoids is a static clustering algorithm that allows for using non-Euclidean data and arbitrary distance functions. The algorithm takes a dissimilarity matrix and the number of clusters to find k. The algorithm returns k clusters where the total dissimilarity in each cluster is minimized. The K-medoids algorithm is less sensitive to outliers than other traditional clustering algorithms like K-means, because all medoid centers are data points. It has been shown that the general K-medoids problem is NP-hard [22]. Therefore approximation algorithms like Partitioning Around Medoids (PAM), or FasterPAM [23] are used. FasterPAM implementations have a runtime of $O(n^2)$ [24]. Another algorithm, BanditPAM [25], is able to reduce the runtime complexity to $O(n \log n)$, but at a cost of a high constant factor, making it suitable only for problems with $N \geq 10^5$ [24]. These approximation algorithms make it feasible to use K-medoids in this paper despite being NP-hard in the general case.

2.5.6 Spectral clustering methods

Spectral clustering is a group of clustering algorithms that use eigenvalue decomposition as a step in the clustering. Experimentally they have been shown to result in high-quality clusterings, but at the cost of high time complexity [26]. These algorithms apply eigenvalue decomposition on the distance matrix as part of the algorithms. Using the top d eigenvectors on a distance matrix is one of the techniques spectral clustering algorithms use eigenvalue decomposition.

2.5.7 Hierarchical clustering

Hierarchical clustering algorithms are clustering algorithms that create a hierarchy of clusters. This means that we get a tree of clusters, where each internal node is a cluster, and leaves are objects. The result of hierarchical clustering is presented in a dendrogram, which is a tree visualizing the relationship between clusters and sub-clusters. A cut in the dendrogram at level ε is used to get a flat clustering [27]. By changing the value of ε

we can choose the number of clusters we want. Figure 2.4 shows a dendrogram with two large clusters. These clusters have a significant distance between each other and smaller sub-clusters within them. A cut made at distance $0.8e^{-6}$ would result in the two clusters seen in yellow, while a cut at $0.65e^{-6}$ would give four clusters.

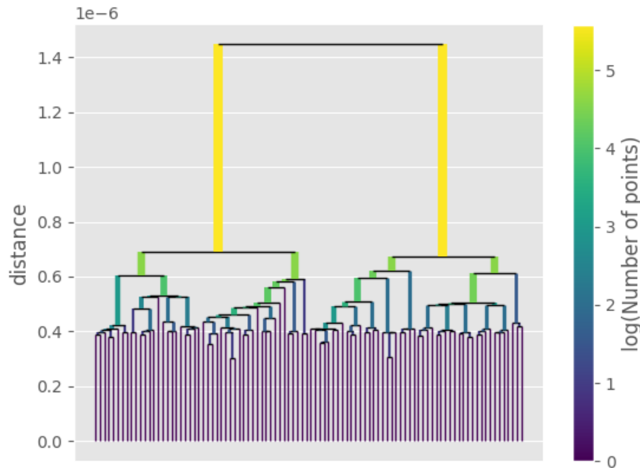


Figure 2.4: Example dendrogram from network measurements on ARM1

2.5.8 SSSTree

SSSTree [19] is a clustering algorithm for metric spaces. It uses SSS (see Subsection 2.5.3) for dynamically selecting cluster centers. By using sparse spatial selection clustering it will automatically determine the number of clusters based on the complexity of the data. SSSTree is a dynamic clustering algorithm, because it determines the number of clusters based on the data, and not based on some static input. The SSSTree algorithm builds a hierarchy of clusters using a top-down approach.

Tree construction

Let $\mathbb{X}$ be the collection of objects we want to cluster, and d be the distance function. We first start by creating what we call a *base tree*. The *base tree* is constructed by creating a single cluster and adding all objects as children of the cluster. The first child of a cluster is used as the center of the cluster.

We construct the SSSTree by recursively applying the algorithm seen in Figure 2.5. The algorithm uses the *radius* of a cluster. The radius is the maximum distance between the center of the cluster and other objects in the cluster. We use $M = 2 \cdot \text{radius}$ as an approximation of the maximum distance between two objects in the cluster. This works

because the upper bound on the distance between two objects inside the cluster is $2 \cdot$radius. This approximation reduces the runtime of this step from $O(n^2)$ to $O(n)$.

```
def construct_tree(node, parent):
    if parent: node.parent = parent
    if (not node.is_cluster() or
        len(node.children) < MIN_CHILDREN):
        return node

    node.radius = calculate_radius(node)
    m = 2*node.radius
    root = node.children[0]
    clusters = [root]

    for child in node.children[1:]:
        c, dist = find_closest_cluster(child, clusters)
        if dist >= m*ALPHA:
            clusters.append(new_cluster_with_child(child))
        else:
            add_to_cluster(c, child)

    for i, c in enumerate(clusters):
        clusters[i] = construct_tree(c, node)
    node.children = clusters
    return node
```

Figure 2.5: Tree construction algorithm for SSSTree

2.5.9 DBSCAN

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [28] is a dynamic clustering algorithm that automatically finds clusters and cluster centers. It takes two parameters; the neighborhood distance ϵ and the minimum number of samples per cluster $\min_{\text{pts}}$. DBSCAN works by finding regions of high and low density in the dataset, grouping points in regions with high density that are separated by low-density regions into clusters. The algorithm also deals with outliers, and will return a list of points that was classified as “noise“. One of the advantages over traditional clustering algorithms like the K-Means algorithm is that DBSCAN is able to detect irregular shapes, varying clustering densities, and noise.

2.5.10 HDBSCAN

HDBSCAN [27] is a hierarchical clustering algorithm that expands on the DBSCAN algorithm. It incorporates hierarchical clustering to the DBSCAN algorithm and is therefore

not only able to find clusters similar to the original algorithm, but also returns a hierarchical representation of the clusters. It differs from the DBSCAN algorithm in the parameters that it takes. It takes two parameters `min_cluster_size` and `min_samples`. These are easier to determine in practice than the neighborhood distance ϵ and minimum samples $\min_{\text{pts}}$ used in DBSCAN, and therefore make HDBSCAN less sensitive to parameter choices. The hierarchical organization of clusters also makes it more applicable to hierarchical data, such as network distance matrices.

2.6 Related work

2.6.1 Measurements of the LogP parameters

Kielmann *et al.* wrote a paper on "Fast Measurements of LogP parameters" [1], that describes a faster technique for measuring pair-wise PLogP parameters. Previous methods for measuring LogP parameters used link-saturation which takes a significant amount of time. Their paper presents a new method that does not use link saturation and is able to run the pairwise measurement faster than previous methods. A limitation of their work is that it only measures parameters for a single pair.

2.6.2 Optimization of broadcast and other collective operations in MPI

In their paper "Performance of MPI broadcast applications" [29], Wadsworth and Chen explore and benchmark broadcast algorithms in MPICH2. They compare several algorithms for implementing MPI broadcast and show that it is possible to improve the performance of the broadcast operation in MPICH2 by careful selection of the algorithm.

The paper "MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems" by Kielmann *et al.* [30], describes techniques for improving MPI collective operations in wide-area networks. They implement optimized collective operation routines that minimize communication across slow links. The work was published in 2000, and the wide-area latency is 10 ms, which is significantly higher than is common for inter-node communication in modern high-performance computing clusters. In their paper, the authors state that MagPIe was able to outperform MPICH2 with a factor between 2x and 8x.

Gong *et al.* created network performance-aware MPI collective operations for use in the cloud [31]. They define a network performance metric that depends on the bandwidth and latency of links. Measurements of latency and bandwidth between pairs are then used to create distance matrices based on this metric. They use a greedy method to find a performance hierarchy in the network. Their method achieves 30% improvement for the broadcast operation.

Chapter 3

Methodology

This chapter presents the methodology used in this thesis. Subchapter 3.1 covers the PLogP micro-benchmark. Subchapter 3.2 goes into detail about how we use the PLogP micro-benchmark together with clustering methods to optimize network topologies. Subchapter 3.3 describes the automated parameter search used in the clustering step. Subchapter 3.4 shows the table of conducted experiments. Finally, Subchapter 3.5 describes how we assess clustering algorithm performance.

3.1 PLogP measurements using LGate

3.1.1 LGate

LGate is a topology-aware test harness for testing all pairs of ranks in MPI. It provides functionality for initialization, synchronization, and parallel execution of pairwise tests. The user implements the `test_pair(i,j)` function that is called by LGate to run a pairwise test between ranks i and j. By default, only one pairwise test will be run at a time in order to avoid interference. However, a network topology can be specified in order to allow for parallel pairwise tests. The topology file defines which groups of ranks can be tested simultaneously. Figure 3.4 shows a topology file that tells LGate to allow the four groups of ranks {0-7},{8-15},{16-23} and {24-31} to be tested at the same time. This type of configuration is valid when we know that these groups of ranks will communicate independently and without interference, *e.g.* if they are separate nodes with internal interconnects.

A custom `subset_barrier` function is used instead of the builtin `MPI_Barrier` function for synchronization provided by MPI. The subset barrier takes an array of ranks to be synchronized and uses an algorithm based on `MPI_ISend` and `MPI_Recv` as shown in Figure 3.1. This barrier implementation uses the dissemination barrier algorithm [32].

LGate first runs a pairwise test on all leaves in the topology tree simultaneously. It recursively joins leaves and runs pairwise tests on the combined node. Use of the custom `subset_barrier` function ensures that all ranks in the same sub-tree are synchronized

```
def subset_barrier(ranks):
    if my_rank not in ranks:
        return
    num_stages = math.ceil(math.log2(len(ranks)))
    pos = ranks.index(my_rank)
    for stage in range(num_stages):
        from_idx = (pos - (1<<stage) + len(ranks))
                       % len(ranks)
        to_idx = (pos + (1<<stage)) % len(ranks)
        src, dst = ranks[from_idx], ranks[to_idx]
        MPI_Isend(dst, &send_req)
        MPI_Recv(src)
        MPI_Wait(&send_req)
```

Figure 3.1: Subset barrier algorithm based on dissemination used in LGate

while allowing ranks in different sub-trees to be tested at the same time. The algorithm for running the tests can be seen in Figure 3.2 and Figure 3.3.

```
def join(node):
    if is_leaf(node.left) and is_leaf(node.right):
        join_leaves(node.left, node.right)
    else:
        join(node.left)
        join(node.right)

def join_leaves(left, right):
    all_ranks = left.ranks + right.ranks
    for (l, r) in zip(left.ranks, right.ranks):
        if my_rank in {l,r}:
            test_pair(l,r)
        subset_barrier(all_ranks)
    merge_tree_node(left, right)
```

Figure 3.3: Algorithm for joining leaves in the cluster tree

LGate uses a binary tree to store the topology, but we extend LGate to allow more than two children. The reason is that is gives us more flexibility for other uses of the topology file because we can define any number of independent groups without having to decide how they should be merged.

```
def test_subset(ranks):
    for (i, j) in zip(ranks, ranks[1:]):
        subset_barrier(ranks)
        if my_rank in {i,j}:
            test_pair(i,j)
        subset_barrier(ranks)

def run_tests(tree):
    for leaf in tree.leaves():
        test_subset(leaf.ranks)
    while len(tree.leaves()) > 1:
        join(tree)
```

Figure 3.2: Algorithm for running all pairwise test

```
cluster {
 cluster {
  cluster { group { 0-7 } }
  cluster { group { 8-15 } }
 }
 cluster {
  cluster { group { 16-23 } }
  cluster { group { 24-31 } }
 }
}
```

(a) LGate topology with 4 groups

```
root:
    sub-tree:
        leaf: ranks 0-7
        leaf: ranks 8-15
    sub-tree:
        leaf: ranks 16-23
        leaf: ranks 24-31
```

(b) Tree representation of topology

Figure 3.4: LGate topology file and tree representation

3.1.2 PLogP measurements using LGate

An implementation of a benchmark to estimate the values of the PLogP model was created using LGate. The implementation extends the work of Kielmann *et al.* [1] which describes a technique for fast measurement of LogP parameters between two nodes. Our implementation uses techniques from Kielmann's paper, but measures between all ranks. The primary goal of the benchmark is to be able to detect network topologies in large clusters $(P \geq 250)$ within a reasonable timeframe (in minutes). Several techniques to reduce variance in measurements are applied.

Our implementation finds LogP parameters between all pairs of ranks $\{(i, j) \mid i < j\}$. We call rank i the sender and j the receiver. LogP parameters are only measured one way, from the sender to the receiver, thus we implicitly assume symmetry. Allowing measurement both ways is trivial, but unnecessary because we want to analyze the results using methods that require symmetrical distance matrices (see Subsection 2.5.6). All measurements are made by sending messages between the sender and receiver, timing various

stages, and varying the number and size of messages. We have $\log_2 M$ measurement iterations, where M is the maximum message size to be measured. For each iteration $i = 0 \ldots \lfloor \log_2 M \rfloor$ we measure PLogP parameters for message size $m = 2^i$. The total number of measurements is $O(\log M \cdot r \cdot n^2)$, where r is the number of repetitions per measurement.

Measurement of $g(m)$

The measurement starts with measuring $RTT(0)$. This is done by sending n messages with $m = 0$ from the sender to the receiver. When the receiver has received all messages, it sends a single message back to the sender. $RTT(0)$ is then the total time taken divided by n. We have two methods for measuring $g(m)$. The first method is based on link saturation. Using this method we start by sending $n = 10$ messages in a row of size m and calculate the average $\overline{rtt'} = \sum_{i=0}^{n} rtt_i/n$. If the average RTT for the current n only differs by ε from the previously measured RTT, then we have reached saturation. Otherwise, we increase n and repeat until we reach saturation, or we exceed $n \geq 37500$. By default, ε is equal to 0.01.

The other method uses the method described by Kielmann *et al.* [1], where we only use link saturation for $g(0)$. When we measure $RTT(m)$, the timing diagram for PLogP (Figure 2.1) shows us that $RTT(m) = L + g(m) + g(0) + L$. We can rewrite this as $g(m) = 2 \cdot L + g(0) - RTT(m)$. Because $RTT(0) = 2 \cdot L + 2 \cdot g(0)$, we get Equation 3.1 for $g(m)$.

$$g(m) = RTT(m) - RTT(0) + g(0) \tag{3.1}$$

Instead of measuring $g(m)$ using link saturation, we can calculate it from $RTT(m)$. This has no additional cost because $RTT(m)$ is required to determine $o_s(m)$ and $o_r(m)$. This method significantly reduces the amount of network traffic required, and thus makes it feasible to run the micro-benchmark for large values of P ($P > 256$). We found experimentally that for values of $m \geq 2^{16}$ the method gives values of $g(m)$ that are close to that found using the link saturation method. Therefore, we use the saturation-based method for values $m \leq 2^{16}$, and the non-saturation method for larger values of m.

Measurement of L

When we measure $RTT(0)$ we are sending a zero-sized message, and receiving a zero-sized message. The sending part takes $L + g(0) + o_s(0) + o_r(0)$, but in the PLogP model this simplifies to $L + g(0)$. The receiving part takes the same time in our model. Therefore we get $RTT(0) = 2 \cdot (L + g(0))$. Solving this for L we get Equation 3.2, which we use to calculate the latency L.

$$L = \frac{RTT(0) - 2 \cdot g(0)}{2} \tag{3.2}$$

Measurement of $o_s(m)$

The PLogP model differentiates between the overhead for sending a message, and the overhead for receiving a message. To measure the send overhead for a given message size

```
def measure_latency(i,j):
    char buf[2];
    start = wtime()
    if my_rank == i:
        send(buf[0], 0, j)
        recv(buf[1], 0, j)
    elif my_rank == j:
        recv(buf[0], 0, i)
        send(buf[1], 0, i)
    end = wtime()
    return (end-start)/2.0
```

(a) Measuring latency

```
def measure_send_overhead
                (i,j,m):
  char buf[m];
  start = wtime()
  if my_rank == i:
    send(buf, m, j)
    send_time = wtime()
    recv(buf, m, j)
  elif my_rank == j:
    recv(buf, m, i)
    send(buf, m, i)
  end = wtime()
  rtt = end - start
  o_s = send_time - start
  return rtt, o_s
```

(b) Measure send overhead and rtt

Figure 3.5: LogP measurements

m, we do the following:

1. Synchronize the receiver and sender
2. Take the time needed to send a message of size m by measuring the wall time before and after a call to `MPI_Ssend`. Figure 3.5 (b) shows an implementation that also stores the RTT.
3. We repeat step 2 N times and store the average as o_s.

When measuring the send overhead we use `MPI_Ssend` instead of `MPI_Send`, because the MPI specification allows `MPI_Send` to buffer the send, but `MPI_Ssend` only returns after a matching `MPI_Recv` has been posted.

Measurement of $o_r(m)$

The receive overhead is the overhead of the `MPI_Recv` call. It is the time required for a `MPI_Recv` call when no time is spent waiting for the corresponding `MPI_Send`. We measure the receive-overhead for a given message size m with these steps.

1. Synchronize sender and receiver
2. Send a message from the sender to the receiver
3. The receiver immediately replies with the same message
4. We make the sender wait for $RTT(m) \cdot k$
5. The sender calls and times `MPI_Recv`. We call this the receive-overhead

6. We repeat steps 2-5 N times and store the average receive overhead as o_r

We use the expected round-trip time (RTT) as the time to wait between send and receive when calculating the receive-overhead. We can wait longer than the expected RTT if we want more precise measurements of $o_r(m)$. When a message round-trip takes longer than RTT, the `MPI_Recv` call will block while waiting for the message, and we will get a higher $o_r(m)$ value. If we wait for $k \cdot RTT$ time, we can find a minimal value for k such that only an acceptable amount of messages cause a block in `MPI_Recv`. Choosing a good value for k is important because waiting for more time than necessary is wasted time, but a k that is too low gives inaccurate measurements. We choose $k = 1.3$, so we will at worst wait 30% longer than necessary. To get an accurate timing we use a busy wait between the `MPI_Send` and `MPI_Recv` call. Figure 3.6 shows an example implementation.

Accounting for clock overhead

To account for the overhead of `MPI_Wtime()` we measure the expected time `MPI_Wtime()` takes each iteration. We use a `busy()` function as shown in Figure 3.7 that runs the increment operation 300 times in a loop. We use a variable with `volatile` to prevent compiler optimizations from removing the loop. The Godbolt Compiler explorer [33] was used to confirm that the compiler does not remove the loop for the compilers used in this thesis (Clang, GCC & ICC) using optimization level `-O2`.

We can now time 1000 calls to `busy()` and calculate the average time per function call as $\overline{t_b}$. In another loop, we have the same number of calls to `busy()`, but with each call surrounded by `MPI_Wtime()`. We store the start and end time per call to `busy()` for each iteration i as t_{s_i} and t_{e_i}. We can now calculate the average time for each call $\bar{t} = (\sum_i^N t_{e_i} - t_{s_i})/N$. We repeat this procedure 5 times and calculate the overhead from $\min_{i=0\ldots4} \overline{t(i)} - \min_{i=0\ldots4} t_b(i)$. The reason we chose 5 is that it is the lowest number that gives consistent results.

```
static void busy(void) {
  volatile int i, dummy;
  for (dummy=42, i = 0; i < 300; i++) dummy++;
}
```

Figure 3.7: busy() function used to measure clock overhead, same as used in [1]

The clock overhead is used to correct for calls to `MPI_Wtime()` inside measurements. We take the minimum average value from 5 repetitions because we want to be sure not to use a higher value than expected.

Additional measurements

Inspired by Kielmann *et. al* [1], we check the results for all measured message sizes. Let m_i be the message size for iteration i. We check that the measured value at iteration

```
def synchronize(i,j):
    subset_barrier([i,j])

def measure_send(i,j,N,msg, m):
    Output: rtt, o_s

    buffer: m bytes
    rtt_sum, o_s_sum = 0,0
    for j in range(N):
        start = MPI_Wtime()
        if my_rank == i:
            MPI_Send(msg, m, j)
            send_time = MPI_Wtime() - start
            MPI_Recv(msg, m, i)
        else:
            MPI_Recv(msg, m, j)
            MPI_Send(msg, m, i)
        rtt_sum += MPI_Wtime() - start
        o_s_sum += send_time

     o_s = o_s_sum/N
     rtt = rtt_sum/N

def measure_recv(i, j, N, msg, m, rtt):
    Output: o_r

    o_r_sum = 0
    for j in range(N):
        if my_rank == j:
            MPI_Recv(msg, m, i)
            MPI_Send(msg, m, i)
        else:
            send_time = MPI_Wtime()
            MPI_Send(msg, m, j)
            spin_wait_until(send_time + rtt)
            recv_start = MPI_Wtime()
            MPI_Recv(msg, m, j)
            o_r_sum += MPI_Wtime() - recv_start

    o_r = o_r_sum/N
```

Figure 3.6: Algorithm for joining leaves in the cluster tree

i is within ε of what we expect from a linear fit from the measured value at i_{i-1} and i_{i+1}. If the difference is greater than ε, then we first make an additional measurement of the midpoint between m_a m_i and m_{i+1}. This repeats until there are no more additional measurements required, or when the difference in the number of bytes is less than or equal to 32: $m_a - m_1 \leq 32$.

This means that we can choose to use fewer measurements, and only use additional measurements if needed.

3.2 Optimizing network topologies

The PLogP parameters we obtain by running the LGate microbenchmark can be used to model the communication performance between ranks in the network. We use clustering algorithms to find groups of ranks that have low RTT communication together. We want to find the grouping such that we minimize RTT between members in a group. By finding the pairwise distance, we can generate a distance matrix. Clustering methods can be used to find an approximate solution to this problem.

We use two methods to determine how two ranks communicate. The first method uses linear regression to find the slope $p(m) = am + b$ for each parameter for every pair of ranks (i, j) in the network. The other method is to use a simple linear regression between consecutively measured points. This can give more accurate results for the PLogP micro-benchmark because it takes additional measurements to ensure that a simple linear regression between consecutive measurements is within ε of the real measurement. We calculate the distance between ranks (i, j) for a given m. Equation 3.3 predicts the time required to send or receive a message m between ranks (i, j).

$$t(m) = p_L(m) + \frac{p_{o_s}(m) + p_{o_r}(m)}{2} + p_g(m) \tag{3.3}$$

For both methods, we create a distance matrix $D(m)$ from the prediction $t(m)$, as seen in Equation 3.4.

$$D(m)_{ij} = t_{ij}(m) \tag{3.4}$$

Several algorithms take a list of n-dimensional data points. To transform our PLogP data into this form, we let the parameters for some message size m; L, o_s, o_r, and g define a four-element vector v_{ij}. For each rank $i = 1 \ldots P$ we create a vector V_i that contains all elements v_{ij} for every $j = 1 \ldots P$. This gives a $4 \cdot P$ element vector for every rank as shown in 3.5

$$V_i = [v_{i0} \ldots v_{i1} \ldots \cdots v_{ij} \ldots] \tag{3.5}$$

With V we can create a similarity matrix $A(m)$ by calculating the Euclidean distance, shown in Equation 3.6, between all pairs of vectors V_i.

$$d(x, y) = \|x - y\| \tag{3.6}$$

Using ”fast” measurements

The PLogP micro-benchmark has functionality for enabling ”fast” mode, where we only check a certain percentage f_p of ranks within each group as defined in the topology file. With this mode, we choose a random selection of ranks per group and disregard the rest. Because the number of pairwise tests scales quadratically, we can run our benchmark for a higher node count than otherwise.

Measure of accuracy

To be able to compare different clustering algorithms, we need a way to measure the quality of results. There are often several valid results because there might be several ways to group ranks in a way that describes the network. For example, we might have two nodes each with two sockets. The following are all possible valid groupings of ranks: one cluster with all ranks, two clusters grouped by nodes, four clusters with one group per socket, and one group per rank. The detail required to be able to detect topologies vary. No detail is needed to detect one cluster, a few measurements might be able to detect the topology on a node level, but much more detail is needed to identify topology on the socket level. We can improve the granularity by making more measurements at a cost of more time.

Given the specification of each system, we can compare the result of the clustering to what we expect based on knowledge about the system. The PLogP micro-benchmark saves the relationship from rank to node, and node to machine in its output. The program `lstopo` from the hwtopo module in OpenMPI is used to collect information about the hardware on each node, such as the number of sockets and the assignment of cores to sockets. We define three measures of accuracy:

- Machine level accuracy: ratio of ranks that are mapped to the correct machine
- Node level accuracy: ratio of ranks that are mapped to the correct node
- Socket level accuracy: ratio of ranks that are mapped to the correct socket

$$\text{accuracy}_{\text{machine}} = \frac{|\{\text{ranks mapped to correct machine}\}|}{|\{\text{ranks}\}|} \tag{3.7}$$

$$\text{accuracy}_{\text{node}} = \frac{|\{\text{ranks mapped to correct node}\}|}{|\{\text{ranks}\}|} \tag{3.8}$$

$$\text{accuracy}_{\text{socket}} = \frac{|\{\text{ranks mapped to correct socket}|\}}{|\{\text{ranks}\}|} \tag{3.9}$$

Each clustering method is run several times, using different parameters for each data set. This is required because each clustering method has different parameters, and the optimal parameters depend on the data. We run each algorithm with several parameters and let the score be the maximum value among them to account for this. We parameterize

the accuracy and score with the method m, and enumerate the accuracy for each run by i. Equation 3.10 shows the function used to calculate the score.

$$\text{score}_{\text{metric}}(m) = \max_{i=0\ldots n} \text{accuracy}_{\text{metric}}(m) \tag{3.10}$$

For static methods such as k-medoids, we use the known number of machines, nodes, and sockets as the number of clusters. For non-static methods such as SSSTree, we run it using a variety of parameters that influence the number of clusters. For SSSTree this means running the algorithm for several values of α. The static methods consistently get a higher score, but this is heavily impacted by the fact that they are given the number of clusters as input. The dynamic methods have no knowledge of how the clustering should be and are therefore prone to getting different answers.

3.3 Automated clustering parameter search

The results of clustering algorithms depend on the parameters chosen. For many algorithms, we have a single parameter, like α for SSSTree, and *threshold* for connected-components clustering. Let S_x be a parameter space for the parameter x, defined by $S_x = (\text{low}, \text{max})$. Our parameter search takes an evaluation function $f(x) : \mathbb{R} \to \mathbb{R}$, that maps the parameter value x to a real number.

We use a basic linear search which takes the number of points to examine k. The linear search searches from low to high in increments of $\frac{\text{high}-\text{low}}{k}$. This method is useful for static methods like K-medoids and Spectral-clustering because a linear search in the range $[1, P)$ with increments of 1 is an exhaustive search. It can also be applied to algorithms with non-integral parameters like SSSTree and DBSCAN. For SSSTree the parameter space $(0.0, 5.0)$ is used with non-integral increments. For DBSCAN the parameter space $(\min A, \max A)$ is used, where A is the distance matrix.

We use $k = 50$ for our experiments. The value was chosen because it was the minimal value that got useful results for $P = 32$. This means that all clustering algorithms are called 50 times.

3.4 Experiments

Table 3.1 shows the conducted experiments. For each experiment, the PLogP microbenchmark is run with several values of R. The automated clustering parameter search is then used for several clustering algorithms. The clustering accuracy measure is then used to compare the results for different systems, algorithms, and values of R.

Every experiment has a specific purpose. First of all, we want to understand what the limitations of our method are. Secondly, we want to understand to what level we can extract information about the topology. We want to know if it is possible to find clusters with distinct characteristics at the CPU level, Socket level, Node level, Machine level, and Cluster level. Thirdly, we use the method on an unknown system and find out if we can extract useful information about it. Finally, we want to understand the performance of our method on large node counts and $P \geq 512$.

System	Nodes/P	Purpose
FRAM	1/32	Find node internal topology
FRAM	2/32	Find inter-node topology
FRAM	4/32	Find cluster topology
FRAM	8/32	Understand performance characteristics for large P
FRAM	8/1	Find cluster topology for larger node count
FRAM	16/1	Find cluster topology for larger node count
FRAM	32/1	Find cluster topology for larger node count
ARM	1/96	Discover the currently unknown topology of a research system
BETZY	4/128	Find topology for BETZY
HPCLAB	1/4	Baseline

Table 3.1: Table of experiments

3.5 Clustering algorithm performance

For every experiment, the automated clustering parameter search method is used. We can then compare the accuracy of clustering algorithms under certain conditions. By running the algorithms under different conditions we are able to get an understanding of the relative performance characteristics. We test all clustering algorithms on all experimental measurements and compare them.

Chapter 4

Experimental setup

4.1 Compilation

All benchmark code is written in C using MPI. These benchmarks are compiled using `mpicc` with `-O2` optimization level, using `-std=c99`. The C implementation of SSSTree is compiled using `gcc` with `-std=c99`. It uses the GSL and BLAS libraries which are linked with the linker flags `-lgsl -lcblas`. The BLAS implementation used is either Intel MKL or OpenBlas.

4.2 SLURM and MPI

The `sbatch` program from SLURM is used to start jobs on clusters like IDUN, FRAM, and BETZY. For each job, a job description file is generated that contains information about the job, such as node count and cores per node. A configuration file for the PLogP micro-benchmark is also generated for each job. `mpirun` is used as a process launcher for OpenMPI and `srun` is used for IMPI. All experiments use the default rank assignment if not specified otherwise. Experiments on the clusters IDUN, FRAM, and BETZY use IMPI. Experiments on ARM1 use OpenMPI, and HPCLAB uses MPICH.

4.3 Analysis and processing

While experiments are run on clusters, the analysis is done locally on the personal computer HPCLAB. Several software packages are used to process, analyze and present the results of the experiments.

4.4 `HPC Lab 06 computer - HPCLAB`

We use SPACK [34] to build all software except for MPICH from the source code.

Name	Version
Python3	3.10.6
SKLEARN	1.2.2
NumPy	1.21.5
Matplotlib	3.7.0
Seaborn	0.12.2
Scipy	1.10.1
Networkx	3.1
Pandas	1.5.3
HDBScan	master (as of May 23)
KMedoids	master (as of May 23)
graph_based_clustering	0.1.0

Figure 4.1: Python packages and versions

Name	Version
Spack	0.20.0.dev0
GCC	12.2.0
MPICH	4.0

(a) Software versions

Processor	Intel Core i7-6700K CPU @ 4.0 0GHz
Cores	4
Sockets	1
Memory	15 GiB

(b) Hardware

Figure 4.2: HPCLAB system properties

4.5 IDUN

IDUN is a computing cluster at NTNU [35] that is used for prototyping and rapid testing of HPC applications. The cluster consists of different types of machines. In order to achieve consistent results, we choose to use only selected nodes that have the same hardware on IDUN.

Name	Version
GCC	11.3.0
OpenMPI	4.1.4
IMPI	2021.7

(a) Software versions

Nodelist	idun-07-[01-32]
Processor	Intel Xeon Gold 6348
Interconnect	Star topology [36]
Cores	16 per socket
Sockets	2

(b) Hardware

Figure 4.3: IDUN system properties

4.6 FRAM

FRAM is a cluster for research computing hosted at The Arctic University Of Norway (UiT), and is provided by Sigma2 [37]. It has 1004 nodes in total with 32 cores each.

The nodes are interconnected with an Island topology. Figure 4.4 shows relevant hardware specifications and software versions used.

Name	Version
GCC	11.3.0
OpenMPI	4.1.4
IMPI	2021.7

(a) Software versions

Interconnect	Island topology [37]
CPU Type	Intel E5-2683v4 2.1 GHz
Node count	1004
Cores	32 per node
Sockets	2 per node
Memory	64 GiB per node
NUMA nodes	1 per socket

(b) Hardware

Figure 4.4: FRAM system properties

4.7 BETZY

BETZY is the most powerful supercomputer in Norway [38], and is made available for use in this thesis through Sigma2. The supercomputer is installed at NTNU in Trondheim, where it entered production in 2020. Figure 4.5 shows the relevant software versions and shows the hardware specifications used on BETZY.

Name	Version
GCC	11.3.0
OpenMPI	4.1.4
IMPI	2021.7

(a) Software versions

Interconnect	InfiniBand HDR 100, Dragonfly+ topology [38]
CPU Type	AMD® Epyc™ 7742 2.25GHz
Node count	1344
Cores	128 per node
Sockets	2
Memory	256 GiB per node

(b) Hardware

Figure 4.5: BETZY system properties

4.8 IDUN-ARM1

IDUN-ARM1 is a single-node research ARM64 system with 96 cores with a total of 254GiB of RAM available. Figure 4.6 shows the relevant software and hardware.

Name	Version
GCC	9.2.0
OpenBLAS	0.3.6
OpenMPI	4.1.4

(a) Software versions

CPU Type	ARM aarch64
Cores	96
Sockets	2
Memory	254 GiB
Numa nodes	4 total

(b) Hardware

Figure 4.6: ARM1 system properties

Chapter 5

Results and discussion

In this chapter, we present results from experiments on the systems FRAM, BETZY, and ARM1. The chapter is organized by the experiments in Table 3.1. Section 6.1-6.4 discusses finding topologies for a single node, two nodes, and many nodes on FRAM. In Section 6.5 we discuss our findings on ARM1. Section 6.6 discusses finding topologies on BETZY with 4 nodes. Section 6.7 presents a baseline experiment on HPCLAB. Finally, Section 6.8 and 6.9 presents the scaling and performance of clustering algorithms.

5.1 Finding internal node topologies for a single node

We conduct several experiments on a single FRAM node. The main goal is to understand whether all cores have the same communication characteristics, or if there are certain cores that can communicate with lower latency or RTT than others. We experiment with varying numbers of measurement repetitions to understand how the detail changes when we increase measurement repetitions.

Figure 5.1 shows the distance matrix for a single node with all 32 cores for several values of R. Note that the only difference between the $R = 8$ and $R = 32$ distance matrix is that there are fewer points deviating from the mean. For $R = 8$ several outliers with higher predicted $RTT(0)$ are seen. Some outliers can be seen for $R = 32$, but fewer than for $R = 8$. As shown in Figure 5.2, the variance in the distance matrix for $R = 128$ is less than half of that of $R = 8$. This shows that more measurements will reduce the variance for measurements on a single node, and increase the detail in the distance matrix.

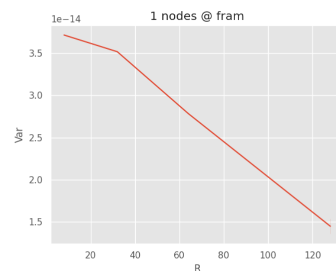


Figure 5.2: Variance for values of R

From the distance matrix, we can observe that there is an alternating pattern. The alternating pattern results from using round-robin rank assignment with the 2-socket configuration of FRAM nodes. Figure 5.3 shows the result

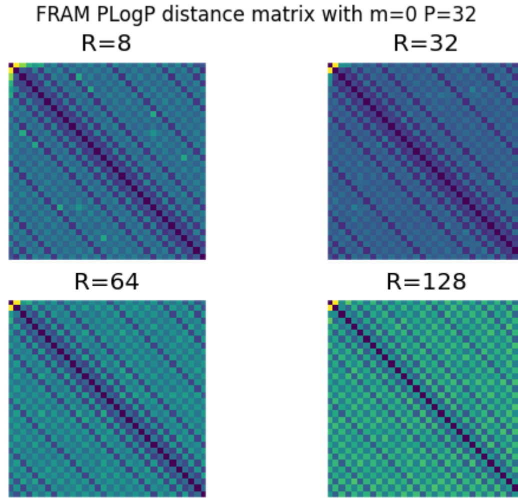


Figure 5.1: Communication time prediction for 1 node @ FRAM. The clear alternating pattern at R=64 and R=128 is caused by inter-socket communication.

when running with block distribution. Here the two groups are apparent, even though no other changes to the micro-benchmark were made. We can also see that this pattern increases in detail with increasing values of R. An automated clustering search reveals the following groups:
$\{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30\}$,
$\{1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31\}$. This is equivalent to the alternating pattern and shows the two-socket configuration.

We run the PLogP measurements for $R \in \{8, 32, 64, 128\}$. All clustering methods are able to find a clustering with 1.0 machine, node, and socket accuracy (see Figure 7.1). It is important to be aware that the PLogP micro-benchmark is run with the `additional_measurements` flag set. This causes it to take additional measurements when inconsistencies are detected, which occurs more often for lower values of R. This is the reason that the $R = 8$ gives results that match $R \geq 32$. The key takeaway from this is that it is possible to detect internal node topology in the form of socket-level topologies for a single node with $R \leq 64$.

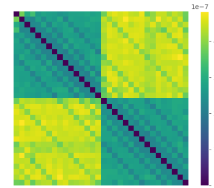
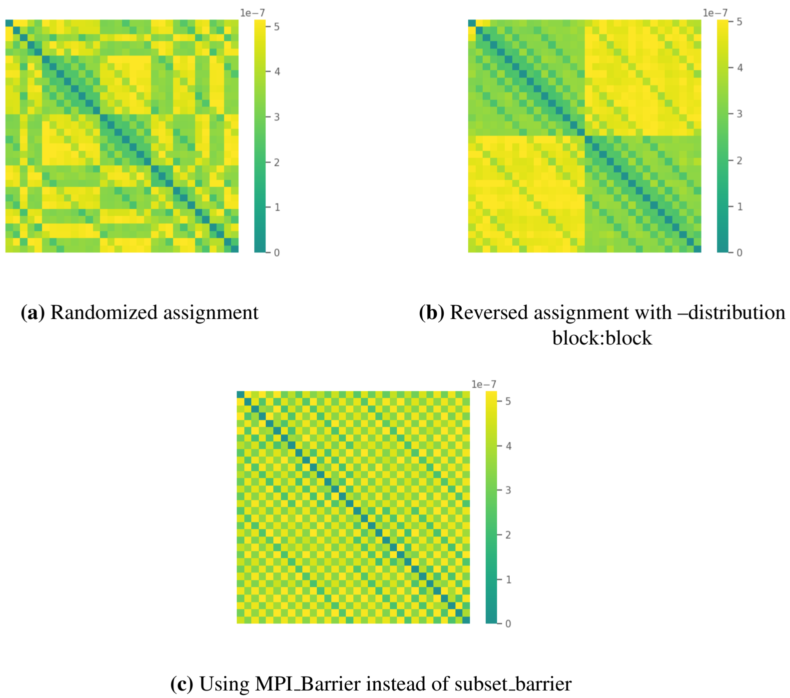


Figure 5.3: Communication time prediction for 1 node, $R = 256$ @ FRAM. Using –distribution block:block

5.1.1 Outliers for first two ranks

As seen in Figure 5.1, the measurements between the first two ranks (rank 0 and 1), as seen in the upper left corner,



(a) Randomized assignment

(b) Reversed assignment with –distribution block:block

(c) Using MPI_Barrier instead of subset_barrier

Figure 5.4: Two different rank assignments and alternative subset_barrier, m=0, P=32, R=128 @ FRAM

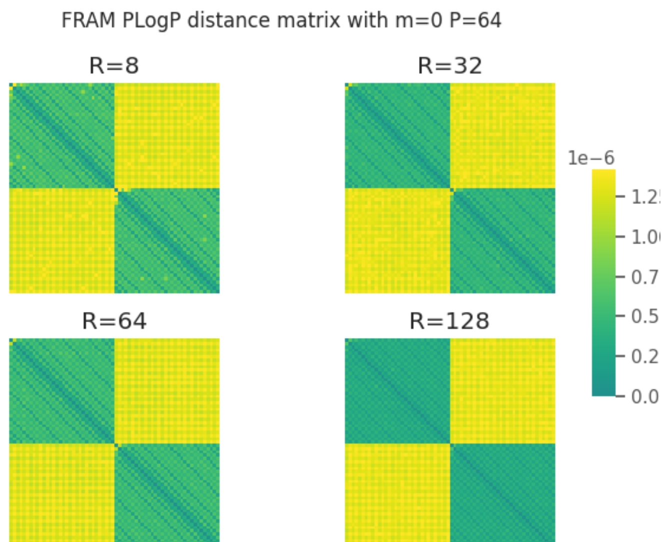


Figure 5.5: $RTT(0)$ prediction for 2 nodes @ FRAM. Here the same pattern as for 1 node appears, but the divide between the two nodes is also visible. Outliers $\geq$ 95th percentile are removed to maintain a reasonable color scale.

differ from the rest. A band of higher measurements is also visible from rank 0. If ranks 0 and 1 have unique properties, then we expect the reversed order to show the same pattern for ranks 30 and 31. As shown in Figure 5.4, the discrepancy appears in the upper left corner for both the randomized and reversed rank assignment experiments. This means that the measurements are not dependent on differences between cores, and are instead due to measurement artifacts.

5.1.2 Cause of diagonal bands

We can also see that the diagonal bands in the distance matrices are in the same place regardless of the rank assignments. If these bands were representing an inherent property of the system we would expect them to be different for each rank assignment. We observe that the diagonal bands occur at the same places regardless of the rank placement, and are therefore not due to inherent properties of the core. When the LGate `subset_barrier` function is replaced with one based on `MPI_Barrier` we get a different pattern of diagonal bands. Figure 5.4c) shows that the diagonal bands are discontinuous when we use `MPI_Barrier`. We conclude that these diagonal bands are measurement artifacts.

5.2 Finding inter-node topologies between two nodes

We conduct experiments using two FRAM nodes. All these experiments used a topology with one cluster of 32 ranks per node. Figure 5.5 shows the distance matrix for 2 nodes

on FRAM. There is a clear divide between the predicted inter-node and intra-node communication time, which is seen as the two groups in the distance matrix. The inter-node communication time is $\approx$ 2x the intra-node communication time. For intra-node communication, we can see the same pattern as for a single FRAM node, but with more noise for $R = 8$.

As shown in Figure 7.2, the clustering accuracy is 1.0 for all static clustering methods. The dynamic clustering methods HDBSCAN and Connected Components have a socket accuracy of 1.0 for all $R \geq 8$, while SSSTree requires $R = 128$ to get the same accuracy. An interesting result is that DBSCAN has very low scores for all values of R, while HDBSCAN has a top score. One reason for this might be the fact that DBSCAN is not a hierarchical clustering algorithm, and therefore might not find the clusters we expect to find. Another explanation is that DBSCAN is too sensitive to the selection of a parameter value, such that the parameter search never finds a suitable value. Spectral clustering also has a similarly low score for all values of R.

5.3 Finding node interconnection topologies

The PLogP parameters L, $o_s(m)$, $o_r(m)$ and $g(m)$ all model different properties of network communication. The latency can be used to extract information about how close two nodes are to each other in the network. The overhead of sending a message varies based on how MPI does the communication, which itself depends on the architecture. FRAM uses InfiniBand, which are low-latency, high-bandwidth interconnects. We look at how these parameters can be used to extract information about the topology.

Figure 5.6 shows the latency, where there are two levels of clusters. The first level is the nodes, and the second level is the machine. The PLogP micro-benchmark generates a metadata file with information about the mapping from rank to the hostname of the node. We group by machine on FRAM by using the hostname prefix. For $R = 8$ the ranks $32 - 95$ form a group that has lower latency than other inter-node pairs.

Here the nodes are identified by c48-8, c48-9, c49-2, and c49-3. They are assigned per our definition to two machines, but no latency variation between them can be observed.

These latency measurements show that intra-socket communication is ≈ 2 faster than inter-socket communication on FRAM nodes. They also show that due to the way the nodes are interconnected, some nodes communicate with ≈ 1.5 lower latency than others. This result means that the specific interconnection between nodes has a higher impact on the latency than sharing the same machine. This also explains why dynamic clustering algorithms achieve higher node accuracy than machine accuracy (Figure 7.3).

Figure 5.7 shows the latency for 8 nodes on FRAM. Here the 8 nodes are divided into two clusters that have lower communication latency.

5.4 Determining the cluster topology for a large number of nodes

Several experiments were conducted with high node counts on FRAM. These experiments use a small number of cores, ≤ 4 of cores per node, and instead scale up the number of

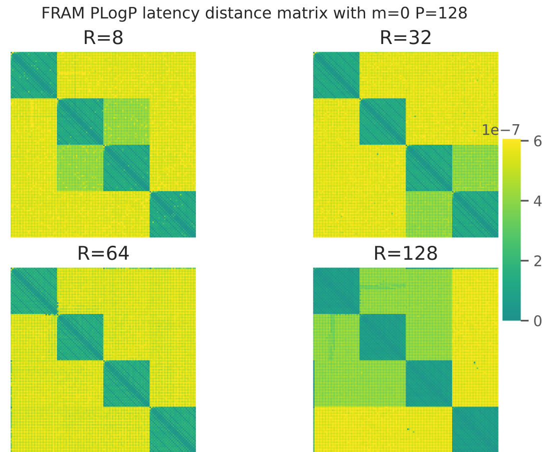
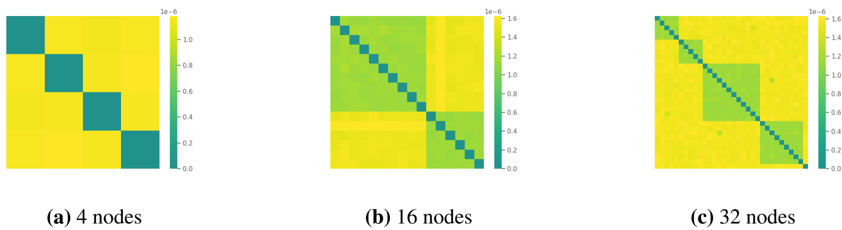


Figure 5.6: Latency for 4 nodes @ FRAM. Here intra-node latency is apparent. Differences between runs are due to different node assignments. Outliers $\geq$ 95th percentile are removed to maintain a reasonable color scale.

nodes. The goal is to see if it is possible to detect the topology caused by the interconnection of nodes.



(a) 4 nodes **(b)** 16 nodes **(c)** 32 nodes

Figure 5.9: RTT(0) with 1 core per node, R=64 @ FRAM

Figure 5.9 shows three experiments with 4, 16, and 32 nodes. We can see that for 16 and 32 nodes, there are visible groups of nodes that can communicate faster and slower. For 16 nodes we find two clusters $\{0-9\}$, $\{10-15\}$ and for 32 nodes a clustering search finds the following 5 clusters:

```
{1-4},{5-9},{10-21},{22-30},{31}
```

These match the areas in the heat plots with the lowest values for $RTT(0)$ between ranks. The latency within each detected cluster is $\approx 0.45\mu s$, while the latency between clusters is $\approx 0.65\mu s$. Therefore we have been able to automatically detect inter-node topology on the FRAM cluster.

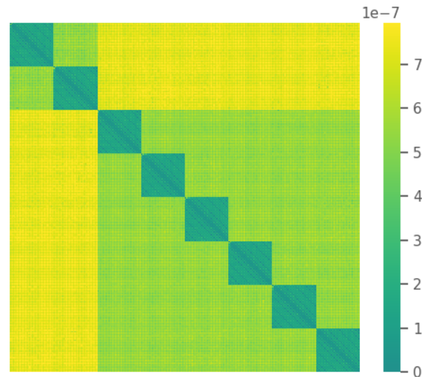


Figure 5.7: Latency for 8 nodes, R=64 @ FRAM. Here intra-node latency is apparent. Differences between runs are due to different node assignments. Outliers $\geq$ 95th percentile are removed to maintain a reasonable color scale.

5.5 Discovery of currently unknown topology for ARM1

No information about the internal topology of ARM1 is published. Therefore ARM1 is a suitable candidate to verify that our PLogP micro-benchmark can be used to extract useful information for systems with unknown topology. Figure 5.10 shows the predicted $RTT(1)$ and latency $R = 128$. The first observation is that the plots are the inverse of those from FRAM and BETZY, with groups of high latency and $RTT(1)$ surrounding each other. This is because the experiments on ARM1 were done using OpenMPI, which by default maps ranks consecutive ranks to different sockets.

An automated parameter search on these measurements finds the following cluster results:

- 2 clusters of 48 cores each: $\{0, 2, 4, 6, \dots\}, \{1, 3, 5, \dots\}$
- 4 clusters of 24 cores each: $\{0, 4, 8, \dots\}, \{1, 5, 9, \dots\}, \{2, 6, 10, \dots\}\{3, 7, 11 \dots\}$

The cluster of high-latency ranks in the lower right corner is not explained by our understanding of the system. Therefore we repeat the experiment but with the `--map-by core` parameter that tells OpenMPI to map by core instead of by socket. Figure 5.11 shows the corresponding results from this experiment. Here we can see that ranks that are just next to each other along the diagonal have low latency. This corresponds with the pairs seen on the bottom of the dendrogram plot in Figure 5.12. The division into two 48-core sockets is clearly seen in both the $RTT(1)$ prediction plot and the latency plot. Several features in the plot, like the circular grouping in the top-left corner, cannot be explained without more information about the system design.

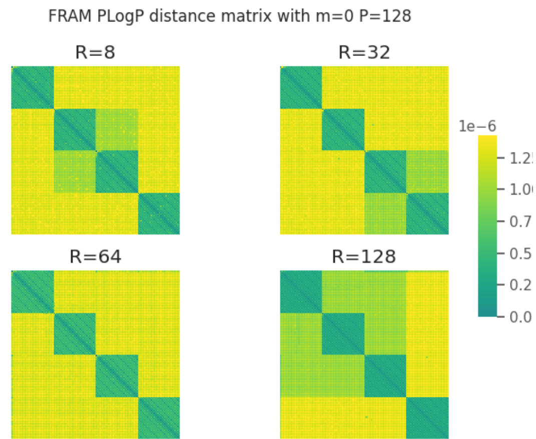
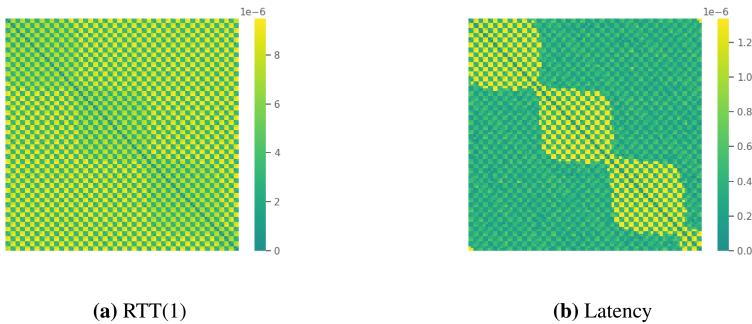


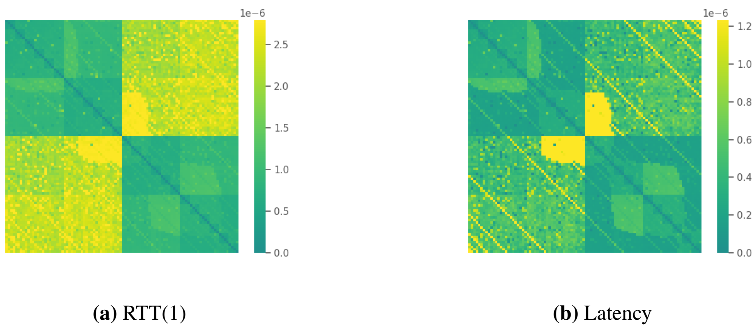
Figure 5.8: Communication time prediction for 4 nodes @ FRAM. Here intra-node latency is apparent. Differences between runs are due to different node assignments. Outliers $\geq$ 95th percentile are removed to maintain a reasonable color scale.



(a) RTT(1)

(b) Latency

Figure 5.10: Predicted RTT(1) and latency, R=8192 @ ARM1



(a) RTT(1)

(b) Latency

Figure 5.11: Predicted RTT(1) and latency, R=256 with –map-by core @ ARM1

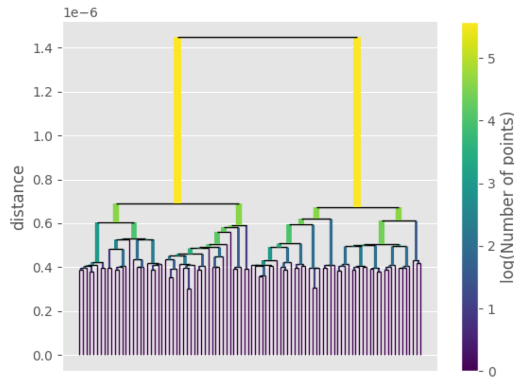


Figure 5.12: HDBSCAN Clustering dendrogram from measurements on ARM1

5.6 Find topology for BETZY

We want to understand how the topology of BETZY differs from that of FRAM.

Figure 5.13 shows the result of PLogP measurements on BETZY using 4 nodes with all 128 cores. Four clusters can be clearly seen that contain ranks belonging to the same node. For each node, we also see a partition into two groups. These groups match how cores on the same socket are assigned ranks. The measured $RTT(0)$ is $\approx 1.8\mu s$ for between nodes, $\approx 0.8\mu s$ between sockets, and the core-to-core communication between each core is $\approx 0.6\mu s$. This shows us that both inter-node topology and intra-node topology is detectable using only $R = 8$.

We are able to find several clusters from the PLogP measurements. An automated search finds a clustering of 128 clusters. Each cluster is 4 consecutive ranks, and this matches the clusters seen in Figure 5.14. The reason for this pattern is unknown. The clustering accuracy for the socket level is at 1 for all algorithms. The node and machine-level accuracy is 1.0 for all hierarchical clustering algorithms, except for SSSTree which has a $\approx 50\%$ node accuracy. See Figure 7.4 in the appendix for more details.

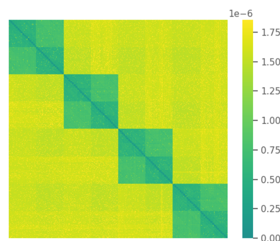


Figure 5.13: RTT prediction for m=0,R=8,P=512 @ BETZY

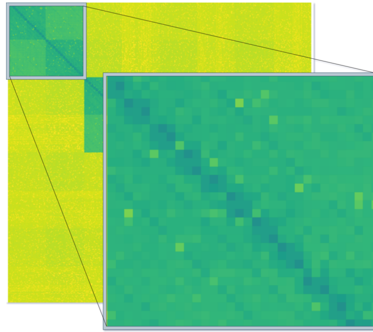


Figure 5.14: Zoom-in to top left corner of Figure 5.13

5.7 Baseline

Figure 5.15 shows the result of a PLogP micro-benchmark run on HPCLAB using all 4 cores. It shows the predicted $RTT(0)$ for the system for different R values. The matrix is different for each R, and without a clear pattern. The same search technique used to find clusters for the other systems only found the trivial clustering $\{0, 1, 2, 3\}$ on $RTT(0)$. For $RTT(2^{24})$ it found $\{0, 1\}, \{2, 3\}$, however $2^{24} \gg M$ so any measurement artifacts are amplified to a great extent. Using RTT does not work when the latency between ranks is so low that measurement artifacts are the dominating factor. Figure 5.16 shows the latency measured using the PLogP micro-benchmark for $R = 8196$. Here the latency between ranks is in the range of 15 ns to 70 ns. An exception is for the outlier for measurements between the first two ranks which have $160ns$ latency, but this extreme value is due to a measurement artifact. Figure 5.17 shows the core latency measured using compare-exchange operation from the open-source project `core-to-core-latency` [2]. The latency between cores on the Intel Core i7-6700k that is on HPCLAB has a core-to-core latency $\approx 20ns$, which is in the same order of magnitude as the values we measure using the PLogP micro-benchmark. The outlier for pair $(0, 1)$ has a significant impact on the experiment because the number of datapoints is so low for $P = 4$. Therefore it is not possible to detect intra-CPU level topologies using our PLogP measurement method.

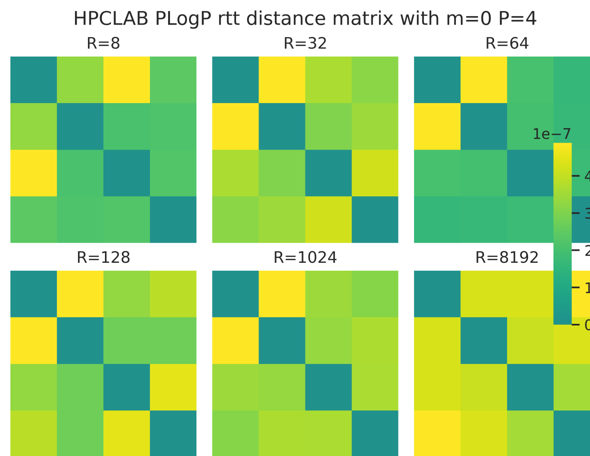


Figure 5.15: Predicted RTT(0) @ HPCLAB

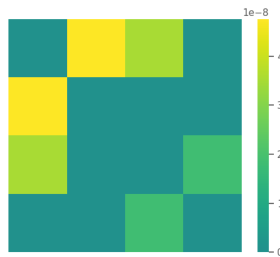


Figure 5.16: PLogP Latency, R=8196 @ HPCLAB

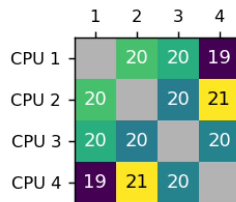


Figure 5.17: Core to core latency [2]

5.8 Scaling

Scalability is important for applications of our network topology detection technique. The $O(n^2)$ number of pairwise tests means that doing tests in parallel is a requirement for scaling to a large number of nodes. The PLogP extension of LGate allows for specifying a tree topology, where all subtrees for a given level in the tree can be tested in parallel. Therefore the performance of the micro-benchmark will heavily depend on the high-level topology it is given initially.

Figure 5.18 shows how the PLogP micro-benchmark scales with an increasing number of nodes and repetitions on FRAM. LGate was provided with a topology with all nodes. This allows LGate to run tests between all pairs on each node in parallel and is the reason why the runtime scales sub-exponentially. More information can be provided to LGate such that more tests can be run in parallel, which will further lower the total time for all pairwise tests.

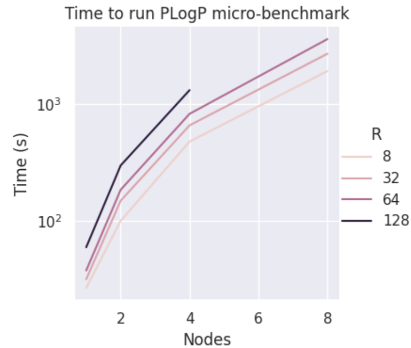


Figure 5.18: Time to run PLogP measurements for different numbers of nodes and repetitions, using all 32 cores of each node @ FRAM, M=2MiB

The time plot also shows that $R = 32$ only takes $\approx 50\%$ more time than $R = 8$. This is because the high number of additional measurements required for $R = 8$ causes it to take several times more time than expected. For $R = 32$, only a few additional measurements are needed. A key takeaway is that using a larger number of measurements can be better than the penalty of additional measurements.

5.9 Clustering algorithms

We have experimented with a variety of static and dynamic clustering methods for the automatic detection of network topologies. The static clustering methods score higher on our accuracy measurements than dynamic methods. This can be seen in Figure 7.3, where static methods have the highest score for each metric and value of R. Several other accuracy plots are available in Appendix A, where the same pattern occurs. However, static algorithms are only useful when the number of clusters is known, whereas dynamic algorithms will automatically determine the number of clusters in the dataset. The results show that for the methodology used in this thesis, static clustering algorithms like KMedoids and Agglomerative Clustering achieve high accuracy. Spectral clustering differs from the other static clustering algorithms and achieves the lowest overall accuracy. DBSCAN has the lowest accuracy among the dynamic clustering algorithms.

Hierarchical clustering algorithms like Agglomerative Clustering and HDBSCAN not only achieve high accuracy scores but also creates a hierarchy of clusters. From this hierarchy, it is possible to create a topology tree starting with nodes at the top and single

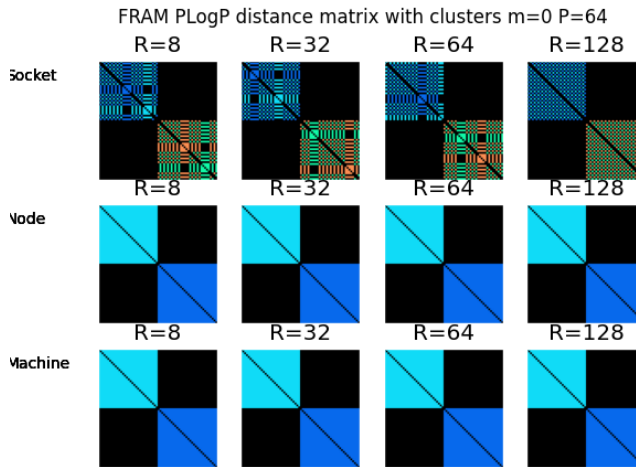


Figure 5.19: Some clusters found for two-nodes @ FRAM

cores at the bottom. Figure 2.4 shows such a graph. Hierarchical clustering algorithms are therefore useful for the problem of automatically detecting network topologies.

It should be noted that our accuracy results are computed from a limited selection of the entire parameter space of these clustering algorithms. Therefore the accuracy results is a representation of how difficult it is to find suitable clustering parameters.

Chapter 6

Case study: topology aware MPI broadcast

In this case study, we focus on improving broadcast operations in OpenMPI and Intel MPI, by using benchmarking and information about the network topology. We implement broadcast algorithms that use information about the network topology and compare it with the performance of OpenMPI and Intel MPI. The network topology is determined through PLogP measurements and clustering methods, as described previously in Chapter 3. Performance benchmarks are used to compare algorithms for different message sizes and cluster configurations. We use the results from the benchmarks to generate a parameterized broadcast function that selects the best-performing algorithm for a given message size.

6.1 Motivation

There are several reasons why we chose to look at the broadcast operation. First of all, the broadcast operation has different performance characteristics for different implementations of MPI, which we presume is the result of different implementations. Initial testing showed large differences between OpenMPI and Intel MPI. While there exist many different algorithms for the operation, the problem of choosing the best one to use in any single situation still has room for improvement. By default, open-source implementations such as OpenMPI use a pre-compiled heuristic to select the algorithms at runtime [39]. The heuristic function selects the broadcast algorithm based on the message size and the number of ranks in the communication group.

Wadsworth and Chen demonstrated through their paper *Performance of MPI Broadcast algorithms* [29] that changing broadcast algorithms can improve performance for MPICH2. In this case-study we will look at topology-based optimization of the broadcast operation for IMPI and OpenMPI.

6.2 Background

The broadcast operation is a collective operation that is used to send data from a single rank, called the root, to all other ranks in a group. It is useful when we have data in the root rank that needs to be shared with many other ranks. A trivial but slow implementation of the broadcast operation consists of the root rank sending to all other ranks in the group one by one. In MPI the broadcast operation is available through the `MPI_Bcast` function.

6.2.1 Broadcast in OpenMPI and Intel MPI

OpenMPI implements the following broadcast algorithms [40]:

- **Flat tree**: root sends to all other ranks one by one.
- **Chain tree**: rank r receives from rank $r - 1$ and sends to rank $r + 1$, making a communication chain.
- **Binary tree**: Ranks receive from one rank and send to up to two child nodes.
- **Split Binary** tree: the same assignment as the binary tree, but we split the message into two parts. The root then sends one part to its left sub-tree and the other to the right sub-tree. In the end, the left and right sub-tree swap their values and combine them into the entire message.
- **Split Binary** tree: the same assignment as the binary tree, but we split the message into two parts and send one part to the left sub-tree and the other to the right sub-tree, and swap between them at the end.
- **K-Chain tree**: similar to chain tree, but we create k chains, and the root sends to the start of each chain.
- **Binomial tree**: similar to the binary tree algorithm, but using a balanced binomial tree.

Many algorithms use message segmentation to avoid the rendezvous protocol by dividing the message into smaller messages and sending them after each other. We refer to these algorithms as *pipelined*, because they form a chain of N messages that are sent sequentially. A pipelined chain tree algorithm works by sending splitting the original message into parts and pushing these parts through the chain one after the other like in a pipeline.

Intel MPI (IMPI) provides other algorithms such as Binomial, Knomial Recursive Doubling, Shumilin's algorithm, and topology- and NUMA-aware algorithms [41]. Intel does not publish implementation details about those algorithms. An important detail is that OpenMPI does not use topology-aware collective operations by default, but IMPI might be using topology-aware operations.

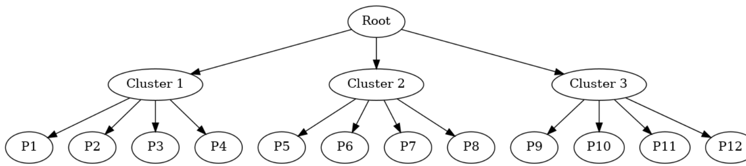


Figure 6.1: Fat tree with 12 processes in 3 clusters

6.3 Topology based optimization

6.3.1 Intra-node and inter-node optimization

The bandwidth and time to communicate are often lower between ranks that are running on the same node, than for ranks running on separate nodes. The same can apply to other levels in the topology hierarchy, such as the socket level and machine level. A useful optimization is therefore to minimize communication between siblings in the topology hierarchy.

6.3.2 Fat-tree-based communication

Fat tree-based communication can reduce total communication time if there is a difference in performance characteristics between processes. For example, we measure communication performance between 12 processes and find a cluster of size 3, as seen in Figure 6.1. If communication within each cluster is significantly lower than communication between clusters, then we should try to minimize communication between clusters. One way to achieve this can be seen in Figure 6.2 where P1 is the root broadcasting to all other processes using the binary tree broadcast algorithm. Here we only have two communications between clusters, and the rest are internal to the clusters, compared to the standard binary tree algorithm as seen in Figure 6.3 which has seven communications between clusters.

6.3.3 MagPIe

MagPIe is a collective communication library for MPI that is optimized for WANs (wide area networks). In their paper [30] the authors of MagPIE write that their implementation was up to 10 times faster than MPICH for a system with 10ms latency on links between clusters. MagPIe minimizes communication on links with high latency, with their broadcast only using it once. The communication pattern for MagPIe broadcasts is a fat tree.

6.4 Implementation

Several implementations of the MPI standard, such as OpenMPI and Intel MPI have support for at least some way to control the way collective operations like MPI_bcast are implemented. OpenMPI has built-in support for several `MPI_Bcast` implementations. By default, it uses a heuristic to select the implementation based on information like communicator size and message size. OpenMPI has a feature called the MCA (Modular Compo-

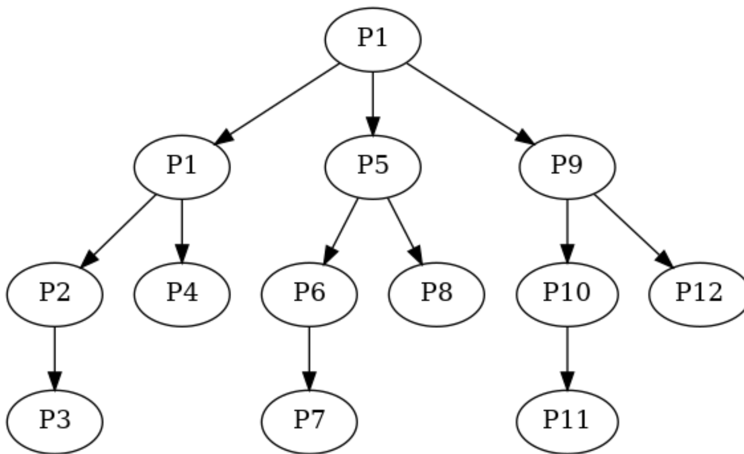


Figure 6.2: Communication ordering using clustering and binary tree bcast

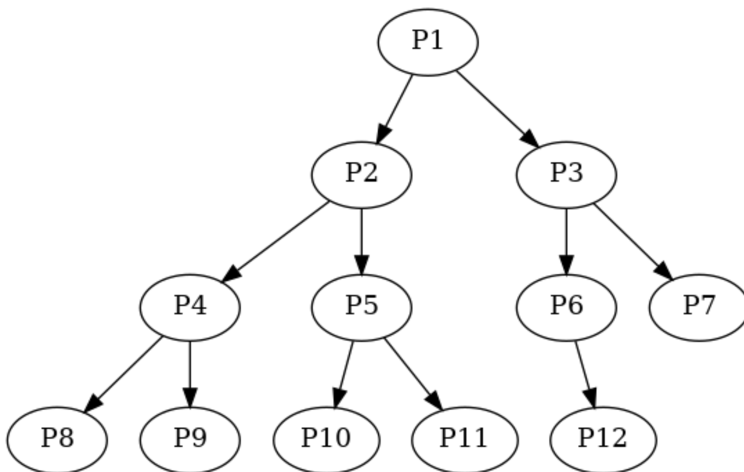


Figure 6.3: Communication ordering using binary tree bcast

nent Architecture) that allows setting configuration parameters for modules in OpenMPI. Using MCA it is possible to change some parameters of the ”coll” module for collective communication, thereby changing the selection of the broadcast algorithm. IMPI (Intel MPI) also has similar functionality for selecting `MPI_Bcast` implementation but uses a different interface.

We re-implement the algorithms from scratch by using point-to-point based communication functions such as `MPI_Send` and `MPI_Recv`. Both blocking and non-blocking point-to-point communication is used depending on the algorithm. This makes our implementation portable because it does not rely on details of the MPI implementation used.

6.4.1 Limitations

Some of the broadcast algorithm implementations only support sending from rank 0, and therefore all benchmarks are on only sending from rank 0. Adding support for sending from any other rank is trivial, and can be implemented by swapping indices in a lookup table.

6.4.2 Algorithms

We make a distinction between base-level algorithms and top-level algorithms. The base-level algorithms are only provided with a group of ranks and are not topology-aware. The top-level algorithms are topology-aware, and build a communication tree based on the topology provided by the caller. They use point-to-point based communication together with base-level algorithms as building blocks to create topology-aware broadcast functions.

Table 6.1 shows the base algorithms implemented and used in the case study. Note that all of these algorithms are implemented in OpenMPI. Table 6.2 shows the list of top-level algorithms used.

Name
Binary tree
Binary-tree with swap
Chain
Flat tree
Pipeline
Scatter-Gather

Table 6.1: List of base broadcast algorithms used

Name	Description
Magpie	Minimize use of slow links. Uses point-to-point communication.
Magpie2	Like magpie but using base-level algorithms to send to groups.
Interpipe	Like magpie, but with message segmentation.

Table 6.2: List of top-level broadcast algorithms used

6.5 Methodology

We created a benchmark suite to make it possible to compare the performance of MPI_bcast implementations. The benchmark takes a list of message sizes, topologies, and benchmark functions as input parameters. For each combination of these parameters it runs N measurements with W warmup rounds.

6.5.1 Statistical error

Due to the variable nature of network communication, we have to be cautious with how we benchmark. Experimentally, it was found that the first few measurements often deviated significantly from the rest of the measurements. Therefore we run some number of iterations W where we discard the result. These warmup rounds run before we start the real benchmark. For our benchmarks, $W = 50$ or $W = 100$ was used, depending on the system. We chose to use $N = 1000$ as the number of measurements for each configuration. The mean, variance, maximum and minimum time were recorded. Only values within the 90th percentile were used in further analysis.

6.5.2 Predicting and algorithmic selection of best broadcast function

The benchmark measurements can be used to predict the best broadcast function. By using the measurements for each message size for a selection of broadcast implementations we used linear regression to find the best function given the message size, as seen in Figure 6.4.

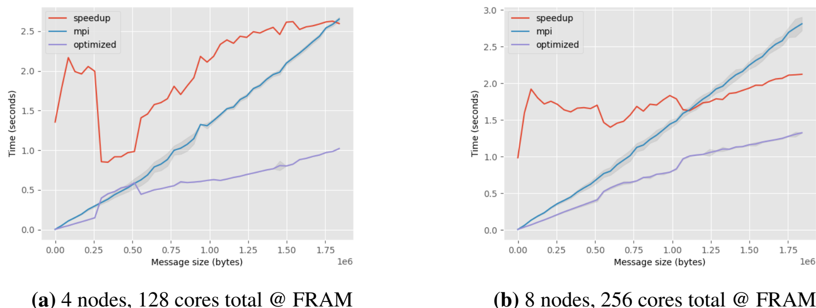
```
def choose_bcast(P, msgsize, impls):
    impl_preds = [(impl, predict_time(impl, msgsize))
                                    for impl in impls]
    return min(impl_preds, key=lambda x: x[1])
```

Figure 6.4: Selection of best broadcast implementation

Figure 6.12 shows a decision function generated by using this method on ARM1. Some limitations can be seen in the resulting function: most notably the alternation between the two very similar performing implementations magpie with `BCastDefault` and magpie with `BCastBtreeWSwap`.

6.6 Experiments

We run several experiments, on IDUN, FRAM, and ARM1. Each experiment starts by taking PLogP measurements using the LGate test harness. We then use clustering methods for each system. Using these clustering results we generate topology files. These topology files are then passed to the broadcast micro-benchmark which tests many different broadcast implementations for the different topologies. We use the `choose_bcast`



(a) 4 nodes, 128 cores total @ FRAM

(b) 8 nodes, 256 cores total @ FRAM

Figure 6.5: Broadcast results using OpenMPI @ FRAM

function on various message sizes to generate an optimized broadcast function. We run the broadcast benchmark again, testing the default `MPI_Bcast` and the optimized version we generated.

6.7 Results and discussion

We run several experiments on IDUN, FRAM, and ARM1 to see how our topology and optimized broadcast function performs compared to the default broadcast in Intel MPI and OpenMPI.

6.7.1 FRAM

We experiment with both Intel MPI and OpenMPI on FRAM. We use 128 and 256 cores. The topology used is generated from socket-level clusterings as shown in Section 5.3. Figure 6.5 shows a comparison of the default OpenMPI `MPI_Bcast` and the automatically optimized version. Here we can see that the speedup is around 1.5x for $P = 128$. Note the dip in speedup around 0.4 MiB. This is because the choose_bcast function chose the wrong algorithm because we only ran the benchmark for messages greater than 0.5 MiB. In general, the optimized version will always have a speedup ≥ 1.0 because it can choose to use the default implementation whenever it outperforms our algorithms. For messages above 0.5 MiB the average speedup is close to 2x, with a speedup of ≥ 2.5x for messages with size between 1.25 and 2 MiB. For $P = 256$ we can see that our implementation has a speedup of ≥ 1.5x for nearly the entire interval from 0 bytes to 2 MiB.

Figure 6.6 shows how each algorithm performs compared to the default in Intel MPI. It shows that there is a large difference between the different algorithms. Figure 6.7 shows Intel MPI compared to the optimized broadcast function. Here we are able to get a $1.5-2$x speedup for messages sizes $\{32, 128, 256\}$ MiB for $P = 128$.

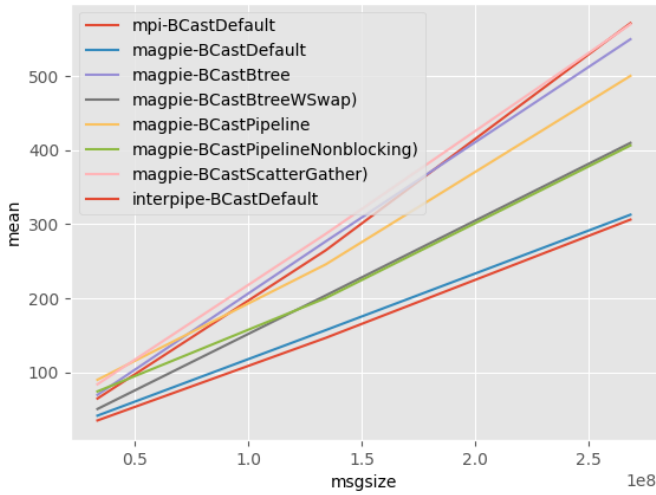


Figure 6.6: IMPI - 4 nodes, 128 cores total @ FRAM, 32MB, 128MB, and 256MB]

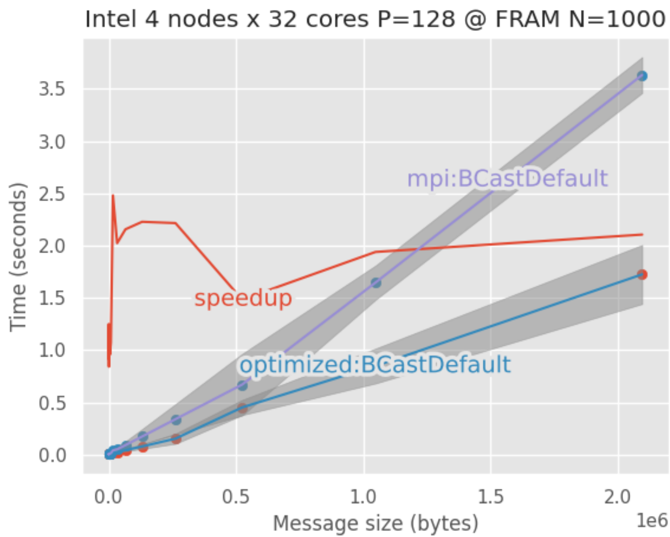


Figure 6.7: IMPI vs optimized - 4 nodes, 128 cores total @ FRAM]

P	OpenMPI speedup	Intel MPI speedup
128	≈ 2	≥ 1.5
256	≥ 1.4	??

Table 6.3: Speedup for P=128,256 for OpenMPI and Intel MPI @ FRAM

Table 6.4 shows part of the selection function used in the optimized broadcast function for several message sizes and values of P. Here we can see that using binary tree broadcast with magpie performs best for all tree P configurations. Scatter-gather performs better than the default implementation for $P \geq 128$, and magpie and scatter-gather outperform the default implementation as P increases.

P	≤ 16	4096	$\leq 2^{18}$	$\geq 2^{18}+1$
64	btree$_m$	m	default	default
128	btree$_m$	m	m	scatter gather
256	btree$_m$	m	scatter gather	scatter gather

Table 6.4: Best broadcast algorithm for message size and P, OpenMPI @ FRAM
* - non-blocking, `m` - using magpie

We have shown that our topology-aware broadcast is able to outperform both OpenMPI and Intel MPI in these experiments on FRAM.

6.7.2 BETZY

We run experiments by using two different types of topologies to P=512,1024 and 2048 on BETZY. The first topology is the socket level that we found using PLogP experiments on BETZY. The other is on the node level. We compare both Intel MPI and OpenMPI. For both IMPI and OpenMPI, we first run the benchmark to compare all algorithms for various message sizes. From this, we generate an optimized broadcast function and compare it against the baseline.

Table 6.5 shows the speedup of our optimized broadcast function, compared to the default MPI_Bcast for different P for both Intel MPI and OpenMPI. Here we can see we achieve higher speedup using OpenMPI than using Intel MPI. This is because the broadcast function Intel MPI performs better on BETZY than OpenMPI in these experiments. We only run measurements for certain message sizes. In our experiments, we used message size doubling and evenly spaced numbers between $m = 0$ and $m = M$. Figure 6.8 shows the results for Intel MPI. Here we can see that the node-level topology outperforms the socket-level topology. The reason for this is that we used a flat topology for the socket level, which can cause up to twice as many inter-node communications. This is due to a limitation in the current implementation of the topology-aware broadcast. A two-level topology with node level at the top, and socket level as the leaf level, would have higher performance if inter-socket communication time is higher than the extra overhead.

For BETZY we check for all powers of two up to 2^{21}. Table 6.6 shows a selection of the decision function for the optimized broadcast version found. We can see that for small sizes that binary tree with magpie is the best for both $P = 512$ and $P = 2048$, while

P	OpenMPI speedup	Intel MPI speedup
512	≥ 2.0	≥ 1.5
1024	≥ 1.2	1.2
2048	1.1	1

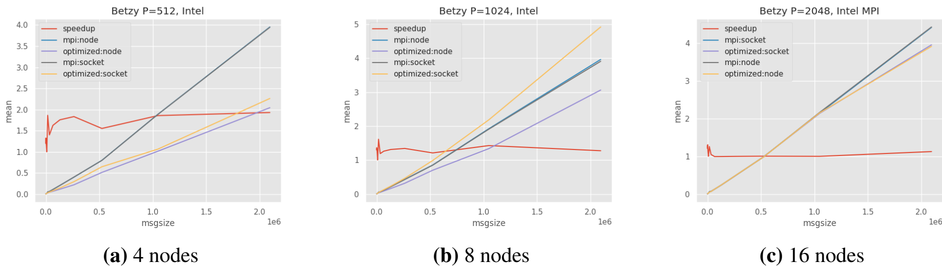
Table 6.5: Speedup for P=512,1024,2048 for OpenMPI and Intel MPI @ BETZY

P	≤ 64	$\leq 2^{19}$	$\leq 2^{20}$	$\leq 2^{21}$	$\geq 2^{21}+1$
512	btree$_m$	btree$_m$	pipeline$*_m$	m	scatter gather
1024	m		pipeline$_m$	scatter gather	scatter gather
2048	btree$_m$	various	scatter gather	pipeline$*_m$	pipeline$_m$

Table 6.6: Best broadcast algorithm for message size and P, OpenMPI @ BETZY
* - non-blocking, m - using magpie

$P = 1024$ selects to use magpie with the default broadcast implementation. We can also see that the scatter-gather is selected for large messages when $P = 512$ and $P = 1024$, but pipeline with magpie is the best for $P = 2048$. We can see that the selection of the broadcast algorithm varies significantly with the number of processes P and the message size M. For $P = 2048$ it is best to use a non-blocking pipeline for messages with sizes in the range $[2^{20}+1, 2^{21}]$, while for larger messages the blocking variant performs better.

The results show that our optimized implementation outperforms state-of-the-art implementations such as OpenMPI and Intel MPI when using node-level topology for $P \in \{512, 1024, 2048\}$ for power-of-two message sizes. The results also demonstrate that the selection of broadcast algorithm varies significantly depending on message size and process count. This suggests that it is difficult to find a generic hard-coded variant that also utilizes the system optimally. and therefore could be a good application for auto-tuning.

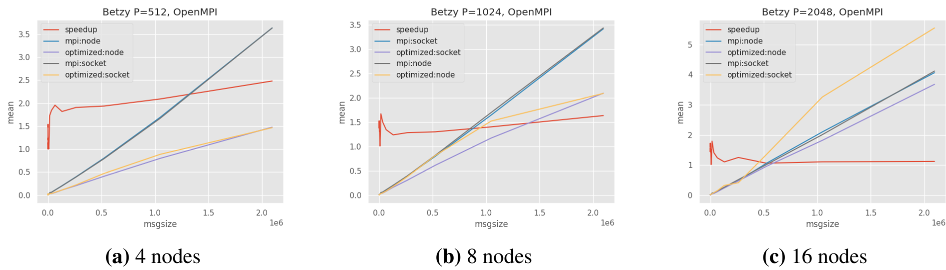


(a) 4 nodes **(b)** 8 nodes **(c)** 16 nodes

Figure 6.8: Broadcast benchmark results using Intel MPI @ BETZY

6.7.3 IDUN

For IDUN we run experiments on 16 nodes with 28 cores per node using OpenMPI and Intel MPI. We use a topology with 16 clusters with 28 cores each. This topology matches the node level we would find using the PLogP micro-benchmark. Figure 6.10 shows that using



(a) 4 nodes **(b)** 8 nodes **(c)** 16 nodes

Figure 6.9: Broadcast benchmark results using OpenMPI @ BETZY

this topology with our broadcast selection algorithm we can achieve a near 2x speedup for most message sizes in the range $0 \ldots 8$ MiB compared to OpenMPI. A similar benchmark was run using Intel MPI but this only achieved 0.9x speedup. The optimized decision function chooses to use the built-in MPI_Bcast whenever it performs best. The reason we got a speedup of 0.9x is that the experiment used another rank distribution method, and therefore the provided topology did not match the ranks in the actual experiment.

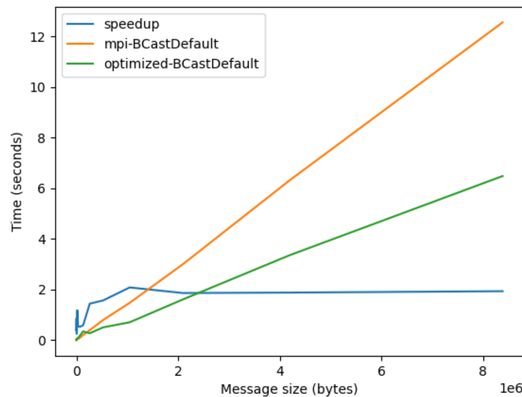


Figure 6.10: 16 nodes with 28 cores comparison of OpenMPI Bcast @ IDUN

6.7.4 ARM1

Being an ARM-based system there is no support for the Intel MPI library on ARM1. It is also the only system with only a single node. Figure 6.11 shows the benchmark results and speedup for the two topologies we found for ARM1 using the PLogP micro-benchmark. We can see that the socket-level clustering and the other with 4 clusters have very similar performance. The difference between them is amplified when comparing the optimized versions. Here the 4 cluster topology gets a ≥ 1.5x speedup compared to the baseline.

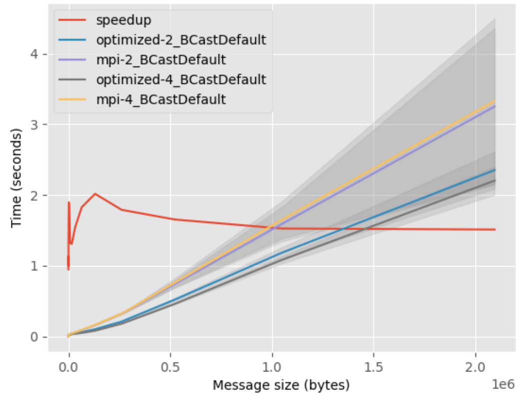


Figure 6.11: Performance and speedup curve of optimized BCast using 2 and 4 clusters compared to OpenMPI default on ARM1. 50 warmup-rounds and 1000 repetitions.

```
def choose_best_bcast(msgsize):
  if msgsize <= 1: use BCastPipeline
  elif msgsize <= 8: use BCastDefault
  elif msgsize <= 256: use magpie with BCastDefault
  elif msgsize <= 512: use magpie with BCastBtreeWSwap
  elif msgsize <= 2048: use magpie with BCastDefault
  elif msgsize <= 4096: use magpie with BCastBtreeWSwap
  elif msgsize <= 262144: use magpie with BCastDefault
  elif msgsize <= 2097152: use magpie with BCastBtreeWSwap
  else: use magpie with BCastBtreeWSwap
```

Figure 6.12: Optimized broadcast decision function for ARM1 with 4 cluster topology

6.8 Summary

In this case study we use the results from the PLogP micro-benchmark to create topologies. We implemented a topology-aware broadcast algorithm inspired by the work of Kielmann *et al.* that utilizes those topologies. In addition, we implemented several other broadcast algorithms and feed benchmark data to a program that automatically generates an optimized parameterized version for the system it runs on.

The results show that the parameterized topology-aware broadcast algorithm is able to outperform both OpenMPI and Intel MPI on several machines. They also show that the automatic selection method used has potential for both large clusters thousands of total cores and for single-node computers. This case study also supports the conclusion that

our PLogP micro-benchmark and clustering methods are able to detect useful topologies in clusters and single-node systems.

Chapter 7

Conclusion

In this thesis, the LGate pairwise test harness was extended with a PLogP micro-benchmark. We use the parameters found from the micro-benchmark to create several distance matrices that model the properties of the network. The results show that we are able to extract information on the node-, machine- and socket level for several machines. We find that detecting socket-level topologies require more detail in the distance matrix than the machine- and node-level. The results show that we can obtain higher accuracy by increasing the number of measurements we do between each pair. Using $R = 128$ has been sufficient to detect socket-level topology for all systems used in this thesis.

Clustering algorithms are used to extract network topology using the distance matrices created from the pairwise PLogP measurements. We use *apriori* information about the system such as the number of nodes, hosts, and sockets, along with the exact rank assignment, to create accuracy metrics that can be used to understand how well the clustering algorithms work for extracting topological information. We show that K-medoids, HDBSCAN, Connected Components, and Agglomerative clustering are all suitable for extracting topologies from distance matrices. Dynamic hierarchical clustering algorithms like HDBSCAN perform nearly as well as static clustering algorithms and creates a dendrogram which is useful for creating hierarchical topologies. The SSSTree algorithm also creates a tree that can be used to create a hierarchical topology but is more sensitive to parameter values than HDBSCAN.

We implement a parameterized topology-aware broadcast function that takes the topology we find by using the clustering algorithm. We run benchmarks on several algorithms and generate an optimized parameterized broadcast function from these results. Our results show that the optimized broadcast achieves significant speedup for several clusters and a single-node system. We compare the optimized broadcast function with both OpenMPI and Intel MPI. Compared to OpenMPI it is able to achieve a speedup greater than 1.1 for sizes up to $P = 2048$ on BETZY. The speedup is greater for OpenMPI than for Intel MPI, and we are only able to achieve a speedup greater than 1.1 for $P = 1024$ using Intel MPI on the same system. Applied to the single-node system ARM1 we are able to achieve 1.5x speedup by utilizing topology information that was discovered by the pairwise tests.

We find that it is possible to extract useful information from the distance matrices generated from PLogP-measurements with as few as $R = 128$ measurements, and we have demonstrated this by using the topology in our topology-aware broadcast function to achieve a speedup of up to 2.5x.

7.1 Further work

This thesis has focused on three components that demonstrate the usefulness of pairwise measurements to detect topologies; taking measurements, extracting topologies from the measurements, and showing that these topologies can be used to improve algorithm performance. There are opportunities for further work on all three of these components. A limitation of our technique is that it does not work in real-time, and therefore assumes that the performance characteristics measured during the PLogP micro-benchmark do not change. Therefore, work on an adaptive variant that can take measurements during the execution of a program, could lead to better results in real-world scenarios.

Both the PLogP micro-benchmark and the clustering algorithms have parameters and configuration options that heavily impact the performance. More work is needed to determine the optimal parameters for each system, and techniques that allow for automated search, such as auto-tuning techniques, might be applicable.

Our parameterized topology-aware broadcast algorithm has incomplete support for multi-level topologies, and therefore all benchmarks were done using flat topologies. It would be of interest to test and benchmark the multi-level topologies. The selection of optimal broadcast algorithms and benchmarking is already partly automated, but the search techniques used in this case study are not optimal. We run measurements for all methods even if they take 100x more time than the fastest one for a given message size. Utilizing more sophisticated techniques can significantly reduce this search for an optimal broadcast decision function, and therefore allow for application to larger node- and core counts. Techniques that speed up this search can also allow for a more detailed decision function, such as dealing with larger message sizes, non-power-of-two message sizes, and more topologies.

Acknowledgement

The experiments on IDUN were performed on resources provided by NTNU [35]. The experiments on FRAM and BETZY were performed on resources provided by Sigma2 - the National Infrastructure for High-Performance Computing and Data Storage in Norway. The HPC-Lab at NTNU provided the HPCLAB computer used to write this thesis and do baseline experiments.

Bibliography

[1] T. Kielmann, H. Bal, and K. Verstoep, "Fast measurement of LogP parameters for message passing platforms," in *PARALLEL AND DISTRIBUTED PROCESSING, PROCEEDINGS*, ser. Lecture Notes in Computer Science, J. Rolim, Ed., vol. 1800. IEEE Comp Soc, Tech Comm Parallel Proc; ACM SIGARCH, pp. 1176–1183, ISSN: 0302-9743.

[2] nviennot, "core-to-core-latency." [Online]. Available: https://github.com/nviennot/core-to-core-latency/tree/main

[3] K. Barker, K. Davis, A. Hoisie, D. Kerbyson, M. Lang, S. Pakin, and J. Sancho, "Using performance modeling to design large-scale systems," vol. 42, pp. 42–49.

[4] R. W. Hockney, "The communication challenge for MPP: Intel paragon and meiko CS-2," vol. 20, no. 3, pp. 389–398. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819106800219

[5] L. G. Valiant, "A bridging model for parallel computation," vol. 33, no. 8, pp. 103–111, place: New York, NY, USA Publisher: Association for Computing Machinery. [Online]. Available: https://doi.org/10.1145/79173.79181

[6] D. CULLER, R. KARP, D. PATTERSON, A. SAHAY, K. SCHAUSER, E. SANTOS, R. SUBRAMONIAN, and T. VONEICKEN, "LOGP - TOWARDS a REALISTIC MODEL OF PARALLEL COMPUTATION," vol. 28, no. 7, pp. 1–12, publisher: ASSOC COMP MACHINERY, SPECIAL INTEREST GRP PROGRAMMING LANGUAGES.

[7] M. Snir, S. Otto, and S. Huss-Lederman, *MPI–the Complete Reference: the MPI core*. MIT press, vol. 1.

[8] W. Gropp, "MPI (message passing interface)," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, pp. 1184–1190. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_222

[9] M. J. Rashti and A. Afsahi, “Improving communication progress and overlap in MPI rendezvous protocol over RDMA-enabled interconnects,” in *2008 22nd International Symposium on High Performance Computing Systems and Applications*, pp. 95–101.

[10] R. B. Ganapathi, A. Gopalakrishnan, and R. W. Mcguire, “MPI process and network device affinitization for optimal HPC application performance,” in *2017 IEEE 25TH ANNUAL SYMPOSIUM ON HIGH-PERFORMANCE INTERCONNECTS (HOTI)*. IEEE; ARISTA; intel; Mellanox Technologies; Lenovo; IEEE Comp Soc, pp. 80–86.

[11] “Intel MPI,” publication Title: Intel. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html#gs.08sc6s

[12] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, “Open MPI: Goals, concept, and design of a next generation MPI implementation,” in *Proceedings, 11th European PVM/MPI Users’ Group Meeting*, pp. 97–104.

[13] M. Hafeez, S. Asghar, U. A. Malik, A. u. Rehman, and N. Riaz, “Survey of MPI implementations,” in *Digital Information and Communication Technology and Its Applications*, H. Cherifi, J. M. Zain, and E. El-Qawasmeh, Eds. Springer Berlin Heidelberg, pp. 206–220.

[14] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, “A high-performance, portable implementation of the MPI message passing interface standard,” vol. 22, no. 6, pp. 789–828. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167819196000245

[15] R. A. v. d. Geijn and K. Goto, “BLAS (basic linear algebra subprograms),” in *Encyclopedia of Parallel Computing*.

[16] R. C. Whaley, “ATLAS (automatically tuned linear algebra software),” in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, pp. 95–101. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_85

[17] “OpenBLAS : An optimized BLAS library.” [Online]. Available: http://www.openblas.net

[18] “Get started with intel oneAPI math kernel library,” publication Title: Intel. [Online]. Available: https://www.intel.com/content/www/us/en/docs/onemkl/get-started-guide/2023-0/overview.html

[19] N. Brisaboa, O. Pedreira, D. Seco, R. Solar, and R. Uribe, “Clustering-based similarity search in metric spaces with sparse spatial centers,” in *SOFSEM 2008: THEORY AND PRACTICE OF COMPUTER SCIENCE*, ser. Lecture Notes in Computer Science, V. Geffert, J. Karhumaki, A. Bertoni, B. Preneel, P. Navrat, and M. Bielikova, Eds., vol. 4910. Asseco Slovakia; Data Informat Technol & Expert Consulting; European Res Consortium Informat & Math; Hewlett Packard Slovakia; IBM Slovakia; Siemens Slovakia; SOFTEC, pp. 186+, ISSN: 0302-9743.

[20] J. MacQueen, "Classification and analysis of multivariate observations," in *5th Berkeley Symp. Math. Statist. Probability*. University of California Los Angeles LA USA, 1967, pp. 281–297.

[21] M. Ahmed, R. Seraj, and S. Islam, "The k-means algorithm: A comprehensive survey and performance evaluation," vol. 9, p. 1295.

[22] O. Kariv and S. L. Hakimi, "An algorithmic approach to network location problems. II: The p-medians," vol. 37, no. 3, pp. 539–560, _eprint: https://doi.org/10.1137/0137041. [Online]. Available: https://doi.org/10.1137/0137041

[23] H.-S. Park and C.-H. Jun, "A simple and fast algorithm for k-medoids clustering," vol. 36, no. 2, pp. 3336–3341. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095741740800081X

[24] E. Schubert and L. Lenssen, "Fast k-medoids clustering in rust and python," vol. 7, no. 75, p. 4183, publisher: The Open Journal. [Online]. Available: https://doi.org/10.21105/joss.04183

[25] M. Tiwari, M. J. Zhang, J. Mayclin, S. Thrun, C. Piech, and I. Shomorony, "Bandit-PAM: Almost linear time k-medoids clustering via multi-armed bandits," vol. abs/2006.06856. [Online]. Available: https://arxiv.org/abs/2006.06856

[26] A. Ng, M. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," vol. 14.

[27] R. J. G. B. Campello, D. Moulavi, and J. Sander, "Density-based clustering based on hierarchical density estimates," in *Advances in Knowledge Discovery and Data Mining*. Springer, pp. 160–172, journal Abbreviation: SpringerLink.

[28] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, ser. KDD'96. AAAI Press, pp. 226–231, event-place: Portland, Oregon.

[29] D. M. Wadsworth and Z. Chen, "Performance of MPI broadcast algorithms," in *2008 IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL & DISTRIBUTED PROCESSING, VOLS 1-8*. IEEE, pp. 3049–3055.

[30] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "MagPIe: MPI's collective communication operations for clustered wide area systems," vol. 34, no. 8, pp. 131–140, place: New York, NY, USA Publisher: Association for Computing Machinery. [Online]. Available: https://doi.org/10.1145/329366.301116

[31] Y. Gong, B. He, and J. Zhong, "Network performance aware MPI collective communication operations in the cloud," vol. 26, no. 11, pp. 3079–3089.

[32] D. Hensgen, R. Finkel, and U. Manber, "Two algorithms for barrier synchronization," vol. 17, pp. 1–17.

[33] M. Godbolt, “Compiler explorer.” [Online]. Available: https://godbolt.org

[34] T. Gamblin, M. P. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. d. Supinski, and W. S. Futral, “The spack package manager: Bringing order to HPC software chaos,” in *Supercomputing 2015 (SC'15)*. [Online]. Available: http://tgamblin.github.io/pubs/spack-sc15.pdf

[35] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, “Epic: An energy-efficient, high-performance gpgpu computing research infrastructure,” 2022.

[36] s. , “IDUN hardware – high performance computing group.” [Online]. Available: https://www.hpc.ntnu.no/idun/hardware

[37] S. , “Fram sigma2 documentation.” [Online]. Available: https://documentation.sigma2.no/hpc_machines/fram

[38] ——, “Betsy sigma2 documentation.” [Online]. Available: https://documentation.sigma2.no/hpc_machines/betsy

[39] “ompi/ompi/mca/coll/tuned/coll_tuned_decision_dynamic.c at main \cdot open-mpi/ompi,” publication Title: GitHub. [Online]. Available: https://github.com/open-mpi/ompi/blob/main/ompi/mca/coll/tuned/coll_tuned_decision_fixed.c#L512

[40] E. Nuriyev, J.-A. Rico-Gallego, and A. Lastovetsky, “Model-based selection of optimal MPI broadcast algorithms for multi-core clusters,” vol. 165, pp. 1–16. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731522000697

[41] intel, “An introduction to the intelıfmmode\circledR\else®\fi QuickPath interconnect,” publication Title: Intel. [Online]. Available: https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html

Appendix

7.1.1 A - Clustering accuracy

Clustering accuracy for FRAM P=32

Clustering accuracy for FRAM P=64

Clustering accuracy for FRAM P=128

Clustering accuracy for BETZY P=512

Clustering accuracy for ARM1 P=96

7.1.2 B - Latency matrices

Latency matrix for FRAM P=32, single node

Latency matrix for FRAM P=128, 4 nodes

Latency matrix for ARM1 P=96

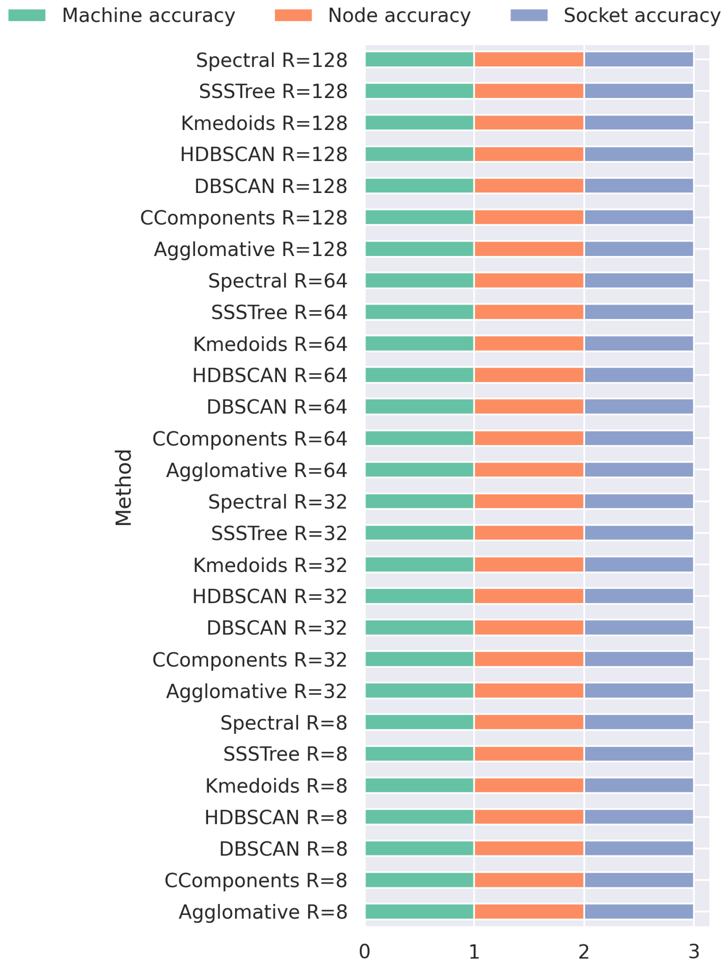


Figure 7.1: Clustering accuracy for values of R, P=32 @ FRAM

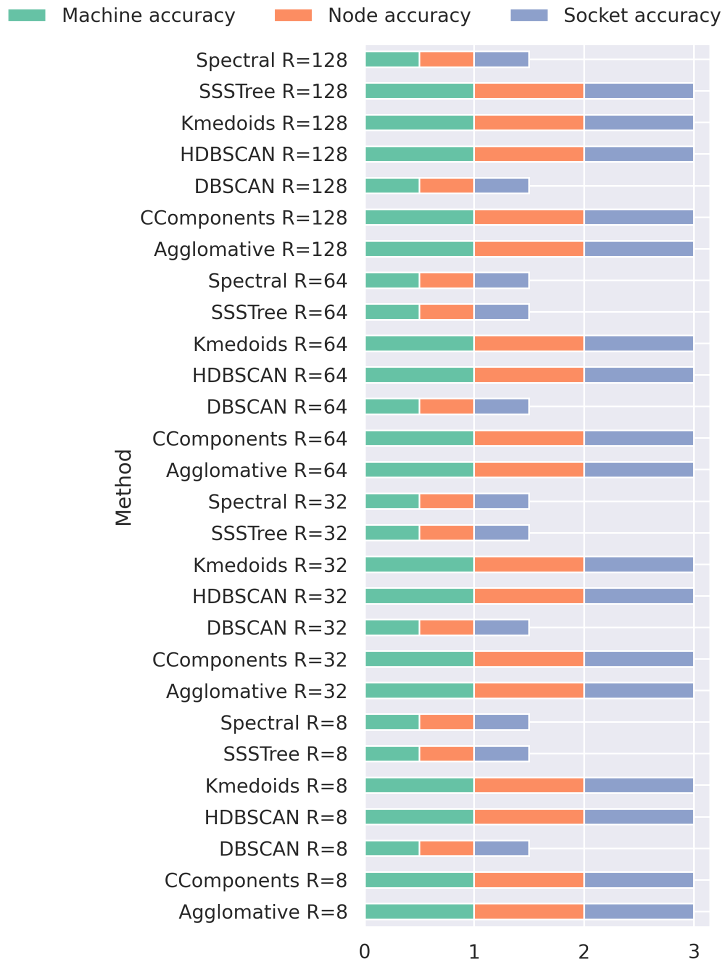


Figure 7.2: Clustering accuracy for values of R, P=64 @ FRAM

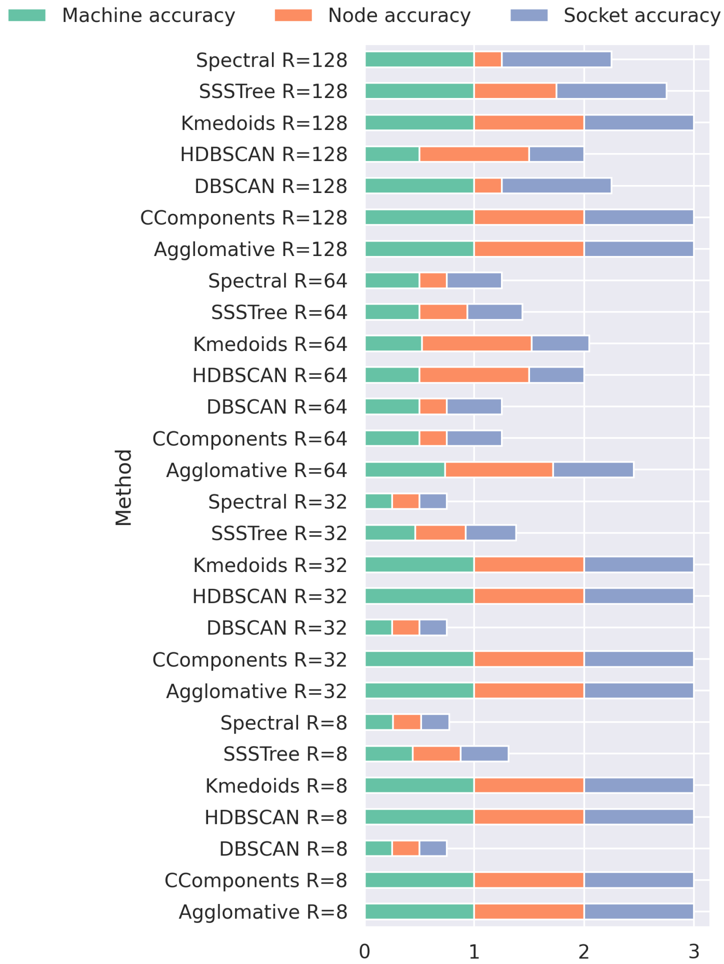


Figure 7.3: Clustering accuracy for values of R, P=128 @ FRAM

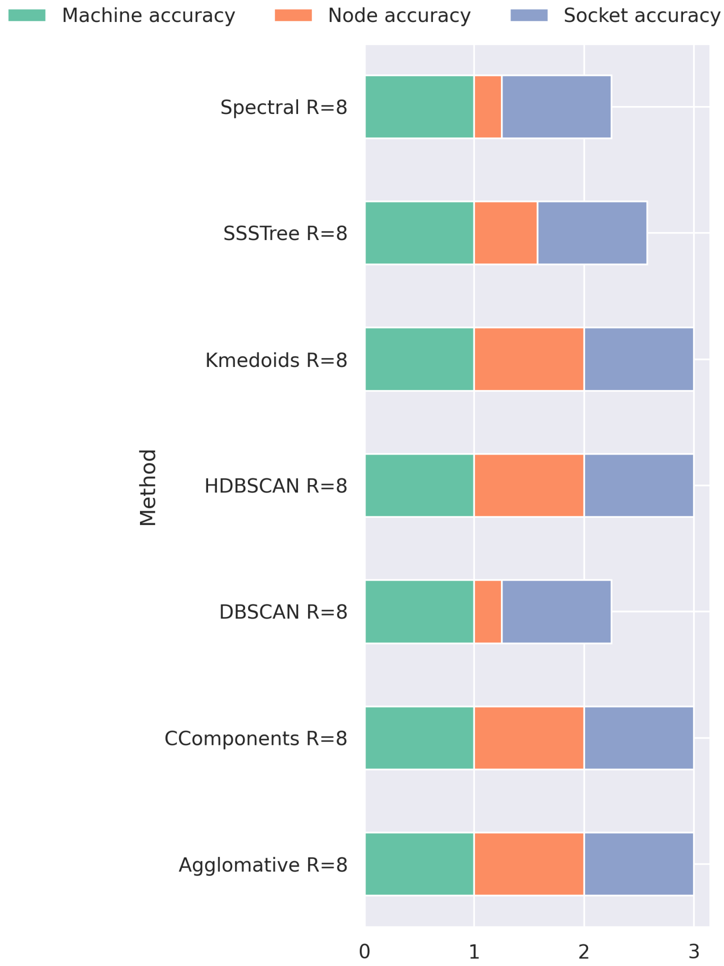


Figure 7.4: Clustering accuracy for values of R, P=512 @ BETZY

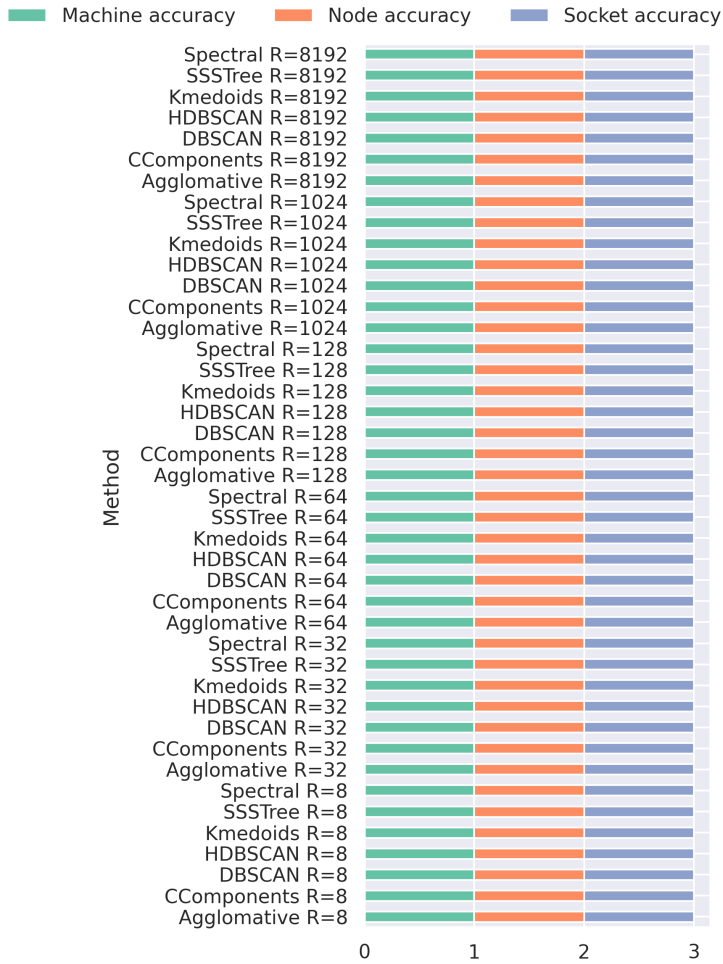


Figure 7.5: Clustering accuracy for values of R, P=96 @ ARM1

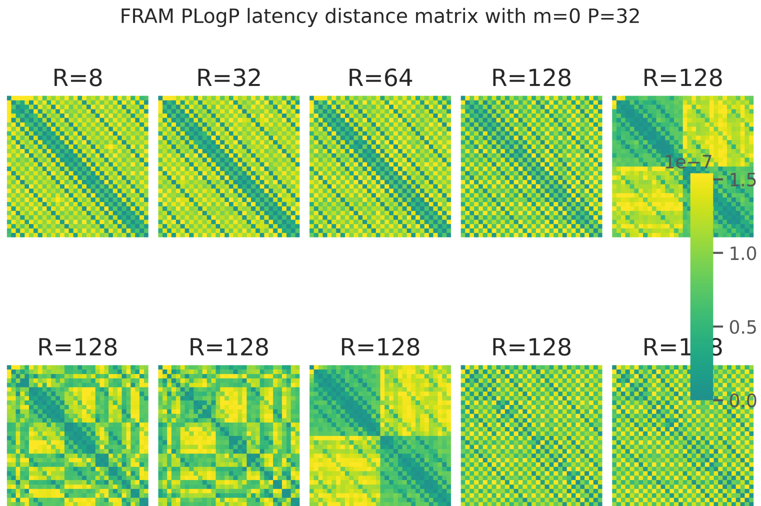


Figure 7.6: Latency P=32 @ FRAM

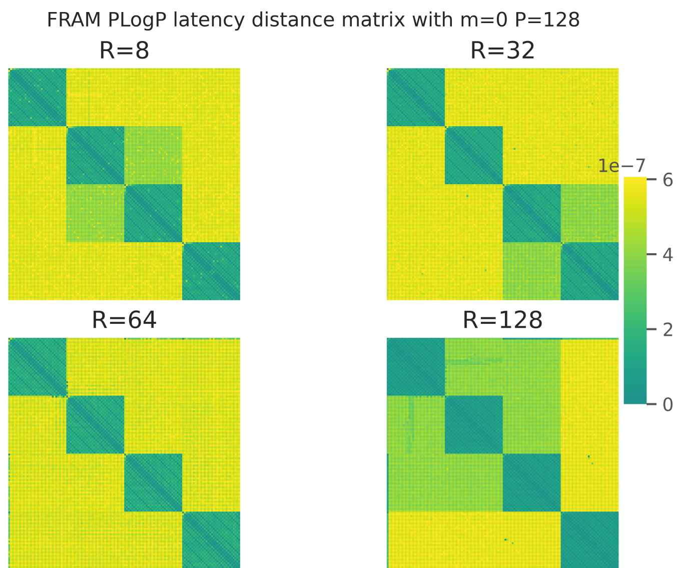


Figure 7.7: Latency P=128 @ FRAM

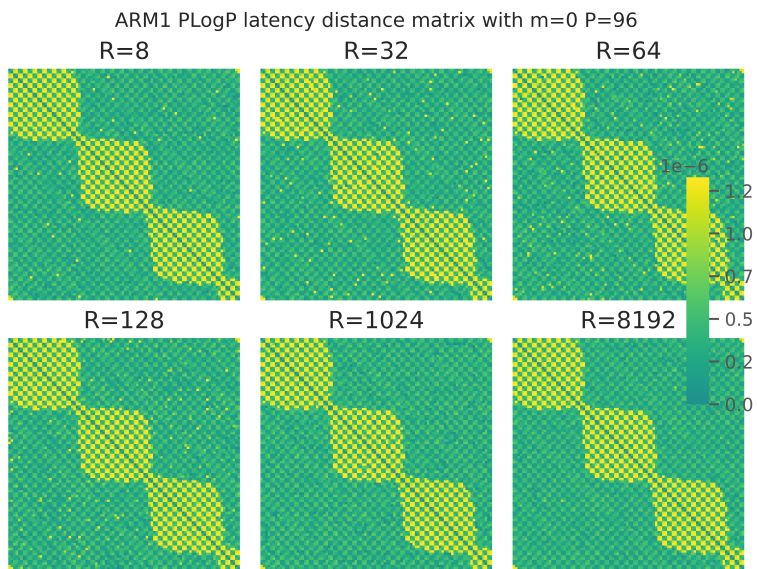
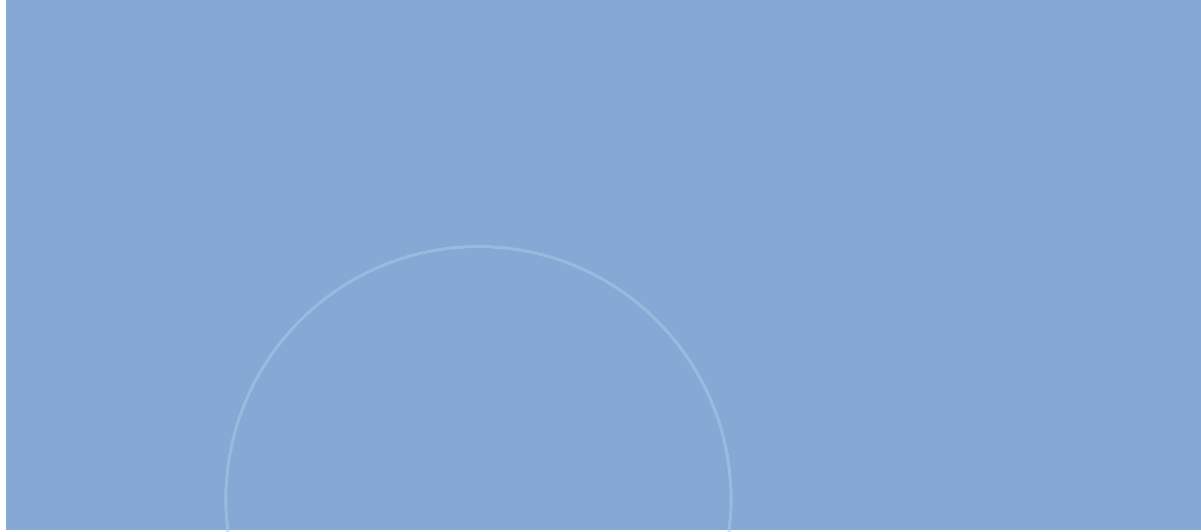
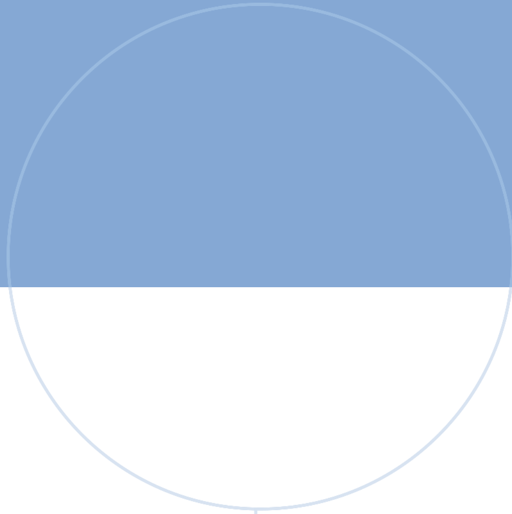
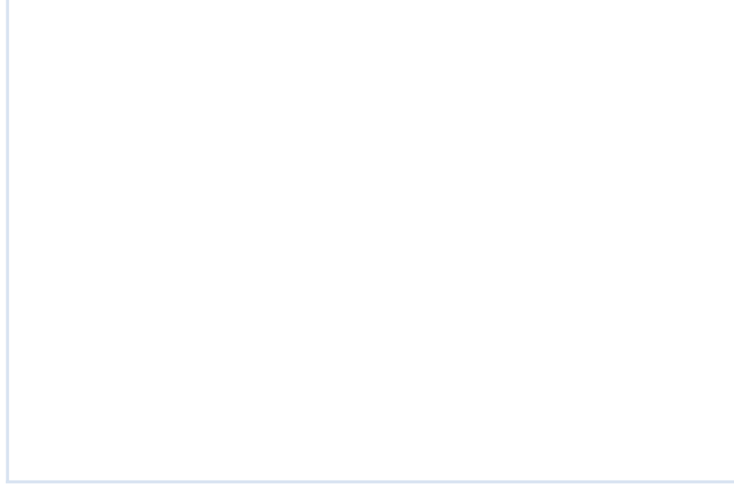


Figure 7.8: Latency P=96 @ ARM1

NTNU
Norwegian University of
Science and Technology