SANTA CATARINA STATE UNIVERSITY – UDESC
COLLEGE OF TECHNOLOGICAL SCIENCE – CCT
GRADUATE PROGRAM IN APPLIED COMPUTING – PPGCA

WILTON JACIEL LOCH

SPARBIT: A NEW LOGARITHMIC-COST AND DATA LOCALITY-AWARE MPI ALLGATHER ALGORITHM

JOINVILLE
2021

WILTON JACIEL LOCH

SPARBIT: A NEW LOGARITHMIC-COST AND DATA LOCALITY-AWARE MPI ALLGATHER ALGORITHM

Master thesis presented to the Graduate Program in Applied Computing of the College of Technological Science from the Santa Catarina State University, as a partial requisite for receiving the Master's degree in Applied Computing.

Supervisor: Guilherme Piêgas Koslovski

JOINVILLE
2021

Ficha catalográfica elaborada pelo programa de geração automática da

Biblioteca Setorial do CCT/UDESC,

com os dados fornecidos pelo(a) autor(a)

Loch, Wilton Jaciel
Sparbit : a new logarithmic-cost and data locality-aware MPI Allgather algorithm / Wilton Jaciel Loch. -- 2021.
99 p.

Orientador: Guilherme Piêgas Koslovski
Dissertação (mestrado) -- Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada, Joinville, 2021.

1. Alto Desempenho. 2. Message-Passing. 3. Comunicação Coletiva. 4. Algoritmos Coletivos. 5. Allgather. I. Koslovski, Guilherme Piêgas. II. Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada. III. Título.

WILTON JACIEL LOCH

SPARBIT: A NEW LOGARITHMIC-COST AND DATA LOCALITY-AWARE MPI ALLGATHER ALGORITHM

Master thesis presented to the Graduate Program in Applied Computing of the College of Technological Science from the Santa Catarina State University, as a partial requisite for receiving the Master's degree in Applied Computing.

Supervisor: Guilherme Piêgas Koslovski

THESIS COMMITTEE:

Guilherme Piêgas Koslovski, P.h.D.
Santa Catarina State University

Members:

Maurício Aronne Pillon, P.h.D.
Santa Catarina State University

Rodrigo da Rosa Righi, P.h.D.
University of Rio dos Sinos Valley

Joinville, August 31st 2021

ACKNOWLEDGEMENTS

I would like to first and foremost thank my parents Wilson Loch and Jaice Aparecida Back Loch for the love, support and their lifelong relentless effort that allowed me to reach this point. For the encouragement, help and love which made all my days brighter I thank my girlfriend Eduarda Cristina Rosa. I am also thankful to my dear friends Vander Junior, Gabrielly Pazetto, César de Oliveira, Rafael Lunelli, Guilherme Schreiber, Bruno Trainotti and many others whose conversations, jokes and nights out provided me joy and motivation. I would like to thank my advisor, professor Guilherme Piêgas Koslovski for the knowledge, help and trust, without which this research would not exist. For the valuable suggestions and motivation to help the work to be better I thank the members of the thesis committee, professors Maurício Aronne Pillon and Rodrigo da Rosa Righi. I am thankful to the Santa Catarina State Research and Innovation Support Foundation (FAPESC) whose funding throughout these dubious times allowed the enrichment of this work. I thank Marcelo Pasin and the University of Neuchâtel for providing access to the experimental infrastructures and finally, I thank everyone who was not directly mentioned but in any way contributed to the development of this work.

“We are led by the work of others into the presence of the most beautiful treasures, which have been pulled from darkness and brought to light. From no age are we debarred, we have access to all; and if we want to transcend the narrow limitations of human weakness by our expansiveness of mind, there is a great span of time for us to range over.” (SENECA, [49 AD])

ABSTRACT

The collective operations are considered critical for improving the performance of exascale-ready and high-performance computing applications. On this thesis we focus on the Message-Passing Interface (MPI) Allgather and Allgatherv many to many collectives, which are amongst the most called and time-consuming operations. Each MPI algorithm for these calls suffers from different operational and performance limitations, that might include only working for restricted cases, requiring linear amounts of communication steps with the growth in number of processes, memory copies and shifts to assure correct data organization, and non-local data exchange patterns, most of which negatively contribute to the total operation time. All these characteristics create an environment that demands careful choices of alternatives to execute the call and where there is no *silver bullet* algorithm, which is the best for all cases. As scientific contribution we propose the Stripe Parallel Binomial Trees (Sparbit) algorithm, which employs the binomial tree distribution to perform data exchanges with optimal time costs and no usage restrictions. It also maintains a much more local communication pattern that minimizes the delays due to long range exchanges, allowing the extraction of more performance from current systems when compared with asymptotically equivalent traditional algorithms. Experimental results indicate that near 40% of all calls to Allgather could experience mean reductions from 20% to 28% on execution time by employing Sparbit, with maximum reductions reaching near 74%. For Allgatherv, results are highly variable depending on the distribution of block sizes along the processes.
Keywords: High-Performance. Message-Passing. Collective Communication. Collective Algorithms. Allgather. Allgatherv.

RESUMO

As operações coletivas têm fundamental importância nos esforços para melhorar aplicações de alto desempenho e em suas preparações para a era de Exascale. Este trabalho volta-se para as chamadas Allgather e Allgatherv de todos para todos, constituintes do padrão Message-Passing Interface (MPI) e que figuram dentre as mais utilizadas e com maior consumo de tempo. Cada algoritmo clássico disponível para implementar tais chamadas sofre de diferentes limitantes de utilização, tanto relacionados a quesitos operacionais quanto de desempenho. Entre estes é possível mencionar o funcionamento restrito à certas quantidades de processos, o crescimento linear de passos de comunicação com o aumento dos processos participantes, cópias e movimentações de memória para garantir a correta organização dos dados, e padrões de comunicação dispersos, muitos dos quais negativamente contribuem para o tempo de conclusão da operação. Todas essas características criam um ambiente em que não há um algoritmo superior em todos os casos e tal fato consequentemente implica na realização de escolhas minuciosas sobre qual opção deve ser utilizada para executar a operação. Como contribuição científica é proposto o algoritmo Stripe Parallel Binomial Trees (Sparbit), que utiliza árvores binomiais para realizar trocas de dados com tempos ótimos de latência e transferência de dados, sem restrições de uso. A proposta também mantém padrões de comunicação locais, que minimizam os atrasos decorrentes das trocas de dados por longos caminhos, permitindo a extração de mais desempenho dos sistemas atuais quando comparada com alternativas assintoticamente semelhantes. Resultados experimentais apontam que cerca de 40% das chamadas Allgather realizadas poderiam ter reduções médias de 20% a 28% no tempo de execução, com reduções máximas próximas de 74%. Para Allgatherv os resultados são altamente variáveis com base na distribuição dos tamanhos de blocos ao longo dos processos.

Palavras-chave: Alto Desempenho. Message-Passing. Comunicação Coletiva. Algoritmos Coletivos. Allgather. Allgatherv.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
ECP	Exascale Computing Project
HPC	High Performance Computing
MCA	Modular Component Architecture
MPI	Message-Passing Interface
MPIF	Message-Passing Interface Forum
NIC	Network Interface Card
NUMA	Non-Uniform Memory Access
RMA	Remote Memory Access
TCP	Transmission Control Protocol

CONTENTS

1 INTRODUCTION

There are currently a number of scientific applications that need High Performance Computing (HPC) resources to achieve the desired results in a reasonable amount of time. Examples include Computational Fluid Dynamics (CFD), Fusion Energy, Wind Farm Simulations, Earthquake Hazard Analysis, Quantum Chemistry and many more (BERNHOLDT et al., 2020; KOTHE; LEE; QUALTERS, 2019). The Exascale Computing Project (ECP) alone develops dozens of such exascale-ready applications that are vital to help research in several areas of knowledge (KOTHE; LEE; QUALTERS, 2019; MESSINA, 2017; BERNHOLDT et al., 2020). By running on massively parallel infrastructures, these applications must rely on technologies that are capable of integrating the hardware resources and cordinating data sharing, message exchanges and computing capabilities. A widely employed option to this end is the Message-Passing Interface (MPI) (GONG; HE; ZHONG, 2015; JEANNOT; SARTORI, 2020) which is used either directly by the final application or through middle ground abstraction layers in the form of more specialized libraries and frameworks.

The MPI standard defines several constructs to handle communication between concurrent processes (GONG; HE; ZHONG, 2015; JEANNOT; SARTORI, 2020). These can be generally divided in two forms of message exchanges: point to point operations, where two processes communicate individually with each other through one- or two-way data movements; and collective operations, where multiple processes contribute as data sources, sinks or both into a broader coordinated exchange (MPI Forum, 2015). Although both forms have significant usage on HPC applications, the latter is responsible for the majority of communication time spent on MPI and also accounts for the most number of calls (CHUNDURI et al., 2018). Therefore, the collective operations are considered critical to improve the performance of parallel and distributed memory systems (GONG; HE; ZHONG, 2015). On this thesis, we focus on the Allgather (`MPI_Allgather`) and Allgatherv (`MPI_Allgatherv`) many to many operations, which amongst other collectives hold a large share of utilization (LAGUNA et al., 2019) and time consumption (CHUNDURI et al., 2018) on applications.

The delay magnitude for the completion of a collective is a product of several factors, which include the underlying hardware topology (BOSILCA et al., 2017), communication protocols (CHAKRABORTY et al., 2018), network capacity (GONG; HE; ZHONG, 2015), placement of processes (KURNOSOV, 2016; ALVAREZ-LLORENTE, J. et al., 2017) and others. However, one of paramount importance is the performance of the algorithm employed to coordinate the high level inter process communication and block transferences (BENSON et al., 2003; KUMAR; SHARKAWI; JAN, 2016). On a purely theoretical regard, the community has long discovered the minimum time costs to perform this task, both in the latency term - or the required number of steps - and in the bandwidth term - or the actual data transfer time (BRUCK et al., 1997). This, in turn, led to the development of theoretically optimal algorithms which are capable of performing complete exchanges while maintaining minimum costs.

For Allgather and Allgatherv, the generally available algorithms are Ring, Neighbor Exchange, Bruck and Recursive Doubling (MPICH Team, 2019; Open MPI Team, 2020). Hereafter we refer to these algorithms as *basic* or *traditional* since they merely define simple mathematical formulas for finding the communication peers of each process, without any particular prerequisite from the infrastructure. The Ring algorithm has a linear growth of both latency and bandwidth time with the increase in the number of processes (ALVAREZ-LLORENTE, J. et al., 2017). Neighbor Exchange has the same asymptotic behaviour but with a less steep increase in time, requiring only half of the Ring's steps, with the downside of only working for even numbers of processes (CHEN et al., 2005). Recursive Doubling reaches the minimum costs by delivering the data with a logarithmic increase in latency, but only works with power of two numbers of processes (THAKUR; RABENSEIFNER; GROPP, 2005). Finally, the Bruck algorithm works for any number of processes and also has a logarithmic cost, requiring only one additional step in non power of two process numbers (BRUCK et al., 1997).

One could expect that for being optimal cost algorithms both Recursive Doubling and Bruck would be employed at all cases. Nonetheless, the logarithmic options may not yield the best performance when compared with theoretically less efficient ones as Ring (THAKUR; RABENSEIFNER; GROPP, 2005) and Neighbor Exchange (CHEN et al., 2005), and the reason for these observations lies on practical limitations that arise when the algorithms are implemented. In practice, Bruck and Recursive Doubling are only employed for reduced data sizes (MPICH Team, 2019; Open MPI Team, 2020), because although both have the smallest costs, their communication patterns are too diffuse, which severely hurts their performance with large data sizes. As the data to be transferred grows, Ring and Neighbor Exchange are employed because despite having a worse cost, their communication patterns are local and hence, less expensive to move data (THAKUR; RABENSEIFNER; GROPP, 2005). Figure 1 presents a heat map displaying the algorithm with smallest average experimental time for each combination of data size and number of processes (experiments are further detailed in Section 6.1), which portrays the aforementioned characteristic of performance degradation for logarithmic algorithms on large data sizes, where linear options achieve the best times. It is also clear how on power of two numbers of processes Recursive Doubling generally achieves better times than Bruck, while a similar pattern can be seen for even numbers of processes with Neighbor Exchange in relation to Ring. All these observations highlight the fact that there is no *silver bullet* algorithm which is the best for all scenarios and test cases.

Even with such potential for focused performance improvement, research on the development of new basic algorithms for Allgather has plummeted through the years, with a shift in focus towards proposals which are only built around the existing ones, like rank remapping techniques and hierarchical approaches, further discussed in the related work in Chapter 3. A possible reason for this decay is the fact that the theoretical minimum costs for performing the exchanges have long been discovered, as well as corresponding algorithms that reach these limits were proposed. These coupled with the natural simplicity of the methods greatly reduces

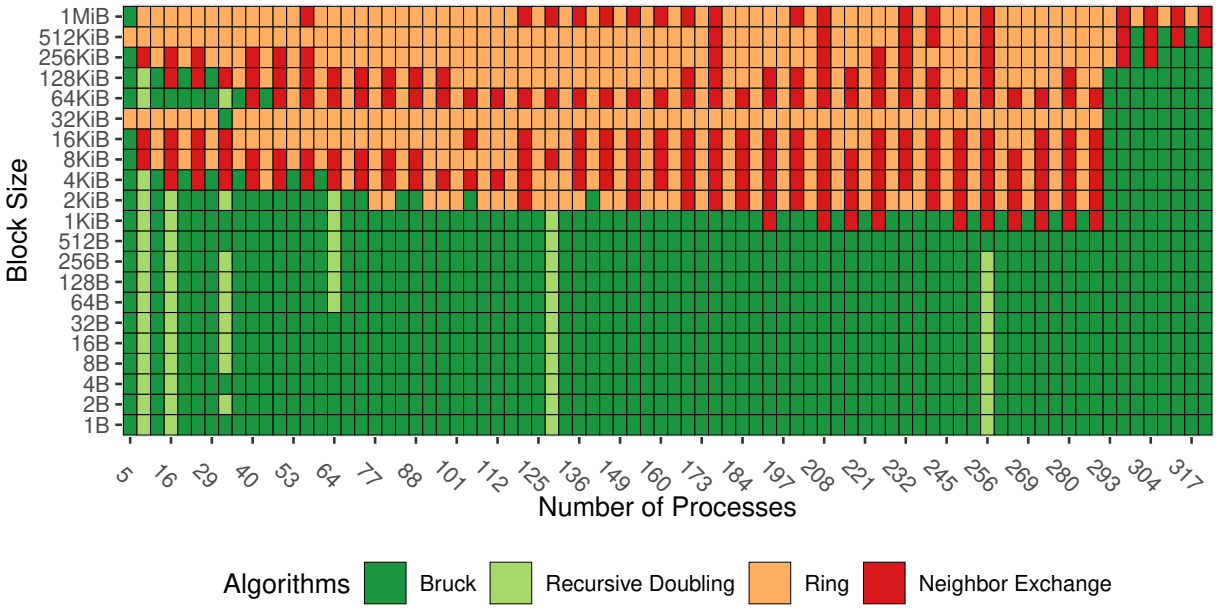


Figure 1 – Heat map portraying the algorithm with the best experimental time for each test.

the available room for improvement and consequently the number of new developments under this direction likewise. We have nonetheless discovered a locality related issue in the current collective algorithm scenario, such that although certain options provide the minimum theoretical costs, their performance is damaged by non-local communication patterns. This hindering phenomenon is so severe that it turns linear algorithms into more efficient alternatives, rendering the advantage of logarithmic number of steps virtually useless in many cases.

Following these discoveries we therefore propose the Stripe Parallel Binonimal Trees (Sparbit) basic Allgather algorithm, which has minimum latency and bandwidth time costs while still maintaining a much more local communication pattern. According to the experiments, an average of about 40% of all Allgather executions could benefit from about a 20% mean and near 74% highest improvement on execution time just by employing Sparbit. For Allgatherv the results were highly variable depending on the block size distribution employed and other factors, as further discussed in Section 6.3.

The additional performance provided by our novel algorithm, which improved upon the state of the art in several cases, will only be leveraged if an efficient algorithm selection policy is employed. This however has by itself a consistent dedicated research effort, as further discussed in Section 6.4. Therefore, we retain our focus on presenting and validating the Sparbit's proposal, while usage guidelines are the aim of other independent works.

1.1 OBJECTIVES

The main objective of this work is the development of a novel MPI Allgather and Allgatherv collective communication algorithm that has optimal latency and bandwidth costs while maintaining local communication patterns. The proposal also needs to allow easy integration into

current systems to simplify wide-spread adoption, not requiring topology or machine information, rank remapping or previous executions to guarantee performance improvements.

Specific Objectives:

- Study the potential for a novel algorithm with the aforementioned characteristics and possible relaxations.
- Formally propose an algorithm with improved locality for Allgather and Allgatherv, able to reduce the state of the art communication time of these calls.
- Develop prototypical code for the proposal, allowing the practical comparison with existing approaches.
- Perform a series of experiments on HPC environments to validate the theoretical discussion and gauge the possible performance benefits.
- Conduct a detailed theoretical analysis about the main factors which influence the performance of the algorithms and compare it with the experimental findings.
- Integrate the developed code into open source MPI implementations as Open MPI and MPICH.

1.2 ORGANIZATION

This work is organized as follows. Chapter 2 provides all the background information necessary for the establishment of the proposal, containing concepts related to the MPI standard, implementations, the existing algorithms and the problem which we aim at solving. Chapter 3 discusses the related work and the alternative solutions provided throughout the years in comparison to our novel approach. The functioning of the Sparbit algorithm – main contribution of this thesis – is showcased in detail in Chapter 4, along with its theoretical cost and practical usage considerations. In Chapter 5 we conduct a deep analysis of the relation between different algorithms and the mapping of processes to machines, outlining cases in which they are expected to be favourable for execution or not. Chapter 6 describes how the experiments for the validation of the proposal were designed and executed, including employed testbeds, benchmarks, parameters and other vital information. It also comprises a thorough analysis of the obtained empirical data, which outline Sparbit's performance in relation to other currently employed algorithms. Finally, Chapter 7 concludes this thesis and presents the future works.

2 BACKGROUND

In this Chapter the concepts over which the proposal's understanding build upon are presented . Section 2.1 provides an overview of the MPI standard, synthesized mainly from its official definition available at (MPI Forum, 2015). Section 2.2 presents and discusses some of the implementations available for MPI, extending with finer detail towards Open MPI and its architecture, since it is the choice for the aggregation and testing of the proposal. The functioning of the MPI Allgather and Allgatherv collectives is introduced in Section 2.3, while the algorithms that implement such data exchange semantics are provided in Section 2.4. Section 2.5 outlines the existing problem on the current scenario of collective algorithms and finally, Section 2.6 presents the partial considerations for this Chapter.

2.1 MESSAGE-PASSING INTERFACE

In order to be able to leverage the pool of computing resources available on supercomputers and on HPC clusters, one must have a way of coordinating all the aspects of machines working both individually and collectively. Factors like management of memory hierarchies, accelerators, Input/Output, and others which pertain to the machine domain are dealt with locally, while addressing inter node synchronization, execution workflows, data sharing and exchange is a naturally communal task. Furthermore, there must also be a way to accurately describe the application semantic and what operations should be executed on the computers to achieve the final result.

To ensure these various capabilities, one of the main models is the Message-Passing Interface standard, which defines a broad set of constructs to allow coordination and communication of processes under parallel and distributed memory environments. Its main goal is to provide a general interface for message-passing programs to exchange information. This is fundamentally supplied on the standard by the definition of calls for point to point operations, where two processes communicate with each other sending or receiving data in one or both ways; and collective communications, where a group of processes interact in routines that usually also involve exchanging data. Further, a large number of complementary functionalities have been added throughout the years, such as process topologies, information caching, neighborhood and one-sided communication, *etc*.

The history of MPI dates back to as early as 1993, with initial meetings between members of both academia and industry to start the process that would latter on give fruition to the first version of the standard on 1994. Since then it has been regularly maintained and updated by the Message-Passing Interface Forum (MPIF), organization which is responsible for its development (MPI Forum, 2015). Currently, MPI is on its 3.1 version and is the spearhead of the HPC and scientific computing endeavour, being the *de facto* standard for parallel applications on distributed memory systems (JEANNOT; SARTORI, 2020), both on supercomputers and on computing clusters (GONG; HE; ZHONG, 2015). Its use is done either directly by the final

application, through the call of the defined interface, or through middle ground abstraction layers in the form of more specialized libraries and frameworks. Examples for the former and latter are respectively NASA's Fun3D CFD simulator (BIEDRON et al., 2019) and the FFTW fast Fourier transform library (FRIGO; JOHNSON, 1999).

The MPI standard is defined for C and Fortran, however there are existing implementation bindings for other languages such as Python, Julia, R (YU, 2021) and others. The direct usage of MPI is done by means of explicitly calling its functions throughout the code and using a proper compiler wrapper. The region of the code involving its use has its start and end explicitly defined by the `MPI_Init` and `MPI_Finalize` functions, respectively. The communication processes on MPI are based on virtual channels called communicators and a process may be a part of several of them. A process receives a *rank* for every communicator that it participates, and it serves as its unique identification for receiving and sending messages on that channel. A rank is mathematically defined as an integer value ranging from zero to the amount of processes in the communicator minus one. Every form of inter process communication requires the specification of a communicator on which it will take place. Point to point operations require also the definition of a sender, destination or both, depending on which call is employed. Upon initialization, a global predefined communicator called `MPI_COMM_WORLD` is created involving all the existing processes. Then, if needed, new subset communicators can then be generated from it.

Once the initialization is completed and there is an available communicator, it is possible to perform communication operations between processes by a call to a function that represents the desired semantics. Below are present all the blocking calls that can be employed by a process in order to start some form of communication and unless explicitly noted, calls that also have a non blocking version are presented with their names appended with an asterisk – Although, on MPI they are actually prefixed with a capital letter *i* as in `MPI_Isend`. There will not be included calls that are purely local operations and others such as data size previews and probes for messages, window openings for one-sided transfers or request polling, since they only act as supporting routines for the actual communication. The semantic unit of transference for every call is a block, which is the term employed to represent a set of one or more items of a data type that gets sent or received during the communication. The block size and type are generally user defined, but in some calls the size can be predetermined by itself. The first group of calls presented is destined for point to point operations and brief definitions are provided alongside each.

- `MPI_Send*`: Sends a block to a defined destination rank. Under MPI implementation's choice the block can either be directly sent to the destination or temporarily stored on a buffer if no matching receive exists.

- `MPI_Recv*`: Receives a block from a defined source rank. Must be matched by a corresponding send to return.

- `MPI_Ssend*`: Synchronously sends a block to a defined destination rank without intermediate buffering. For the call to return there must be a match with a corresponding send and some data already transferred.
- `MPI_Bsend*`: Buffers the block in auxiliary memory space before sending it to the destination rank if no matching receive exists. May return before a corresponding receive is matched.
- `MPI_Rsend*`: Sends the block to the destination rank only if there is a corresponding receive matched, otherwise the call is invalid.
- `MPI_Sendrecv`: Simultaneously sends and receives a block. The destinations for sending and receiving may differ. Can possibly be matched with regular send and receive operations on the target nodes.
- `MPI_Sendrecv_replace`: Same semantic as in `MPI_Sendrecv`, however the block sent is overwritten by the received one.

The second group of calls is destined for collective communication operations. On these cases each involved process must have issued the call for it to be able start and if there is data transfers, each process acts as a block destination, source or both into a broad exchange. Certain calls have versions which allow processes to send blocks with sizes different from other peers, while other versions allow both blocks with different sizes and data types. These are respectively called vector and struct versions of a call and if they exist for a given case, a *v* for vector or a *w* for struct are appended between braces. For this group all calls have non blocking versions and thus the previously employed asterisk is omitted.

- `MPI_Bcast`: One to all operation where a process broadcasts the same block to all the other participants.
- `MPI_Scatter[v]`: One to all operation where a process sends a different block to each of the other participants.
- `MPI_Gather[v]`: All to one operation where a process receives a different block from each of the other participants.
- `MPI_Barrier`: All to all operation where each process stops on this call until every other one has reached it. Then, all continue the execution together.
- `MPI_Reduce`: The blocks from all processes are combined through a mathematical operation – process also named reduction – and the result is stored in one process (All to one operation).
- `MPI_Scan`: Each process receives the reduction of its block combined with the ones from all processes with ranks smaller than his own (operation also named prefix reduction).

- `MPI_Exscan`: Exclusive scan operation, similar to the `MPI_Scan` call but excludes each processes' own block from the reduction operation.
- `MPI_Allreduce`: Similar to the `MPI_Reduce` call, however all processes receive the result of the reduction operation.
- `MPI_Allgather[v]`: Semantic union of the `MPI_Bcast` and `MPI_Gather` calls on every peer. Each process both sends its only block to all other processes, and receives their individual blocks in return.
- `MPI_Neighbor_allgather[v]`: Similar to `MPI_Allgather` but executing only with neighbor processes.
- `MPI_Alltoall[vw]`: Semantic union of the `MPI_Scatter` and `MPI_Gather` calls on every peer. Each process sends a different block to each other participant, and receives a different unique block from each one of them.
- `MPI_Neighbor_alltoall[vw]`: Similar to `MPI_Alltoall` but executing only with neighbor processes.
- `MPI_Reduce_scatter_block`: Semantic sequence of `MPI_Reduce` and `MPI_Scatter`. Performs a reduction operation and then scatters the result to all participants on equal sized blocks.
- `MPI_Reduce_scatter`: Semantic sequence of `MPI_Reduce` and `MPI_Scatterv`. Performs a reduction operation and then scatters the result to all participants on possibly differently sized blocks.

The last group of standardized communication calls are the ones destined for Remote Memory Access (RMA) operations as presented below. For these calls, the non blocking versions are prefixed with the capital letter *r*, as in `MPI_Rget`, instead of *i*.

- `MPI_Put*`: Writes a block of data to a possibly remote memory window.
- `MPI_Get*`: Reads a block of data from a possibly remote memory window.
- `MPI_Accumulate*`: Combines a local block of data with the one on a possibly remote memory window through a mathematical operation.
- `MPI_Get_accumulate*`: Similar to the `MPI_Accumulate` call, however remote data is sent back to the calling process before the accumulate operation is performed on the target block.
- `MPI_Fetch_and_op`: Similar to the `MPI_Accumulate` call, but operates only with blocks of unitary size.

- `MPI_Compare_and_swap`: Replaces the data on a possibly remote memory window with a third value if the target and origin blocks are equal. Also operates only with unitary sized blocks.

All these functions coupled provide the communication tools for MPI processes to interact and exchange information vital to the execution flow. By means of their usage, large computational tasks can be divided into smaller segments and individually handled to distributed nodes, which together are able to cooperate for solving the whole problem.

2.2 MPI IMPLEMENTATIONS

The MPI standard on itself is a document that defines the interface, function signatures and other requirements that must be met in order to provide message exchanging capabilities. The Message-Passing Interface Forum does not charge itself with the task of implementing the MPI interface, but actually transfers this duty to any third-party organization interested on its development (MPI Forum, 2015). Therefore, the standard does not provide or sanction any particular real world implementation of the library as official. This throughout the years stemmed the creation of several distinct implementations, each with possibly different goals like portability for MPICH (MPICH Team, 2020), flexibility for Open MPI (GABRIEL et al., 2004), fault tolerance for FT-MPI (FAGG; DONGARRA, 2000), and many others.

MPI implementations can differ in various factors, such as the way the applications are launched and run, their internal functioning, languages to which they are available, licences on which they are distributed and the level of compliance with the standard. In this sense, an implementation is fully complaint when it follows all the specifications for the interface defined in the standard. An implementation can also be partially compliant or have full compliance only with some previous version. Information regarding implementation popularity can be obtained from the TOP500 list (DONGARRA; LUSZCZEK, 2011), which presents a biannual list of the fastest supercomputers in the world along with some of their architectural and software stack information. Taking the November 2020 list (TOP500, 2020), the most popular proprietary implementations are IBM Spectrum MPI (IBM Corporation, 2016), Cray-MPI and Intel MPI (Intel, 2021), while the open source implementations are Open MPI (GABRIEL et al., 2004) and MPICH (MPICH Team, 2020). A similar trend is also expected to be valid for HPC clusters.

Due to this popularity information, all the source code development as well as the experiments for the proposal outlined on this work were done employing Open MPI. This choice is also backed by its flexibility and easiness for integration and usage of new components. Additionally, the fact that Open MPI is an open source tool allows the visualization and understanding of its internal structures, which are vital to the fine-tuned execution and validity of experiments as well as future aggregation of the proposed algorithm. Further details of Open MPI and its internal functioning are presented in Section 2.2.1.

2.2.1 Open MPI and the Modular Component Architecture

As mentioned in Section 2.2, Open MPI is the implementation of choice for the development of the proposal and further execution of experiments. The choice is made firstly by its popularity and prevalence on the HPC and supercomputing environments, and secondly by its easiness for integrating new modules, which allow the experiments to be run on a vanilla pre compiled distribution that only loads the binaries related to the proposal on runtime, thus eliminating the need for compiling the whole library from the source with the modifications.

Open MPI was proposed in 2004 with intents to address the incompatibility scenario for the existing MPI implementations at the time. Such a problem arose due to the focus of implementations on solving specific research problems or aspects in need, which led to multiple options being simultaneously employed for different goals (GABRIEL et al., 2004). Open MPI aims at creating a general purpose and modular implementation that is highly extensible, allowing new functionality to be added via its component architecture. By these means, it serves as a platform for the development of research, experimentation, distribution of software products and also naturally as a scalable high performance parallel computing library.

The core of Open MPI, and what actually supports all of its features, is the Modular Component Architecture (MCA). It acts as a management structure that provides the necessary functions for the existence of other more specialized frameworks. The MCA is also responsible for down streaming user defined parameters to the frameworks for proper handling (GABRIEL et al., 2004). Each of the frameworks defines an interface for providing some given functionality to the upper layers and to other parts of the implementation. Examples of such include point to point messaging, collective communications, memory allocation, I/O and several others.

The implementations of such functional interfaces are called *Components* and can be statically linked with MPI on compile time, or be shipped as dynamic libraries to be loaded at runtime. Multiple of such components can coexist inside of Open MPI and their usage is defined at runtime, thus the framework and components relation allows the shipping of a standard vanilla version of a functionality that can be latter replaced by a component which is able to leverage a particular available hardware technology or that is better suited for one's particular environment (Open MPI Team, 2021). This ensures a high level of flexibility and customization of the MPI implementation, since practically all functions performed by the library can be manually selected and fine tuned, as well as if needed, new capabilities for existing functions can be developed and distributed by third-party organizations.

The last structural piece is the *module*, which represents an instantiation of a component to be used on a certain context. In this sense, a module structure containing all the entry points required for executing its component's functionality is created and returned to the parent caller. Certain modules might need to employ the functionality of other components and hence instantiate and use a corresponding module instance as a part of their operation. Multiple modules can be employed simultaneously either in a higher level or deep inside the library.

On the former side, if multiple communicators exist for example, each will have an attached collective communication module and thus the same collective component might have multiple instantiated modules for different communicators or each communicator may instantiate a different module depending on its characteristics. For the latter case, if for example a node has multiple network interfaces, the module for point to point messaging will instantiate an underlying different module for each of the hardware units, in order to be able to split messages over all of them or to select a more appropriate transport given the message parameters.

The general hierarchy and workflow between architectural parts defines that MCA base functionality will be employed by framework functions to find and load components, added either during the compile time or available in a default search directory (GABRIEL et al., 2004). Modules of such components are then created using the exposed functions and attached or returned to the correct requiring context (Open MPI Team, 2020).

2.3 MPI ALLGATHER CALL

Among all the collective operations available on the MPI standard, we direct our focus towards the Allgather and Allgatherv calls, which hold large shares of utilization (LAGUNA et al., 2019) and time consumption on applications (CHUNDURI et al., 2018). The `MPI_Allgather` is a many to many collective operation where each process sends a fixed size block of data to all other processes participating on the call and receives their individual blocks as well. The block sent by a process has one or more items of a chosen datatype and both the size and type signature of the block must be equal to all destinations.

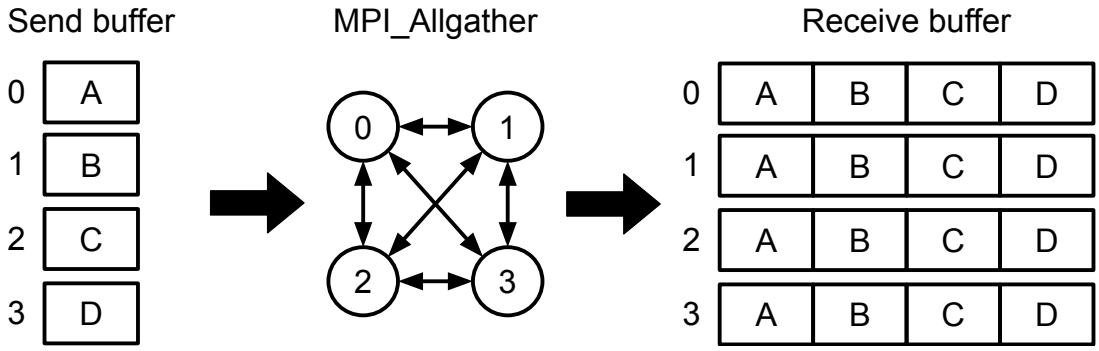


Figure 2 – Buffers' organization and abstract communication pattern of the MPI Allgather call.

As shown on Figure 2, after the execution of the Allgather operation the block sent by the jth process is received by all others and is placed on the jth position of their receive buffer (MPI Forum, 2015). Each process sends the same one block to all its peers and receives the number of processes minus 1 blocks in return. The resulting receive buffer is equal for all the processes. This call also has a vector (MPI_Allgatherv) version, which maintains the same general semantics but allows each process to send a block with arbitrary size. Figures 2 and 3 show the execution and buffer configurations with 4 participating processes for the regular and vector forms of the

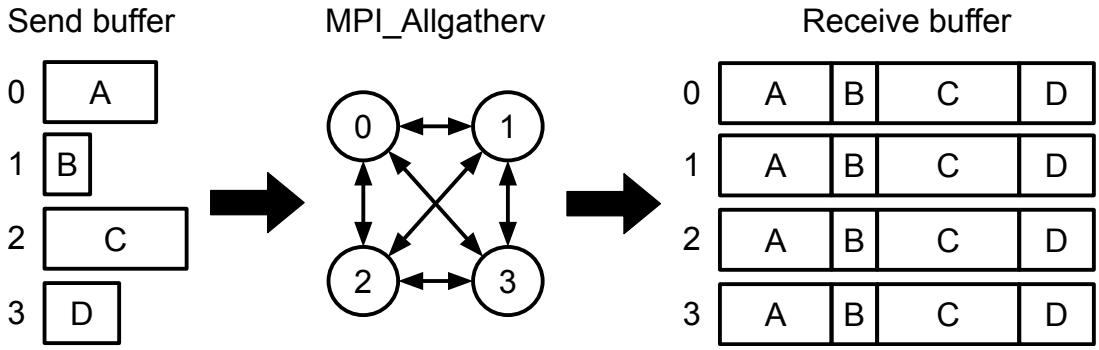


Figure 3 – Buffers' organization and abstract communication pattern of the MPI vector Allgather call.

Allgather operation, as described above. The varied square dimensions on Figure 3 represent different block sizes.

2.4 ALGORITHMS

Although each collective operation available on MPI has a very well defined data exchange semantic, it may not necessarily be the best way of actually performing the transferences on real systems. Since the formal definition of a collective is guided via a final state oriented description and no strictly required way of moving data is instituted on the standard, diverse techniques such as process cooperation, concurrent messaging and data buffering can be employed to make exchanges faster. This allows the existence of different algorithms that implement these calls and devise particular ways of delivering the blocks, being more or less efficient depending on the system, data size, number of participating processes and other factors. In this Section, we delve into the available algorithms employed for both the regular and vector MPI Allgather calls and, while some might also be extended for collectives with distinct communication patterns, our focus is solely on their usage for these exchange semantics. The presented options are gathered from the literature and from source code analysis of both MPICH 3.3.2 and Open MPI 4.0.3, which in the moment of writing this work are the most recent distribution versions of these implementations.

In order to accurately compare the algorithms for the Allgather operations, a cost model is necessary. To this end we present each algorithm's complexity as cost in time following the Hockney model (HOCKNEY, 1994). With this approach the cost of sending a message is given by a latency or start-up term α and a bandwidth or transference cost per byte term β. We also suppose that the nodes have a single communication port, which is generally the case in many modern systems. According to this, a process can concurrently send and receive data, but only one message can be transferred at a time on each direction. Since all the presented algorithms on their pure theoretical form essentially implement communication patterns built of individual point to point messages, the total cost of each algorithm following the Hockney model is given by the time sum of all the building block messages employed during its execution. This

also naturally implies that if n equally sized messages are sent over n distinct channels on any given step, and both their start and ending points in time are equal, then the total time for the transfer operation is the same for any number of concurrent messages n, with $n \geq 1$ and $n \in \mathbb{N}$ – considering, as the model suggests, that the channel speeds are equal.

2.4.1 Ring

The Ring algorithm performs the block exchange by shifting data around on a ring communication topology. In the first step, a process with rank r will send its original block to the process with rank $r+1$ and receive the original block of the process with rank $r-1$ (wrapping around if a destination or origin is out of bounds). From the second until the last step, this communication pattern is repeated but instead of each process sending its original block it sends the block it has received in the previous step (ALVAREZ-LLORENTE, J. et al., 2017). Therefore, for the data of any given process to traverse the whole topology and reach its left neighbor $r-1$ there will be needed $p-1$ communication rounds. In order to provide the total cost in time, the number of processes involved in the algorithm is hereafter represented by p, while m represents the total amount of data that a process must have at the end of the operation. In this sense, each process has to receive a block with size $\frac{m}{p}$ from all the other $p-1$ participants as well as send its own. This determines the bandwidth cost and can not be further reduced, since at least this much data must be exchanged. As all sends happen in parallel, according to the previous discussion we can account only the time to send one block per step. Since each send has both a fixed α start cost and a variable bandwidth β term dependent on the message size, the total cost is given by $C_{ring} = (p-1)(\alpha + \frac{m}{p}\beta)$.

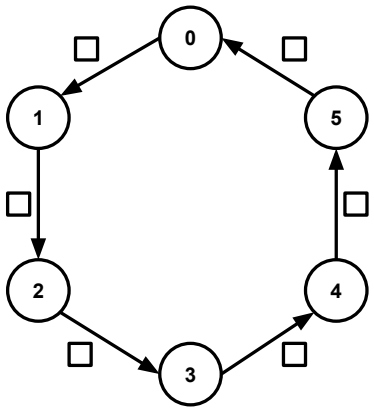
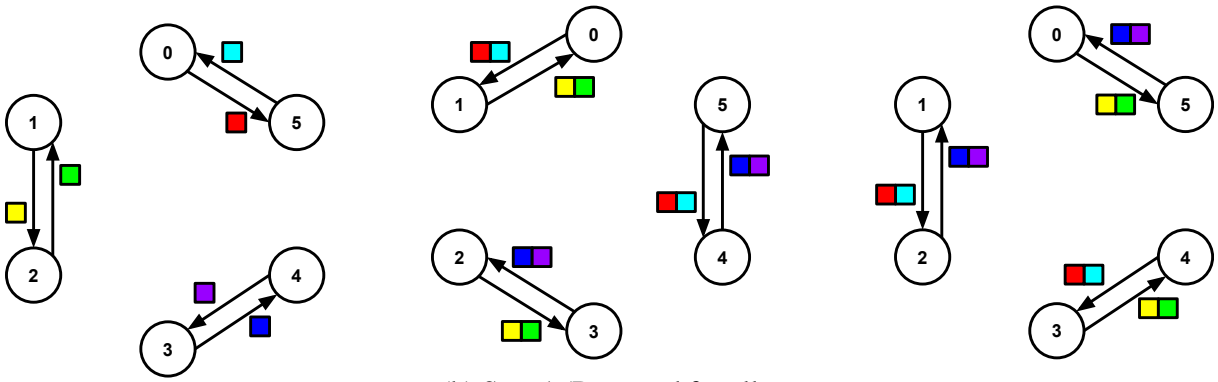


Figure 4 – Communication pattern of the Ring algorithm on all steps for 6 processes.

Figure 4 displays a diagram of the ring's functioning for 6 participants. As can be seen, its execution is fairly simple and processes merely forward data to their fixed destinations, either their own block on the first step, or the previously received one, on others. The ring algorithm is employed both on MPICH and Open MPI for Allgather and Allgatherv on various cases (MPICH Team, 2019; Open MPI Team, 2020).

2.4.2 Neighbor Exchange

The Neighbor Exchange is another linear complexity algorithm, however its latency growth is smaller than the Ring by half. The main driving idea behind this algorithm is to make use of pairwise exchanges to better leverage TCP communication. As pointed by (BENSON et al., 2003), upon usage of pairwise communication, data can be piggybacked on ACK packages sent as replies, thus reducing the number of total packages sent across the network and potentially speeding up the data exchange. The algorithm employs this fact by making all the communications pairwise, so that in any step a process both sends and receives data to and from either its left or right neighbor in an alternating fashion. Therefore, on every step $0 \leq s < \frac{p}{2}$, an even process r will send data to its neighbor $r + (-1)^s$ while an odd process r' will send data to its neighbor $r' - (-1)^s$, both wrapping around in the case of out of bounds destinations. On the first step each process sends its original block, on the second step it sends both the original block and the one received on the previous step. From the third until the last step, each process sends the two blocks received in the previous one (CHEN et al., 2005). As two blocks are sent in all steps except the first, for the complete exchange to happen the algorithm takes $\frac{p}{2}$ rounds and its cost is given by $C_{ne} = \frac{p\alpha}{2} + (p-1)\frac{m}{p}\beta$. The downside of this algorithm is that its pairwise nature of communication implies that it can only work with an even number of processes.



(a) Step 0.

(b) Step 1 (Repeated for all odd steps).

(c) Step 2 (Repeated for all even steps).

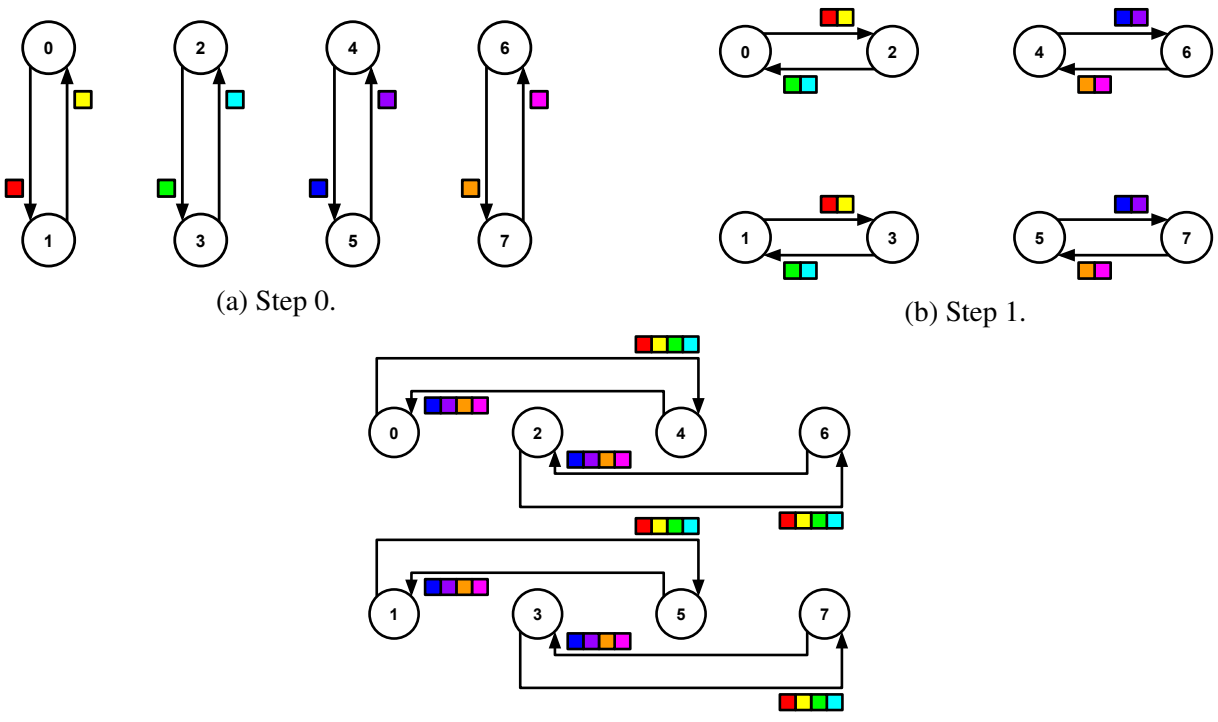
Figure 5 – Communication pattern of the Neighbor Exchange algorithm on different steps for 6 processes.

Figure 5 shows the functioning and exchange pattern of the Neighbor Exchange algorithm for 6 processes. On the first step, processes exchange only their original block with one of their predefined peers, as portrayed on Figure 5a. On the second step, presented on Figure 5b, every process then alternates its neighbor and sends both its original block and the one received in the previous step. From the third until last steps, neighbors and communication patterns cycle between the ones presented on Figures 5b and 5c, starting from the latter, with the two blocks received on the previous step being sent on the current one. Since on this case with 6 processes only 3 steps are needed, Figure 5 happens to display the complete execution of the algorithm. Blocks were colored so that each one and their original owner can be distinguished. On each send

blocks are also displayed in a sorted manner, with the ones originating from smaller ranks placed first from left to right. These two visual characteristics allow the verification that each process receives all the differently colored blocks throughout the execution. The neighbor exchange is not employed on MPICH (MPICH Team, 2019), but is utilized for several cases on Open MPI (Open MPI Team, 2020), both for Allgather and Allgatherv.

2.4.3 Recursive Doubling

The third available algorithm is the Recursive Doubling, which performs the complete exchange in a logarithmic amount of steps. The main difference of this algorithm in relation to the previous ones is that instead of maintaining a fixed distance offset and sending the same amount of data, it doubles both the distance of the communication and the data being sent. In every step $0 \leq s < log_2 p$ a process with rank r will exchange data with a process that has rank $r \oplus 2^s$, where $\oplus$ represents the exclusive *or* operation and serves to interlock the pairwise destination calculation. A process will send all the data received so far along with its original block on every step, which makes the total data being send double at every new communication round (MIRSADEGHI; AFSAHI, 2016).



(a) Step 0.

(b) Step 1.

(c) Step 2.

Figure 6 – Communication pattern of the Recursive Doubling algorithm on different steps for 8 processes.

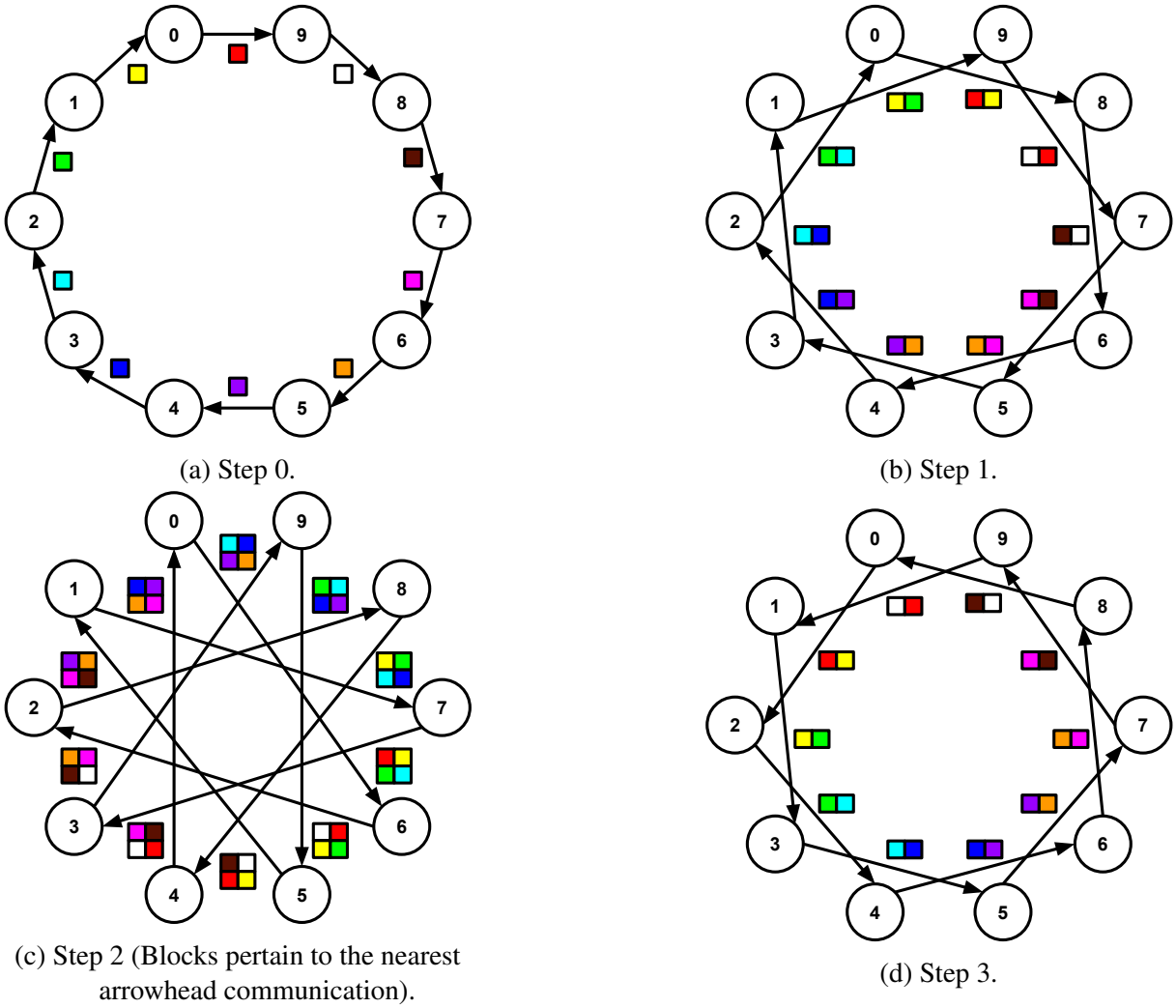
From a behaviour perspective, in the first step all even processes exchange data with their immediate superior odd neighbors and from the second to the last steps, even and odd processes communicate only with peers of the same parity and through increasingly longer distances. From this, it is then natural that any process will take $log_2 p$ steps for gathering

all the blocks. As all communications happen in parallel, the cost of this algorithm is given by $C_{rd} = (log_2 p)\alpha + (p-1)\frac{m}{p}\beta$. The operation of the Recursive Doubling is limited only to numbers of processes that are powers of two and thus this is the only case where it is employed on MPICH (MPICH Team, 2019) for Allgather and Allgatherv, as well as on Open MPI (Open MPI Team, 2020) for Allgather. There are existing versions of this algorithm that work for numbers of processes that are not powers of two, however they do not attain to the same optimal complexity.

Figure 6 demonstrates the functioning of the Recursive Doubling algorithm for 8 processes. Since the communication on all steps happens in a pairwise fashion, on the first one – portrayed on Figure 6a – each even ranked process exchanges its original block with an odd one and vice-versa. On the second step, communication distances double and each process exchanges both its original block and the one received on the previous step with a process away by two ranks, as visible on Figure 6b. On the final step, distances once again double as does the exchanged data size. Now each process sends its original block, the one received on the first step and the two received on the second to a process away by four ranks. Figure 6 once again displays blocks colored and sorted on the sends according to their original owner's rank, allowing the verification that each of the processes receives all the blocks.

2.4.4 Bruck

The Bruck was created in order to provide an algorithm of logarithmic time cost that works for any given number of processes. It is very similar to the Recursive Doubling on its core, in the sense that it doubles both the distance of the communication and the data send throughout the process. The main difference in the Bruck is that it abandons the pairwise exchange and employs a different destination calculation. Hence, on every step $0 \leq s < \lfloor log_2 p \rfloor$, a process r will send all the data it has received to the process with rank $r - 2^s$ and receive new data from the process with rank $r + 2^s$. If the number of processes is not a power of two, then an additional step is needed where each process sends only the first $p - 2^{\lfloor log_2 p \rfloor}$ blocks in the receive buffer. To guarantee that only contiguous chunks of memory will be read, and therefore only one send will be needed per process per step, a memory shift before the algorithm's execution is employed to put each process' original block in the start of the receive buffer. The choice of using the minus in the destination calculation is made to ensure that the order of blocks in the final buffer will be ascending. As an initial shift was done and at the end most of the blocks are out of their proper places in many processes, an additional (un) shift operation is needed to finalize the operation (BRUCK et al., 1997). The same behaviour observation made on the Recursive Doubling applies to the Bruck, however as it does not use pairwise exchanges and the direction of sending remains the same, its execution can be seen as continuous expanding rings of data shifting. The cost of the Bruck is given by $C_{bruck} = \lceil log_2 p \rceil \alpha + (p-1)\frac{m}{p}\beta$ and it is employed both on MPICH (MPICH Team, 2019) and Open MPI (Open MPI Team, 2020) for various scenarios on Allgather and Allgatherv.



(a) Step 0.

(b) Step 1.

(c) Step 2 (Blocks pertain to the nearest arrowhead communication).

(d) Step 3.

Figure 7 – Communication pattern of the Bruck algorithm on different steps for 10 processes.

Figure 7 outlines the execution of the Bruck algorithm for 10 processes. On the first three steps its working is very similar to the Recursive Doubling as distances and data sizes sent double. The difference is that instead of performing a pairwise exchange, each process sends and receives the blocks to and from different peers. Along these steps, the semantic of block exchange is also similar to the Recursive Doubling and on the first step each process sends its own block (Figure 7a), on the second it sends both its own and the one received on the previous step (Figure 7b), while on the third it sends its own, the one received on the first step and the two received on the second (Figure 7c). If the number of processes was a power of two, then this would be the complete behaviour of the algorithm. Since however, the amount of processes is not a power of two, then a last step is performed (Figure 7d) again with doubled distances but sending only the $p - 2^{\lfloor log_2 p \rfloor}$ first blocks on the receive buffer. Figure 7 shows blocks colored for identification and sends sorted according to each processes' buffer organization, which always starts with its own block and grows sequentially in ascending and cyclic order (wrapping around) on the rank dimension. These visual information allow the verification that each process receives all of its peers' blocks.

2.5 PROBLEM DEFINITION

So far while presenting the algorithms we have only discussed usage limitations that are inherent to the architecture of each alternative itself. However, in practice the algorithms also have usage limitations that stem from performance issues, and these tend to be side effects of the inescapable relation between the way the algorithm works and the supporting technologies. The formal definitions of the algorithms generally assume equal communication costs among all peers, but computing clusters and supercomputers often employ hierarchical network topologies (KURNOSOV, 2016) such as Fat-trees and Dragonflies. On these networks the cost of performing communication between two nodes is highly dependent on their physical location (ALVAREZ-LLORENTE, J. et al., 2017), so the further away they are, the longer are the physical paths that connect them and therefore the higher the latency (KIM et al., 2008; AL-FARES; LOUKISSAS; VAHDAT, 2008). From a bandwidth perspective, the further away two nodes are the higher is the chance that their communication will cross the core of the network, whose bandwidth is more expensive and supports less saturation than the edge (WANG et al., 2018), possibly leading to slowdowns or contentions. Additional factors like the need for memory shifts and copies as well as asymmetrical or diffuse communications can further hurt an algorithm's performance (THAKUR; RABENSEIFNER; GROPP, 2005). Therefore, with a higher locality of the communication and reduced need for memory juggles, the higher the potential performance of an algorithm.

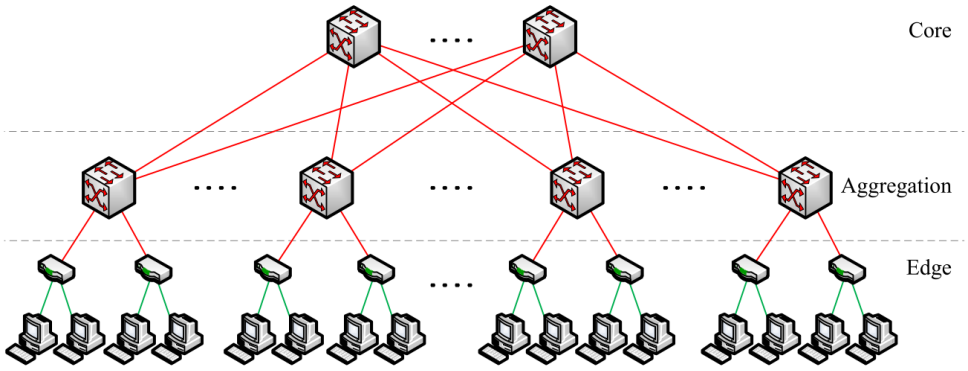


Figure 8 – Diagram of a Fat-tree topology. Extracted from (AL-FARES; LOUKISSAS; VAHDAT, 2008).

As exemplification of such network related problems, Figure 8 shows the topology of a Fat-tree, which is divided in three layers: edge (or periphery), aggregation and core. The edge layer comprises the hosts over which processes are mapped and the edge switches, each of the latter tasked with connecting several machines to form a local communication group. The aggregation layer comprises intermediary switches tasked with linking multiple groups of hosts, which is done by the integration of several different edge switches. Finally, the core layer comprises the core switches, which provide multiple interchangeable communication routes among intermediary switches, allowing the hosts from all groups to exchange information with each other. The bandwidth capacity of switches usually rises when moving from the edge to core

layers. This is done either by actually having more powerful hardware or by coupling multiple switches and dedicating more physical links to every path. All these characteristics remark the hierarchical nature of this topology and allow the Fat-tree to be an extremely scalable network employed in many HPC environments. Even with such positive features, it is already visible from its organization that the latency of communication between two hosts will vary depending on whether they are on the same edge, aggregation or core switch. When exploring the bandwidth saturation, Figure 9 shows a plot extracted from (WANG et al., 2018) that portrays the results of experiments where traffic was injected on the edge layer while the amount supported on the upper ones was recorded until congestion. It shows that the core can withstand only about 65% of the traffic supported on the periphery, highlighting the greater care required when employing its bandwidth.

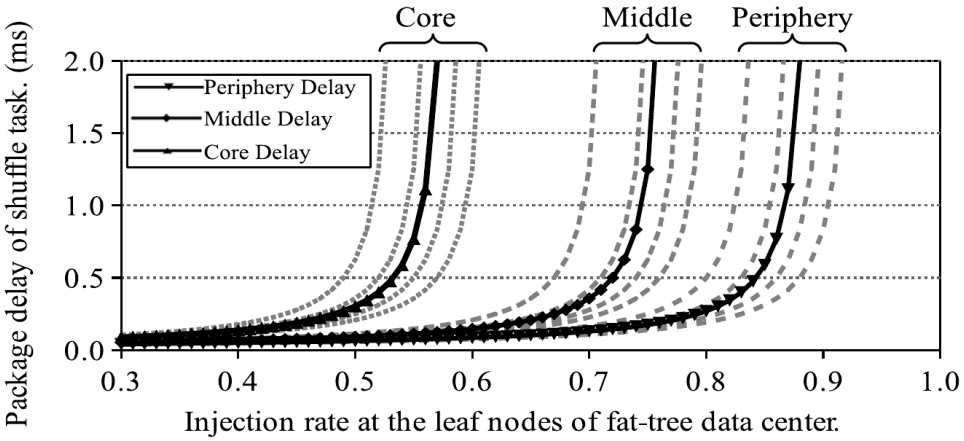


Figure 9 – Traffic limit on different layers of a Fat-tree network. Extracted from (WANG et al., 2018).

According to the cost formulation established in Section 2.4, the two main variable aspects that influence an algorithm's performance on a system are the coefficients for its cost terms, namely the number of involved processes and the data size to be transferred. These coupled with the more practical and static aforementioned factors define how well each algorithm performs on specific scenarios, thus leading them to have optimal operational areas where they perform their best and usually surpass other alternatives. In this sense, although Recursive Doubling and Bruck have a smaller number of steps - which is interesting for latency-sensitive systems and messages - they naturally communicate over longer paths on the network, implying in higher time costs specially for larger messages. Additionally, Bruck also needs memory permutations and copies that further deteriorate its performance with the increase in data size. On the other hand, Ring and Neighbor Exchange have linear latency costs - resulting in poor performances for small messages, whose bandwidth cost is very low - but have local communication patterns always with the same neighbors, which implies in reduced additional transfer costs with the growth of message size (THAKUR; RABENSEIFNER; GROPP, 2005).

Table 1 summarizes the main discussed aspects of each algorithm, presenting their cost, communication pattern and usage restrictions. We further remark that the communication pattern

is here presented in terms of the rank dimension and as so, will be reflected to the physical communication pattern with varying degrees of fidelity, depending on the mapping strategy employed – discussion detailed in Chapter 5.

Algorithm	Cost	Comunication	Use restrictions
Ring	$(p-1)\alpha + (p-1)(m/p)\beta$	Local	None
Neighbor Exchange	$\frac{p\alpha}{2} + (p-1)(m/p)\beta$	Local Pairwise	Even p
Recursive Doubling	$(log_2 p)\alpha + (p-1)(m/p)\beta$	Non-local pairwise	Power of 2 p
Bruck	$\lceil log_2 p \rceil \alpha + (p-1)(m/p)\beta$	Non-local	None

Table 1 – Cost and characteristics of current Allgather algorithms.

The current scenario outlines a conflict between the theoretical quality of the algorithms and their actual efficiency: so far, the performance of logarithmic algorithms is degraded by their non-local communication patterns to such an extent that they achieve worse times than linear options in many cases. This practical caveat prevents such class of algorithms from reaching their potential reduced costs of time, limiting the overall communication and application performances. Therefore, the core concept for solving this problem is to somehow make these exchanges more local while still maintaining the logarithmic time cost.

2.6 CONSIDERATIONS

This Chapter has provided a comprehensive set of information regarding the current scenario of MPI collective communication, which serve as base for the development of the proposal. The MPI is a multi-decade old standard that defines how messages should be exchanged among processes executing in parallel. Initially, a brief history of its creation and development was presented followed by summarized explanations of all the communication primitives available on the standard, extracted from its official definition. Since MPI is on its core just the standard definition of the interface functions that peers might use to communicate, there is no official implementation. This allows the existence of several options, each with a potentially different focus and level of compliance with the standard. These concepts are introduced before presenting a deeper explanation of the internal workings of Open MPI and its Modular Component Architecture, which receive additional detail since this is the implementation of choice for integration and experimentation with the proposal.

The Allgather block exchange semantic is presented next, defining an all to all exchange where each process sends a block to all other participants and receives their individual blocks in return. This call also also has the Allgatherv variant, which allows a process to send a block with a different size than the one from its peers. For these operations to be executed efficiently, there is the need for algorithms that coordinate the exchanges with the best possible performance on each case. We proceeded to present multiple such algorithms that are employed on MPI implementations: Ring and Neighbor Exchange are linear cost options utilized for medium or large block sizes, because although their complexities are not the best available, they have local

communication patterns well-suited for large messages; On the other hand, Recursive doubling and Bruck are both logarithmic cost algorithms that are employed mainly for small or medium block sizes, since even having optimal complexity their performance is worse in many cases due to diffuse communication patterns. Stemming from the characteristics of the algorithms itself we have stated the problem, outlining the contradiction between the theoretical cost of logarithmic algorithms and their impossibility of achieving the best times in many cases due to practical limitations, mainly non-local communication routines.

3 RELATED WORK

In this Chapter are presented the works which in some form relate to Sparbit, divided into three major groups. The first encompasses existing flat algorithms which are available for implementing the MPI Allgather or Allgatherv calls, focusing on achieving greater communication locality, saving bandwidth or diminishing the overall operation time. The second presents hierarchical algorithmic approaches for implementing these collectives, which allow additional performance improvements and greater computation and communication overlap potential. The third group represents another form of augmenting the communication locality, which is the reordering of ranks or the remapping of the processes to better suite the physical infrastructure.

The works discussed throughout this Chapter were gathered from distinct academic search engines, focusing strictly on intracommunicator Allgather or Allgatherv algorithms as well as other proposals to increase their locality or improve overall performance. According to these guidelines, works that could not be employed on the aforementioned calls or focused only on other collectives were discarded. As a publication date exclusion limit was not explicitly defined, works greatly vary on age. This outlines the decay pattern described in the discussion of Chapter 1, with directly focused algorithmic approaches being in general older than the rank remapping techniques, which are complementary to the algorithms. The next Sections discuss in detail each of the related groups of research.

3.1 ALLGATHER ALGORITHMS

The first set of related work comprises the already presented and compared classic Allgather algorithms (Section 2.4), namely Bruck (BRUCK et al., 1997), Neighbor Exchange (CHEN et al., 2005), Ring and Recursive Doubling. Regarding improvements to such foundational algorithms, the early work of (THAKUR; RABENSEIFNER; GROPP, 2005) proposes versions of the Recursive doubling and Recursive halving (destined for the MPI All Reduce operation) that work for any number of processes. For these versions the complexity, although still logarithmic and thus asymptotically equal, has twice the number of steps. Experimental results surprisingly show that such algorithms have the highest bandwidth for small number of processes when compared to the Ring, but it consistently degrades as participants rise. (ZHOU et al., 2015) proposes a bandwidth saving optimization for the ring algorithm, which is however only applicable after the execution of a binomial tree and is destined for the Broadcast operation, thus not directly employable for the Allgather call. On a similar direction, (TRäFF et al., 2008) proposes a pipelined version of the ring algorithm focused on the vector Allgather call. On this operation, the time for conclusion is mainly determined by the cost for the largest block to traverse the ring. Therefore, dividing all the blocks into b smaller chunks allows a more continuous transference of the data and alleviates the exchange size disparities. The cost of the algorithm is then changed from $p-1$ to $b-1$ on number of steps and the optimal chunk size must be defined according to a set of parameters. Experiments show great improvements over

the regular ring specially as the data size grows.

The work of (MA et al., 2011) is an algorithmic approach that employs the idea of steering the communication pattern to a more suitable form for the physical hierarchy. This proposal is similar in a way to the techniques presented in Section 3.3, however instead of remapping the ranks, they propose a change on the destination of each rank's messages. Therefore, given any initial distribution of processes over a node, the algorithm is able to create for all peers an organization of the message exchanges that crosses the minimum expansive channels or Non-Uniform Memory Access (NUMA) domains as possible. For the Allgather call the proposal shows great bandwidth stability over different mapping strategies in comparison to the regular Open MPI implementation, nonetheless overall improvements are limited. Besides employing a linear cost algorithm and focusing on inner node mappings, there is the need for gathering detailed machine hierarchy information and constructing the communication topology. In comparison to this and to the previous optimizations for classical algorithms, Sparbit is a logarithmic cost option with optimal bandwidth that works for all the available Allgather calls with no need for additional information.

There are also proposals which rely or employ hardware features to improve the performance of the collective operations. The work of (ARAP et al., 2015), for example, presents an adaptive version of the Recursive doubling algorithm based on Network Interface Card (NIC) offloading and employing multicast aided transferences. The algorithm is based on the idea that processes hardly arrive at the step's synchronization points at the same time, and thus waits can create overheads as well as unnecessary message forwarding. The proposal employs a tagging mechanism to indicate a message's originating stage relative to the algorithm and to decide upon the next actions. The presented empirical evaluation shows that the proposed algorithm can produce up to a 26% improvement in relation to the regular Recursive doubling.

Inozemtsev e Afsahi (2012) also proposed a hardware focused solution for improving the Allgather operation. Their approach is based on the Mellanox InfiniBand CORE-*Direct* technology to enable offloading of communication operations to the network interface, allowing greater computation and communication overlap and accelerating the call. The offloading process is done by posting a set of predefined send and receive operations that describe the collective semantic of an algorithm to the network adapter, which later on executes and manages the communication by itself, freeing the CPU. Offloaded flat and hierarchical versions of Bruck, Ring and Standard exchange (k-port Recursive doubling) algorithms are proposed and compared against regular versions on the MVAPICH MPI implementation. Practical results show that the offloaded versions provided consistent and significant improvement on the capacity of overlapping computation and communication, reaching near 100% for certain message sizes. Latency was also improved but on much smaller scales and specially on the hierarchical alternatives.

Another hardware aided work is the one from (TRäFF; HUNOLD, 2020), which presents a series of performance guidelines (collectives built using other collectives) for implementing collective calls on multi-rail, or multi-lane, communication systems. On such infrastructures

there are multiple communication paths over which two distinct processors can communicate independently and simultaneously. Thus a larger bandwidth is expected to be available and the work leverage it by slicing the processes into multiple inter-node subset communicators where data exchanges happen in parallel. Experiments performed on a system with two-lane physical channels show that their proposal is in most cases more efficient in using the additional bandwidth. It provides an average improvement of $2\times$ in execution time in comparison to currently employed algorithms over multiple implementations. When compared to these works, Sparbit is capable of achieving solid performance improvements without the need for additional hardware capabilities. Table 2 compares all the works presented in this Section by means of the presence or absence of certain aspects.

	(THAKUR; RABENSEIFNER; GROPP, 2005)	(TRäFF et al., 2008)	(MA et al., 2011)	(ARAP et al., 2015)	(TRäFF; HUNOLD, 2020)	(INOZEMTSEV; AFSAHI, 2012)	Sparbit
Optimal cost				✓	✓		✓
Free from hardware reliance	✓	✓	✓				✓
Free from additional calculations	✓	✓		✓	✓	✓	✓
Free from topology or machine information	✓	✓		✓			✓
Expansible for Allgather variants	✓		✓	✓	✓	✓	✓
Improvement for any rank distribution			✓				

Table 2 – Characteristics of other novel Allgather algorithms in relation to Sparbit.

From these condensed data it is visible that new algorithmic proposals either fall away from the optimal communication cost or employ hardware capabilities to provide more performance to classical optimal cost algorithms. The work of (MA et al., 2011) specifically has the advantage of providing improvements for any possible initial rank mapping, which naturally brings additional computation costs in return. With the exception of this last criteria, Sparbit presents all the desirable characteristics and therefore a good trade off of performance improvement and simplicity.

3.2 HIERARCHICAL ALGORITHMS

A trend in the development of novel and more efficient collective algorithms is employing the concept of process hierarchy. Considering the current non-uniform and highly hierarchical configuration of memory banks and processors on modern HPC infrastructures, these proposals

select one of the processes which reside on the machine to be the leader of that node. When collective communications happen, instead of every process communicating with several others, all the non-leader ones just transfer their blocks to the leader via intra-node communication. The leaders then exchange their individual sets of gathered blocks among each other employing regular collective algorithms. Finally, non-leader processes collect the blocks that were received on the inter-node phase.

This general approach is effective firstly because the number of processes involved on inter-node communication – which is the bottleneck in many cases – is much smaller, possibly reduced by a factor equivalent to the number of cores on each machine. The second factor is that specific communication optimizations can be better tailored for inter- and intra-node phases, such as using shared memory or zero-copy communication for local exchanges. On this track, the work of (KANDALLA et al., 2009) proposes the use of multiple leaders per node instead of only one, mostly to reduce the memory contention effect on the previous single-leader model. Therefore, rather than of all the processes sending or copying their blocks to only one destination, possibly exhausting the memory bus and crossing NUMA domains, each communicates only with its local leader which resides inside the same processor socket. The experimental results show that a fine-tuned number of leaders per node achieves better execution times than both conventional and hierarchical single leader algorithms.

Parsons e Pai (2014) present the concept of an Universal Hierarchical Algorithm which aims at flexibility, providing a guide implementation where different inter and intra-node communication algorithms can be plugged in to better suite a given system or particular execution. The tests where executed comparing the hierarchical against regular versions on multiple collectives, while varying several parameters such as the number of leaders and internal forms of communication. Results showed that the proposed universal hierarchical model achieved better execution time on most cases, reaching up to $29.9\times$ for the Allgather operation.

On (MA et al., 2012) the authors propose a collective hierarchical framework that focuses on a high integration between the intra- and inter-node parts of the communication. The main characteristic of this work is employing kernel-assisted single-copy memory transfer techniques to offload the internal communication to non-leader processes, while leaders can solely focus on inter-node communication. Reduce, Broadcast and Allgather algorithms were proposed and compared against MPICH2 and two collective components from Open MPI. For the Allgather call on an Ethernet cluster the proposed design surpassed all other options on the achieved bandwidth by varying amounts, although on an InfiniBand cluster it failed to deliver better performances than the alternatives.

The research on hierarchical algorithms is considerably vast and the works presented here are just a subset of the available contributions on this area. Nevertheless, Sparbit could be seamlessly employed on a multitude of hierarchical algorithms to handle the inter-node communication phase. In such a scenario, its improved locality can be explored to extract more performance as well as to better use the resources of the increasingly popular hierarchical

network topologies, as Fat-trees and Dragonflies. We do not provide a comparison table for this class of related work, as was done in Section 3.1, because it is much more a novel form of performing collective communications as a whole than individual comparable algorithms.

3.3 PROCESS REMAPPING

The last discussed form of improving the time of the Allgather operation, specially regarding overheads from poor communication locality, is through rank reordering or remapping. On this approach all the ranks are reorganized over the available machines following some particular order and as the data exchanges are governed by the placement of the processes, this makes the application's communication pattern better suited to the topology and machine's hierarchy. One very direct form of this approach is the one presented by (ALVAREZ-LLORENTE, J. et al., 2017) which changes the mapping from sequential to cyclic and vice-versa depending on the algorithm being executed. This approach shows great results on converting adverse mapping scenarios to good ones, with only local transformation functions and very little overhead. However, on the current form it requires gathering infrastructure information and supposes only equal distribution of processes to machines, thus on highly heterogeneous environments or with irregular mappings it would be unable to deliver great improvements.

The works of (MIRSADEGHI; AFSAHI, 2016) and (KURNOSOV, 2016) have a more Allgather focused approach, using its known communication pattern to create mappings more suited for the algorithms. The first proposes fine-tuned heuristics for Ring, Recursive Doubling and Binomial broadcast (a possible final component of an Allgather or Broadcast execution), with the experimental results presenting improvements up to 78%. Although, it needs either initial sends or final memory shifting to correct the blocks' positions, which adds overhead and even negatively degrades the performance; The second work employs graph partitioning and linear optimization to fit the communication pattern to the topology, presenting high potential improvements but requiring infrastructure information and additional computation time, which severely grows for large numbers of processes.

The works of (JEANNOT; MERCIER, 2010) and (LI; WANG; ZHU, 2013) focus on mappings for hierarchical tree networks and respectively propose the TreeMatch and Isomorphic Tree Mapping algorithms. These approaches assume that the topology structure is a balanced tree and require the profiling of applications in order to extract communication patterns. The information is then employed to map the ranks via the introduced algorithms. Both proposals show improvements over default forms of mapping, however require profiling information. In turn, the work of (JEANNOT; SARTORI, 2020) proposes online monitoring and rank remapping that provides improvements and does not need profiling, however it still requires active modification of application code, which is not always made available. Once again, we employ Table 3 using various criteria to provide a summarized relation between all the works showcased in this Section and Sparbit.

	(ALVAREZ-LLORENTE, J. et al., 2017)	(MIRSADEGHI; AFSAHI, 2016)	(KURNOSOV, 2016)	(JEANNOT; MERCIER, 2010)	(LI; WANG; ZHU, 2013)	(JEANNOT; SARTORI, 2020)	Sparbit
Free from previous application execution	✓	✓	✓			✓	✓
Free from application code modification	✓	✓	✓	✓	✓		✓
Free from additional calculations	✓						✓
Free from topology or machine information							✓
Improvement for any rank distribution		✓	✓	✓	✓	✓	

Table 3 – Characteristics of rank remapping techniques in relation to Sparbit.

All rank remapping techniques require gathering of topology or machine information and only one does not require additional computation in order to work. These drawbacks are in most cases paid back by improvement guarantees on any possible rank distribution. The works of (JEANNOT; MERCIER, 2010; LI; WANG; ZHU, 2013; JEANNOT; SARTORI, 2020) are capable of improving the communication of the application as a whole and not only of the collective fraction, which also comes with the price of needed profiling or application code modification. Sparbit could be potentially coupled with these techniques, however its main advantage in comparison is that it works out of the box, providing significant improvements on communication time for theoretically any hierarchical network, and without need for topology information, computation overhead or additional communication.

3.4 CONSIDERATIONS

This Chapter has been devoted to presenting the related work on improving the communication locality of the Allgather and Allgatherv calls or their performance as a whole. They are categorized in three groups: the first focusing on straightforward algorithms that directly implement the Allgather or Allgatherv semantics *per se*; the second comprising hierarchical proposals that decouple intra- and inter-machine communication; and the third which presents rank remapping techniques that instead of changing the actual communication routines, change the nodes upon which they are performed. In relation to the first group, our proposal presents considerable performance improvements without need for specific hardware or previous communication peer calculations. Towards the second group, Sparbit could be seamlessly coupled with such techniques as an inter-node communication choice with higher locality, and in some cases even in the internal communication phase. Comparing with the third group, our proposed work does not need large remapping calculations, application code modifications or previous

executions to ensure communication time improvements. Therefore, in comparison to the bulk of existing work, the advantage of Sparbit is that it makes no compromises on simplicity – and thus easy integration into existing technologies – to be able to provide consistent performance benefits. For summarization, the overall key aspects of Sparbit against the related work are:

- Optimal latency and bandwidth costs.
- No requirement of information about the hardware or topology.
- No requirement of specific hardware.
- Free from high complexity computations, previous application execution or code modification.
- Simple integration into existing implementations.
- Independent from particular applications or topologies.
- Dependent from the employed rank distribution.

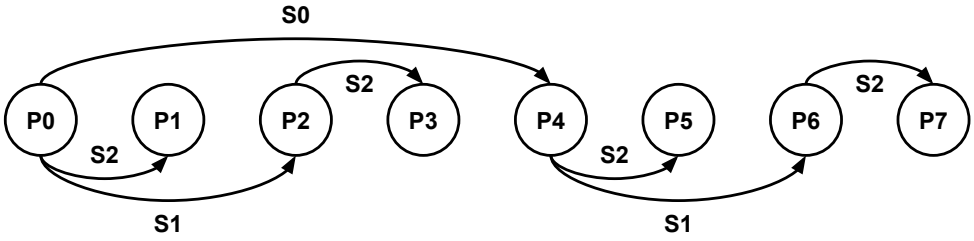
4 THE SPARBIT ALGORITHM

This Chapter outlines the design of the Stripe Parallel Binomial Trees (Sparbit) algorithm. It is inherent to the logarithmic algorithms that the amount of data exchanged at any given step will be larger than the amount of the previous one, usually by a factor of two, implying that the cost of data exchange grows at every step as a product of data size. This is true for Bruck, Recursive Doubling and as Sparbit is a logarithmic algorithm, syllogistically for it too. Additionally, if we suppose the usage of hierarchical network topologies then the cost of sending and receiving data increases with the growth of distance between any two communicating peers. We can then derive a cost estimate for every step as being a product of these two dimensions: the larger the data size and the further the distances travelled by the data, the higher will be the time cost of the communication. The former dimension can not be changed in order to maintain the logarithmic complexity, however there is no restriction to the latter. Stemming from this and through a higher behaviour perspective, the goal of Sparbit is to provide the same data exchange semantics as the Bruck algorithm, however instead of doubling the distances starting from 1 until $2^{\lceil log_2 p \rceil - 1}$, it takes the opposite direction and halves them starting from $2^{\lceil log_2 p \rceil - 1}$ until 1. This creates a more balanced distribution of communication costs along the algorithm's execution, since as the data sizes inevitably grow, the distance they must traverse continuously shrinks. Even though it seems like a minor modification, there are several major practical implications which will be discussed along this Chapter.

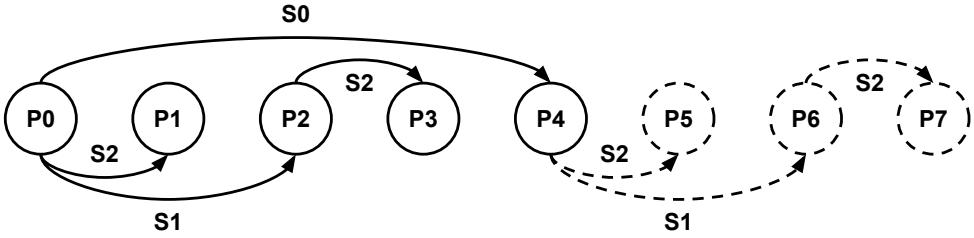
4.1 BINOMIAL TREE

As suggested on its name, Sparbit makes heavy use of the binomial tree to deliver data, which is a one to many logarithmic data delivery algorithm. Figure 10a shows a diagram of its execution for 8 processes, representing cases of power of two numbers of participants. Each circle represents a process and it is identified by a *P* letter followed by its rank. Arrows indicate the sending of a block from one process to another, while a nearby caption indicates the step in which it takes place, defined by an *S* followed by the step number.

The binomial tree is a recursive algorithm and as so, on every step $s \geq 0$, the set of p processes will be divided into smaller groups of size p_s, with $p_0 = p/2$ and $p_{s+1} = p_s/2$. Initially on the execution, the block of data resides in the root process, here assumed to have rank 0. On the first step, this general root will send the block to the process with rank $root + p_0$ and will delegate to it the responsibility of delivering the data to the processes with rank $r > p_0$. Therefore, this first step has divided the participants in two subtrees, with each subroot 0 and p_0 now responsible for sending the block to half of the processes. The algorithm is then repeated recursively and in parallel for each of the halves, continuously performing a subdivision into smaller subtrees until the data has reached all participants. From an individual point of view, if any process r has the block on the beginning of step s, it is considered the root of a tree and will send the data to the process with rank $r + p_s$, turning it into a new subroot as well.



(a) Power of two binomial tree ($p = 8$).



(b) Non power of two binomial tree ($p = 5$).

Figure 10 – Communication pattern of the Binomial Tree distribution algorithm. Circles represent processes and arrows represent the sending of a block. Dashed circles and arrows represent missing processes and ignored sends, respectively.

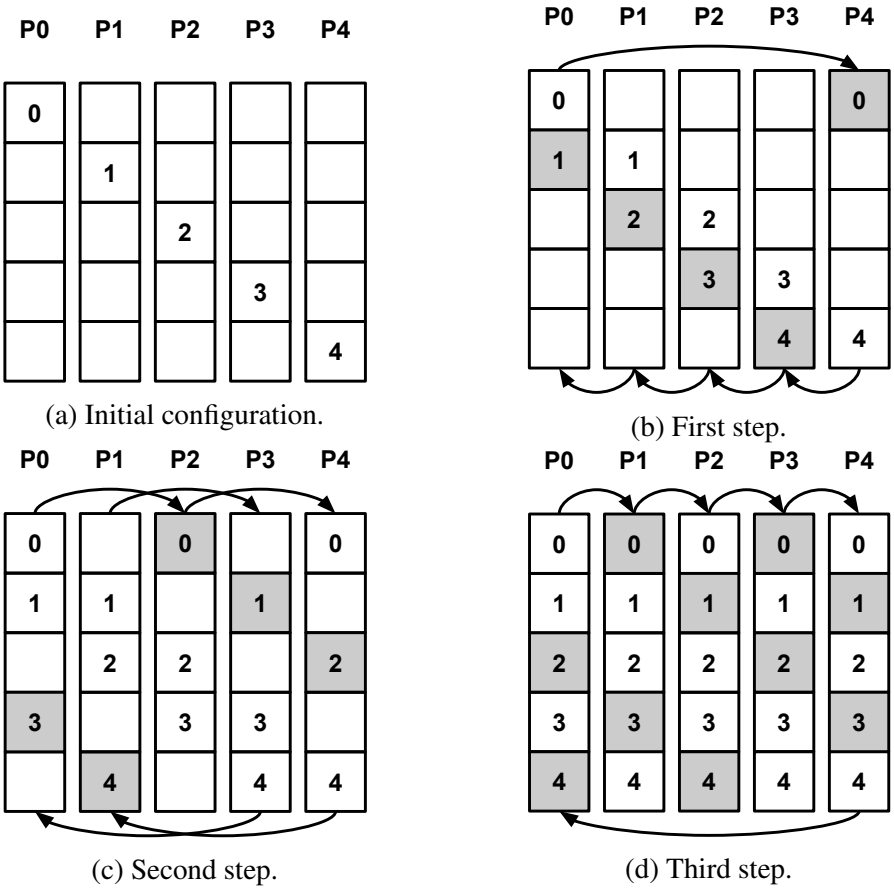
For a non power of two number or processes, as presented with 5 on Figure 10b, the algorithm works by exactly mimicking the way a binomial tree with size $2^{\lceil log_2 p \rceil}$ (immediately superior power of two) would work, however ignoring any sends that are destined to processes with rank equal or greater than p (Open MPI Team, 2020). Since all the sends happen *simultaneously*, then for each step only the time taken to send one block can be accounted for. Additionally, as only the one same block is sent by each process on every step, the final cost of the binomial tree algorithm is given by $C_{binomial\_tree} = \lceil log_2 p \rceil (\alpha + m\beta)$.

4.2 ALGORITHM DESIGN

Regarding the semantics of data delivery on an MPI Allgather call with p participants, each process must somehow send its block to all the other $p-1$ peers. If we take only the process with rank 0 and suppose that it is the root of a binomial tree composed of all the participating processes, then it would be trivial to deliver its block to all the destinations in a logarithmic number of steps. Moreover, if we imagine that the process rank space is circularly connected (*i.e.*, there is wrap-around in the destination calculation), then we can expand our previous supposition to any process r, since the tree organization would be exactly the same only shifted by r units. Finally from this concept, if all processes act as roots of the binomial tree destined to distribute their own block and simultaneously as leafs or intermediate nodes of all the other $p-1$ trees, the whole Allgather semantic can be achieved. This idea is the core of Sparbit and virtually all that it does is start and manage the participation of each process in the execution of all these parallel binomial trees. Bruck employs a similar general concept but with circulant graphs, which manifest themselves in the form of parallel spanning trees (Section 2.4.4). This is ultimately responsible for generating the communication locality issues discussed in Section 2.5.

Although the idea of the algorithm is simple, there is a considerable disconnect between its observed behaviour and the actual logic that makes it work. On the execution of the binomial tree distribution, on each step with distance d a process that already has the block will send it to a peer d away on the process rank space. If the number of processes is a power of two, then the execution is trivial and on every step $0 \leq s < log_2 p$ a process both sends and receives 2^s blocks to and from destinations $2^{log_2 p-(s+1)}$ away on each direction. However, if the amount of processes is not a power of two, then some sends will be ignored to avoid unexisting destinations. When employing wrap-around there are no unexisting destinations, but facing the same execution scenario and not ignoring these sends would imply in double writing the same block, which is not desirable. Hence, if we take again the example shown on Figure 10b with 5 processes and imagine that all the blocks initially reside on process 0, then on the first step process 4 would receive them. Now, since process 4 is a leaf and thus has no destinations, it would simply ignore the regular behaviour of forwarding the received blocks. When looking to the real Sparbit Allgather scenario the exact same logic applies, however the blocks are scattered along all the processes, with each distributing its portion through its own individual tree. Therefore, since each tree is shifted towards a unique root and all processes participate in all trees, then each process must act as the relative fourth destination - $(root + 4)\%p$ - to one of these trees. This implies that the role of ignoring sends, done in the initial supposition only by process 4, now must be done collectively by all processes, each ignoring the blocks received through the trees in which it is a leaf, but still forwarding the ones from trees in which it is an intermediate node.

The collective ignoring and forwarding behaviour can be clearly observed on Figure 11, that shows the buffer configuration for each process on different steps of the Sparbit execution. As complement, Figure 12 demonstrates the execution of the parallel binomial tree distributions which guide the actions of the processes through the steps. These binomial trees – each destined to distribute the block of a particular process, which acts as its root – are shown vertically stacked, with all of its participant processes presented on their relative roles through the horizontal arrangement. Each color represents a particular block and a respectively colored circle indicates that a process has already received that block relative to the horizontal tree where it is located. On step 0, each process r both sends its original block to process $r+4$ and receives the original block of process $r-4$, with wrap-around. Since the received block comes from the left, it is placed on the corresponding upwards distance on the receive buffer, also with wrap around as can be seen on processes 0 to 3 (Figure 11b). On the beginning of step 1 all processes have two blocks, however since the one received on the previous step comes from trees where the processes are leafs, then it must not be forwarded and again only the original block is sent on this step with distance 2 (Figure 12c). On the last step, each process has three blocks and the same one must still be ignored. Nonetheless, the other two belong respectively to the tree in which the process is a root and to the one where it is an intermediate node, thus both must be forwarded with distance one (Figures 11d and 12d). Figure 13 shows the communication pattern and movement of blocks relative to the buffer configurations presented on Figure 11. Blocks



(a) Initial configuration.

(b) First step.

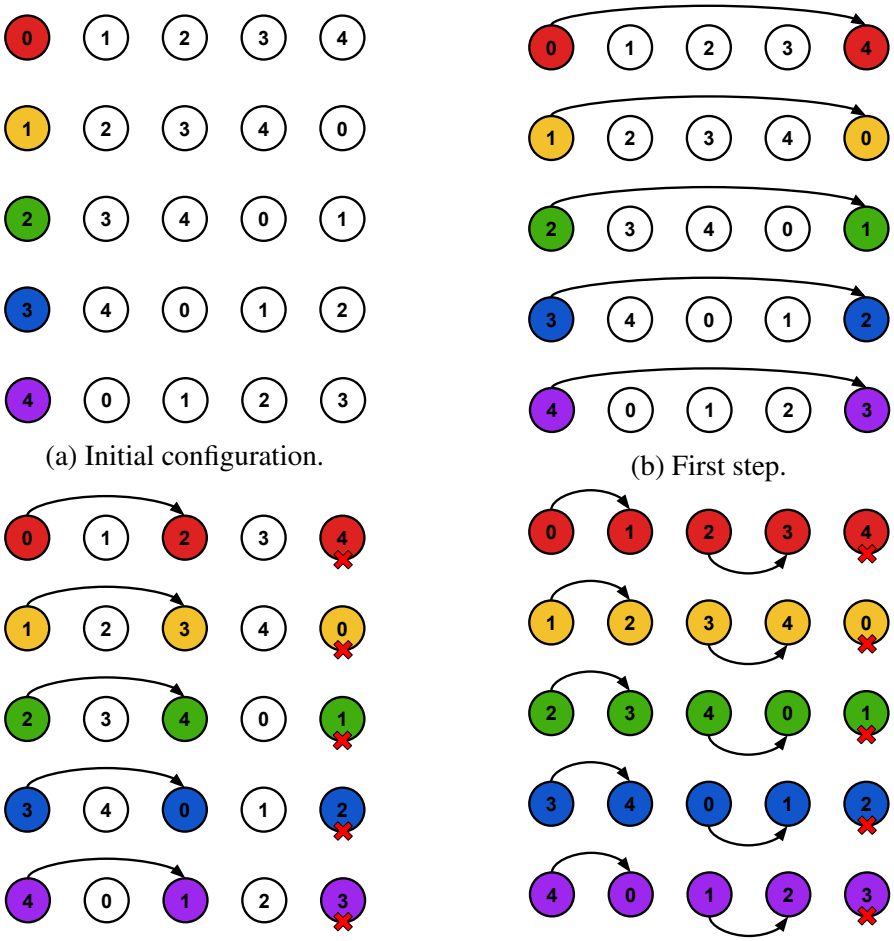
(c) Second step.

(d) Third step.

Figure 11 – Sparbit buffer configuration on different steps for 5 processes. Grey blocks are received on the current step.

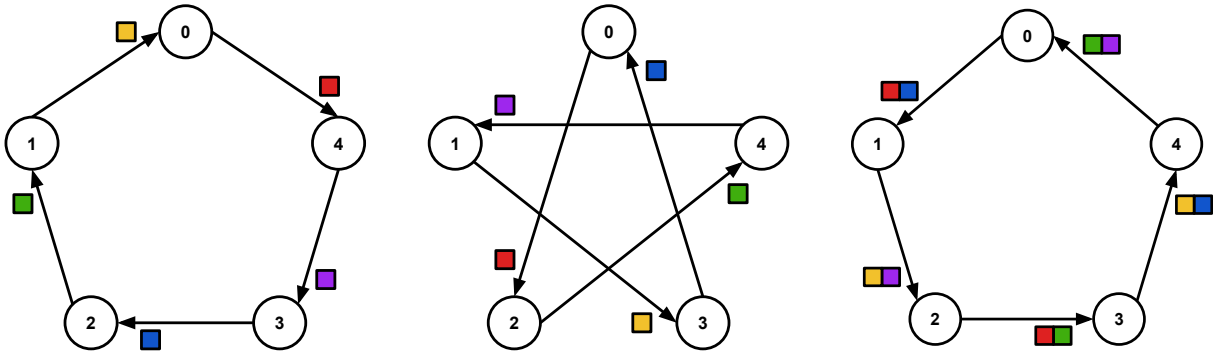
have colors for identification according to the trees of Figure 12 and are sorted on the sends from left to right according to their original process' rank order. From Figures 11 to 13, it is possible to verify that each process receives all the blocks.

Another way to see a binomial tree is as a recursive structure composed of smaller power of two binomial trees, whose roots must receive the data to further distribute it amongst their branches. In the example of Figure 10b there are two such subtrees, one with 4 processes of ranks 0 to 3, and another with 1 process of rank 4. Therefore, if we employ this rationale on the example, the first step takes the data from the root of the first tree to the root of the second one, performing what we call a tree expansion. If the number of processes was larger there would be more subtrees, and therefore more tree expansions during the execution. For example if there were 21 processes, there would be three subtrees of sizes 16, 4 and 1. Also since the distance halves, the trees are organized in descending order, with larger subtrees comprising the smaller relative ranks and the general root always being the root of the first one. Hence, the block flows from the larger to the smaller subtrees and a tree expansion will only happen on a step with distance d if there is a tree of size $s_1 = d$ that has the block and another tree with size $s_2 < s_1$ that does not. Returning to the example of 21 processes, there would be two tree expansions: one to take the data from the 16 processes tree – whose root is also the general root and the only one



(a) Initial configuration. (b) First step. (c) Second step. (d) Third step.

Figure 12 – Sparbit parallel trees on different steps for 5 processes. Colored circles indicate processes that have the block from that tree by the end of the current step.



(a) First step. (b) Second step. (c) Third step.

Figure 13 – Communication pattern of the Sparbit algorithm on different steps for 5 processes.

that has the data on the beginning – to the one with 4 processes; and another expansion to take the data from this tree to the one with a single process. From the 21 and 5 processes examples, it is clear that no matter the number of subtrees t, there will always be needed $t - 1$ tree expansions.

With all this in mind, Algorithm 1 presents the Sparbit's pseudo code. Function signatures for `Isend` and `Irecv` were simplified for brevity, taking destination/source, buffer displacement

Algorithm 1: Sparbit.

input : The amount of processes p; the rank of the current process *rank*

```
1  data ← 1;
2  ignore ← 0;
3  d ← 2^⌈log2 p⌉−1;
4  last_ignore ← count_trailing_zeros(p);
5  ignore_steps ← (¬(p >> last_ignore) | 1) << last_ignore;
6  for 0 ≤ i < ⌈log2 p⌉ do
7      if d & ignore_vector then
8          ignore ← 1;
9      end
10     for 0 ≤ j < data − ignore do
11         MPI_Isend(rank + d % p, (rank − (2j)d + p) % p, 1);
12         MPI_Irecv((rank − d + p) % p, (rank − (2j + 1)d + p) % p, 1);
13     end
14     MPI_Waitall()
15     d ← d >> 1;
16     data ← (data << 1) − ignore;
17     ignore ← 0;
18 end
```

in blocks and number of blocks as parameters. Initially, a set of variables is defined and populated to hold important information about the current state of the algorithm. The `data` variable stores how many blocks are expected to be received and likewise sent on the current step. The `ignore` variable is binary and stores whether a block should be ignored on the current step or not. As discussed earlier, when p is not a power of two, in some cases the sending of certain blocks must be ignored by all processes to avoid double writing it on the destination. Since every process has the role of ignoring the send in one of the p trees and each tree distributes one block, then each process will ignore at most one block at any given step, which justifies the binary nature of the `ignore` variable. The distance of the communication is kept on the `d` variable, and starts from the power of two immediately smaller than p.

Processes	Binary representation	Amount of subtrees	Size of subtrees
5	0b00101	2	4, 1
12	0b01100	2	8, 4
16	0b10000	1	16
21	0b10101	3	16, 4, 1
31	0b11111	5	16, 8, 4, 2, 1

Table 4 – Process numbers along with binary representations and subtrees.

Finally, the number of subtrees inside the general binomial tree, by definition, represents the size of the smallest non-repeating set of powers of two that when summed equal the total amount of processes p. This is conveniently also the definition of the binary representation of any natural number p and as can be seen on table 4, by simply having its binary representation,

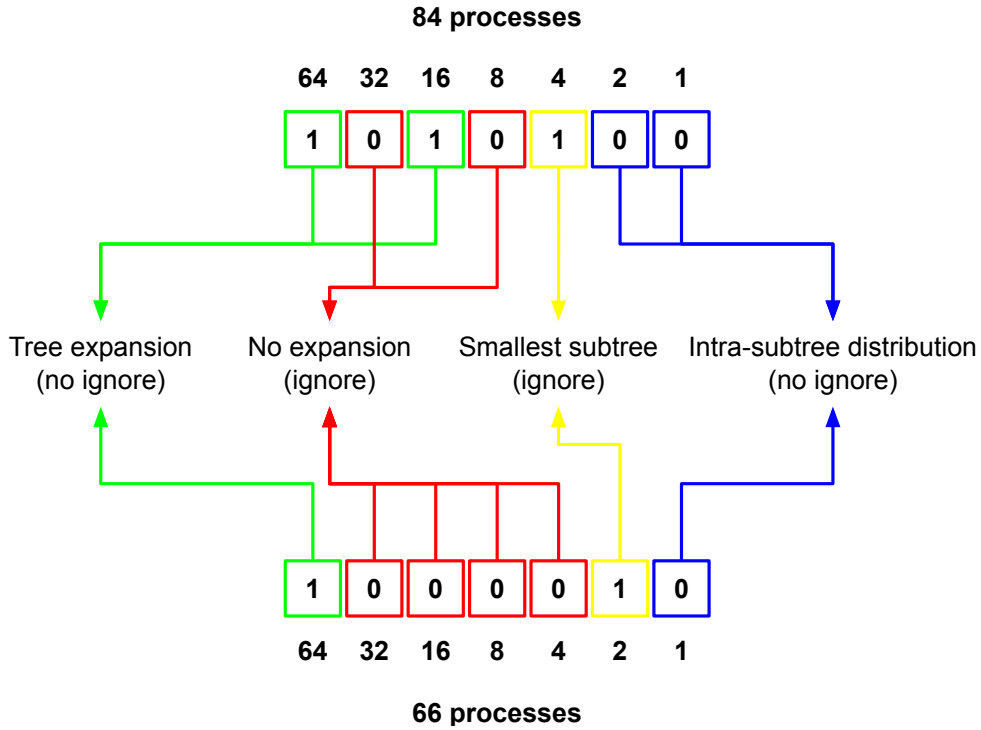


Figure 14 – Binary representation of process numbers and meaning of each set of bits.

the powers of two that represent the component subtrees are also already known. With this information, it is possible to determine whether a tree expansion should be made on the current step, which in practice simply means that no blocks should be ignored. Therefore, if p is a power of two, then there are no component subtrees and no blocks will be ignored on any step. However, if p is not a power of two, then:

- on a step with distance d, if there is a subtree with size $s_i = d$ and another subtree with size $s_j < d$, then a tree expansion should be performed.
- If d is smaller then the size of the smallest subtree s_0, then no blocks should be ignored on any further step, since now distributions are happening inside each subtree – and not between them – and thus all data should be forwarded.
- The smallest subtree, with size s_0, will never be expanded on step with distance $d = s_0$ since there are no smaller subtrees. Therefore, on distance $d = s_0$ a block will always be ignored.
- If d is larger than the size of the smallest subtree s_0 and there is no tree with size $s_i = d$, then a block should be ignored.

To clarify these rules, Figure 14 provides an example with 84 and 66 processes, showing the binary representation and how each set of bits determines the ignoring behaviour that the algorithm should have on a step with a particular distance. It is visible that from the bit that represents the smallest subtree all the way to the right, the binary values are encoded paired

with the ignoring behaviour. Hence, if a bit is set then on the step with that distance a block should be ignored and if it is not, then all blocks should be sent. A similar trend can be observed in the remaining bits on the left, however on this case they are inverted. Thus for this region, when a bit is set all blocks should be sent, while one should be ignored when it is not. Now, if certain modifications are applied to the binary representation of p, it is possible to create an homogeneous encoding that describes on which distances a block should be ignored, if the bit is set, and on which distances all blocks should be sent, if the bit is not set. Therefore, by simply inverting all the bits to the left of the first one that is set we achieve this encoding and that is performed on lines 4 and 5 of Algorithm 1. Line 4 simply counts all the zeros at the far right of the bit array, discovering the position of the first bit that is set. Then on line 5, a series of bit to bit operations are employed to shift the bits to the right, invert them and shift them back to their original position, with the result stored inside the `ignore_steps` variable.

The algorithm then loops for a logarithmic amount of steps on lines 6 to 18. On every iteration, if the current distance requires a block to be ignored, the appropriate variable is set for such on lines 7 to 9 (note that & is the bit to bit *and* operator). All the blocks that should be sent and received on the step get dedicated asynchronous sends and receives respectively issued, on lines 10 to 13. The destination for the sends is the process with rank $r+d$ while the source for the receives is the one with rank $r-d$, both wrapping around in the rank dimension. Since blocks are received from the left when imagining the rank space horizontally, they are also placed to the left of the receiver's rank position on the receive buffer (subtraction of d from the rank on the second parameter of the receive call). Through a similar rationale, as the blocks that get sent on the current step are the ones received on the previous ones, the send call also cycles to the left of the receive buffer to extract blocks to send, always starting from the process' own block (with $j = 0$) since it is inevitably sent on all steps. Finally, all the asynchronous communication requests are waited for on line 14, the distance halves on line 15, the expected amount of data for the next step is calculated on line 16 and the ignore variable reset on line 17. The way the algorithm works allows it to perform the block exchange for any amount of processes, with a logarithmic number of steps and improved data locality. The next Section is dedicated to presenting and discussing the theoretical cost of Sparbit.

4.3 THEORETICAL COST AND NON-CONTIGUOUS DATA

As can be seen from its definition – more specifically on Figure 11 – Sparbit does not employ contiguous usage of memory for block placement. Instead, data sections are intertwined with blank spaces destined to receive future data. This also implies that a process executing Sparbit on a stage with multiple blocks is unable to send all of them to its peer in one send without memory copies. Previous work (BENSON et al., 2003) have highlighted the importance of communicating contiguous data for communication performance. Algorithms such as Bruck achieve contiguous memory organization through the communication direction and data copies

(THAKUR; RABENSEIFNER; GROPP, 2005), compromising part of the total time with memory shifts to be able to reduce the communication fraction by employing just one send per step (BRUCK et al., 1997). Furthermore, performing two sends instead of memory copies although more efficient for large data sizes, performs poorly for smaller ones (BENSON et al., 2003). Thus, the available options are to use memory copies, which are known to hinder the algorithm's performance, employing derived datatypes, whose communication presents several overheads on the MPI implementations (LI et al., 2015), or a third option not yet discussed which is pipelining multiple sends.

On the Hockney model, α represents the communication start-up time and may account several different initial aspects necessary to perform a communication. In turn, the β term for the transfer time per byte can be seen as a direct reciprocal of the channel's capacity or throughput, and if each byte takes β seconds, the channel can thus deliver $B = \frac{1}{\beta}$ bytes per second. If we further generalize a byte as a group or component of some arbitrary transfer unit u (*e.g.*, bits, packets), the amount of transfer units a channel can send per second is given by $U = B\omega$, with $\omega > 0, \omega \in \mathbb{R}$ representing a transformation scalar. We can hence model the total bandwidth cost of a message as the amount of transfer units consumed, so for m bytes, $m\omega$ units would be needed. If a message's units were strictly send sequentially, the total cost in time for bandwidth would be trivially obtained by $c_b = \frac{m\omega}{U}$. Now if we would split the message and send first one half and then the other, while momentarily ignoring the start-up cost, the bandwidth cost would be $c_b = c_{b1} + c_{b2}$, with $c_{b1} = c_{b2} = \frac{m\omega}{2U}$ and thus an equal final cost of $c_b = \frac{m\omega}{U}$. If however, instead of separately sending each half we pipelined them in fractions, alternately using one unit for sending a part of the first half and the subsequent unit for a part of the second one, then it is also easy to verify that the total cost would still be the same. Therefore, in any arrangement of actual data transferring, the bandwidth cost will be the same.

Finally, we now consider the latency in the two pipelined half data sends. If the two start-up procedures, each with cost α, could not be pipelined or parallelized, the difference in time to the one send version would be α, since the first half's transfer time is shadowed by the second half's latency and transfer time. However, if the start-up cost could also be intertwined (which is often the case, like on multiple parallel Transmission Control Protocol (TCP) handshakes) then the difference to the regular version would be given by a τ issuing cost term, which indicates the delay between the start of the first half's communication process and the start of the second one. The time accounted by τ on a system represents the speed in which a node can process and issue new connection requests to the wire, being theoretically a product both of Central Processing Unit (CPU) and NIC performances.

We could further divide the original data size m into $k > 2$ parts and employ the pipeline approach in all of them. Hence, the cost analysis would still be the same, but since now there are k pipelined sends, the issuing cost would be $k\tau$. Considering parallel start-ups over the Hockney model for m bytes and k parts, the total cost of pipelined sends is $k\tau + \alpha + \frac{m}{p}\beta$. When using this technique on Sparbit, the value of k starts with 1 and as data doubles it rises up

to $\frac{p}{2}$ on the last step. Therefore, the cost due to the τ term from the second step onwards ($k \geq 2$) is given by $\tau \sum_{i=1}^{log_2 p-1}(2^i - 1) = \tau(p - log_2 p - 1)$, and with this Sparbit's total cost is $C_{sparbit} = (p - log_2 p - 1)\tau + \lceil log_2 p \rceil \alpha + (p-1)\frac{m}{p}\beta$. The value of τ is expected to be very small on modern systems, and the experimental results suggest the same.

Naturally, as the value of τ grows linearly with the increase of p, its parcel of the total time is also expected to grow and to be more expressive when a large number of processes is employed. When stretching the execution to hundreds of thousands or even millions of processes as expected on the exascale age, the impact of the τ term might be very severe, with possible sources of degradation being either the time itself taken to kickstart so many parallel connections or bandwidth contentions arising from the united demand of all the transferences. For such extreme scenarios, concessions on memory shifts or the employment of indexed datatypes may be needed to reduce performance degradations. Therefore, a number of parallel sends much smaller than $\frac{p}{2}$ could be achieved by grouping blocks and sending them in parallel batches instead of individually, with the block union performed by one of the aforementioned methods. With this approach the total number of parallel sends could be reduced by any factor $1 \leq \gamma \leq \frac{p}{2}$, although the performance improvements are not necessarily proportional to γ and therefore its value should be carefully chosen through experimental verification.

Upon the usage of massive amounts of processes and in comparison with other algorithms, the degrading effects of τ might be partially or even completely damped by Sparbit's greater communication locality. This observation theoretically stems from the fact that a larger number of processes also requires a larger infrastructure with potentially deeper and wider hierarchies, which in many cases has a significant negative impact on other algorithms, as discussed in Section 2.5. This degradation on other alternatives due to the hierarchy could be much worse than the time overheads due to the parallel sends on Sparbit, rendering it still a possibly better choice. All these latter implications of the cost formulation, specially on executions with higher dimensionality, are yet subject to experimentation for validity regarding the impact of the τ term, Sparbit's ability to reduce it and the possibility and fine-tuning of mitigation techniques for eventual performance degradations.

4.4 CONSIDERATIONS

In this Chapter the proposal of the Sparbit algorithm has been discussed guided through its fundamental concepts. First, the general rationale behind its functioning is presented as being an *inverse* logarithmic algorithm, that halves the distances travelled while the data size doubles. This is at odds with other existing logarithmic alternatives which double both the distance and size of information as the number of steps progresses, naturally imposing heavy transference costs due to large data moving over long paths. On the sequence, the aspects that allow Sparbit to actually work are presented, applying large focus on the binomial tree and how the behaviour of each process involved on the communication must be modified and tailored to

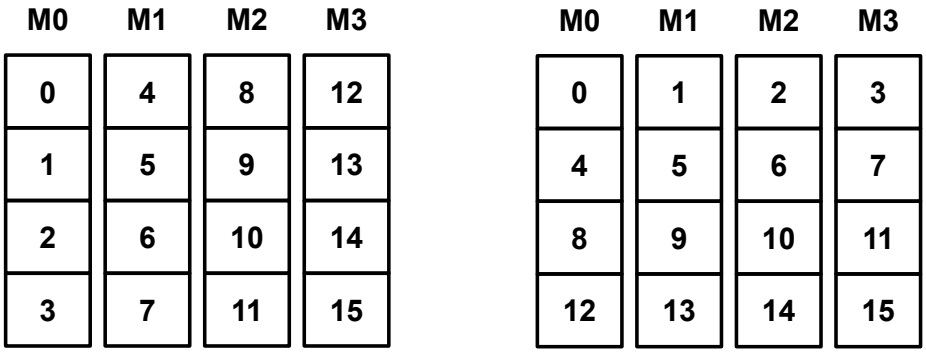
manage participation in several parallel tree distributions. The scenarios of non-power of two number of processes, where certain blocks must not be sent to avoid double writing, are also tackled in detail. The pseudo code describing the actions of a peer throughout the execution is presented on the following segments and related to the previous theoretical behaviour-oriented explanations. Finally, a cost specification is devised for the proposal following the lines of the Hockney model presented in Section 2.4.

5 MAPPINGS AND PERFORMANCE

This Chapter delves into a number of complementary implications that revolve around the individual functioning of the algorithms or that merely emerge from their usage on real systems. Its main subject is the complex relation between forms of mapping processes to the machines and the execution of the algorithms, particularly Bruck and Sparbit. The presented reasoning is key to understand and predict the performance of executions on multiple scenarios of infrastructure and mappings, being a comprehensive and precise tool that explains diverse behaviours experimentally encountered, as further detailed in Chapter 6.

5.1 DEFAULT MAPPINGS

The mapping of MPI processes to the topology is done by the assignment of ranks to process instantiations on the machines, and since any application's communication pattern is done by means of the processes' ranks, then transitively the manner in which ranks are distributed plays a significant role on the communication costs. The two main predefined ways in which MPI implementations map ranks to nodes are sequential and cyclic. The first performs a best-fit approach, completely filling the slots of a machine with sequential ranks before going to the next one. The second uses a round-robin approach, assigning only one rank to a machine and then moving to the next in a circular fashion until they are all filled. Depending on the implementation and on the chosen parameters, either one of these forms is employed in the initial rank distribution, and on regular cases processes stick with their received ranks until the end of the execution. By default, Open MPI employs sequential mapping while MPICH employs cyclic.



(a) Sequential mapping. (b) Cyclic mapping.

Figure 15 – Rank distribution of sequential and cyclic mappings on 4 machines with 4 slots each.

Figure 15 shows the difference in final rank organization on the machines' slots according to sequential and cyclic mappings. On equal number of machines and slots both are transposed versions of each other and while the resulting sequential form is more strongly dependent on the number of available slots on each machine, the resulting cyclic form is more dependent on the

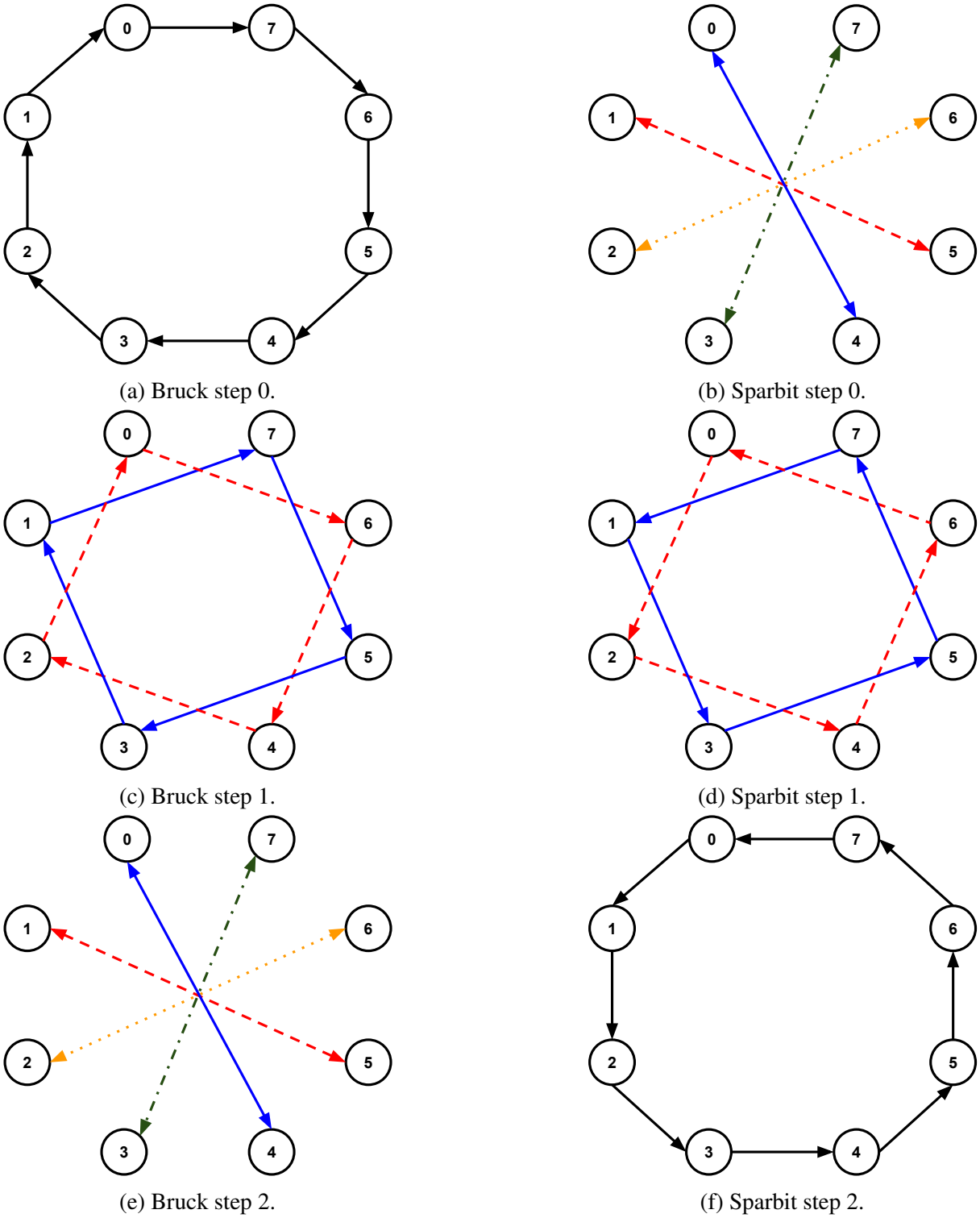
total number of machines. On oversubscribed infrastructure to processes scenarios, the sequential mapping presents more utilization of less nodes, while the cyclic mapping presents the opposite.

If we imagine that the cost of communication among processes that are inside the same machine is smaller than the cost of communication across different machines – which is commonly the case on modern systems – then insights can be drawn about the relation between the mapping and the total communication time of an algorithm. Further, analysis can also be made in order to understand the effects of varying the amount of cores or machines on the final performance of each option. Throughout this Chapter, discussions regarding these aspects of process mappings are presented with focus on Sparbit and Bruck. The latter receives more attention in comparison to other traditional algorithms due to its similar characteristics of theoretical communication costs, working for any number of processes and overall functioning, which renders it the main alternative choice. The goal here is not to provide formulations for the optimal mapping on any case, since this is both a hard problem and also highly variable depending on p. Instead, the goal is to present a deeper discussion about the relation of each algorithm with the default mappings upon which they execute on the experiments. For that to be possible, in Section 5.2 we first decompose the whole functioning of each algorithm into its building blocks, which allows identifying directions to what suitable mappings might be. Then, employing such information we analyse how favourable the default mappings will be for each algorithm on generalized execution scenarios. Finally in Section 5.3, the discussion proceeds to how the default mappings will affect the algorithms' communication performance with varying amounts of machines and cores.

5.2 BEST DEFAULT MAPPINGS FOR EACH ALGORITHM

Another way to understand Bruck and Sparbit is as if the block transmissions on every step formed a set of disjoint parallel rings. Therefore, each communication round can be modelled as a potentially different set of one or more rings over which data is moved, each functioning for a subset of the processes much like one of the iterations of the Ring algorithm, presented in Section 2.4.1. Figure 16 shows the execution steps of both Bruck and Sparbit for 8 processes, with the sending of blocks represented by arrows and the rings to which each message pertains identified by different line types and colors. These diagrams outline that the amount of rings, their compositions or both can change when the steps evolve and these changes are inherent to both of the algorithms' executions, as formalized next.

For any step s, the g_s rings formed and the k_s processes that compose each ring are mainly defined by the distance of the communication and will follow the relation $g_s k_s = p$, with p representing the total amount of processes. This equation creates a multiplicative inverse correspondence between g_s and k_s, since if one grows the other needs to diminish by the same factor to maintain the relation. The highest number of rings employed throughout the execution will be $g_{max} = max(2^i)$, such that $\forall j,i \in \mathbb{N},\ 2 \leq j < \sqrt{p},\ j2^i = p$, or in other terms, it will be the



(a) Bruck step 0.

(b) Sparbit step 0.

(c) Bruck step 1.

(d) Sparbit step 1.

(e) Bruck step 2.

(f) Sparbit step 2.

Figure 16 – Component distribution rings of Bruck and Sparbit on different steps for 8 processes. The line type and color of an arrow indicate to which ring the communication belongs.

highest power of two that divides p. From this reasoning, if p is a power of two, then g_{max} will be $p/2$ and k_{min} will always be 2 (due to the inverse correspondence g_{max} implies k_{min}). Also, if p is odd then g_{max} will be one, since the highest power of two that divides an odd number is 1 and so, on this case all processes will be part of the same ring on all steps, with disregard for the

current distance. At last, if p is even but not a power of two, then the maximum number of rings will be highly variable based on the multiplicative factors of p.

Following the functioning of the algorithms with power of two numbers of processes, Bruck starts with a communication distance of 1 and doubles it until $2^{\lceil log_2 p \rceil - 1}$. This makes the first step to have only one ring in which all processes participate. As distances rise at every new step, the number of rings double and the amount of participants per ring halves, until the last step where there will be $p/2$ rings each with 2 processes. Sparbit on the other hand, starts with a communication distance of $2^{\lceil log_2 p \rceil - 1}$ and halves it until 1. This naturally makes the rings' behaviour to be the opposite, so the algorithm starts with $p/2$ rings each with 2 processes and at every new step there will be half of the previous rings with the double of processes, until the last step with only one ring. These behaviours can be more clearly visualized on Figure 16 and are extended in the same fashion to even numbers of processes, but starting or finishing with the maximum rings as defined for this parity.

Understanding how these component rings manifest themselves in each algorithm is useful because if we are able to map the processes to the machines in a way that is most favourable for the execution of such rings, then as they exactly model the algorithms as a whole, it will also be the most favourable way to map the algorithms itself. In this sense, if all rings transported the same data across processes, the mapping would be a one dimensional problem and any maximal subset of rings could be optimized for communication locality. It could also possibly have a number of different optimal solutions and the same one would be equally valid for Sparbit or Bruck. Nevertheless, rings do not transport the same amount of data and as discussed along this Section, Section 2.4 and Chapter 4, Sparbit and Bruck have distinct relations between communication distances and message sizes, which imply that the most favourable mapping will not be equal for the two options. Naturally, as these algorithms have logarithmic complexity the data transferred doubles with every new step, meaning that the ring or rings closer to the last steps will move more data than the ones closer to the first steps. Therefore, since the mapping is established before the execution, in order to have the smallest data movement cost, rings which have the largest aggregate data should be prioritized in the mapping.

For Sparbit, since it has a non-decreasing data size progression, the ring with the heaviest communication will always be the one from the last step. Bruck does not have the same property and depending on the value of p, the heaviest communication will be on the rings either of the last or the second to last step. This variability characteristic of the latter alone already poses restrictions to whether a default mapping could be optimal on this case, since the potential for optimality following the same rank distribution behaviour would greatly vary according to p. Hereafter throughout the text, the term *hierarchical domain* is used to describe a set of elements whose communication with each other has an overall similar cost (on the same order of magnitude), with examples including cores on the same processor, processors on the same machine, machines on the same switch, *etc.* The number of entities of the same type that compose an hierarchical domain (number of cores, processors, machines, switches and others) will vary

depending on the infrastructure and thus only general analysis are made on this sense. Either way, even without such precise information, the quality of a given mapping will in general be proportional to its ability to keep most of communications close together inside the deepest hierarchical domains.

Under the light of the previous discussions, for Sparbit and Bruck, the best mapping will be the one that is able to couple closest together the maximum number of their heaviest communicating rings. Now, for the mapping of any ring to be the best for a scenario, neighbors which communicate should be placed as close as possible, maximizing the number of local exchanges and minimizing the ones who must cross machines or other hierarchical domains. Supposing a power of two number of processes, as data doubles at every step and from the solution to the sum of 2^i which describes this behaviour, it is known that the last step will transfer the sum of the data of all previous steps plus one. Therefore, if a choice should be made regarding the preferred ring or set of rings to have the best mapping over all others, it should be the ones from the last step, since even if we were able to optimally map all other rings from earlier steps, their combined data volume would still be smaller than the one from the last step. From this and the examples on Figure 16 we can already start to reason about what would be the best default mapping for each algorithm.

Sparbit has only one ring on the last step, which comprises all processes and moves the largest individual data size. Thus to optimize its mapping, the most number of neighboring peers should be placed together and as the distance between neighbors on this ring is one, then the default sequential mapping will be one of the optimal choices. This allows only one send per hierarchical domain to exit it, while all the rest of communications happen locally. Considering a number of machines m and cores c, of the $p = mc$ processes, only m sends will actually cross different machines and the less machines with more cores the better. Changes to include simultaneous mapping improvements for rings of earlier steps are not possible without damaging the cost, as exchanging ranks of this mapping across machines – although possibly improving earlier stages – damages the last ring, whose larger data sizes more significantly impact the total cost in a negative way. This happens because, as can be seen on the example of Figure 16f, the last ring involves processes of both parities while the prior rings, showcased on Figures 16b and 16d, are composed only of processes of the same parity. Hence, if we try two more closely group processes of the same parity – to improve earlier rings – we will by definition separate the ones with different parities, worsening the communication of the last step. Through a similar rationale, the cyclic mapping for Sparbit will in most cases produce unfavourable rank distributions, due to the placement of processes with nearby ranks always on different machines. Therefore, if cyclic mapping was employed, all neighboring processes on the last heaviest ring would need to recur to non-local communication to perform the data exchanges, augmenting the time required to do so.

Bruck has g_{max} rings on the last step that combined move the largest individual data size, each having p/g_{max} or k_{min} processes. Therefore, the best mapping will be the one in which all

processes of every ring are close together. If the number of cores c is equal to the number of processes on every ring, then the last step can be performed with no message having to cross machines or other hierarchical domains, since a whole ring can fit inside a single one of them. If the number of machines m is equal to g_{max} or a power of two divisor, then the default cyclic mapping will position all the processes of each ring inside the same machine, thus being optimal for this case. If m is a power of two divisor of g_{max}, it means that c must be a multiple of the k_{min} processes on each ring and it is possible to fit more than one ring inside each machine. When this is the case and while employing the cyclic mapping, interesting properties start to emerge.

As seen before, the sequential mapping for Sparbit could not be jointly optimized for the last ring and the earlier ones at the same time due to the mixed parity composition of the former. On the Bruck case however, from Figures 16c and 16e it is visible that the processes that form each ring from the last and second to last steps have all the same parity. In fact due to the rings' doubling, this pattern is visible for any p on all steps except the first – the same on Sparbit but inverse. Interestingly, as the steps progress rings are not actually doubling but dividing in two, and this leads to the conclusion that if we are able to completely fit a ring from an intermediate step inside of one hierarchical domain, such as a single machine, then all rings from subsequent steps will also be completely mapped inside of it, as they are merely the result from the fragmentation of the previous one. For an example we can look once again for Figure 16, and if all the even processes of Bruck's step 1 – which form a ring (0, 2, 4, 6) – were mapped inside the same machine, the other two rings formed by even processes on the last step would also be – rings (0, 4) and (2, 6). This happens because distances double and rings are circularly connected, so if any process is part of a ring it has two neighbors, each d ranks away and themselves apart $2d$ ranks from each other. Now, when distances double to form the new rings, all processes who were $2d$ away on the previous one will now be direct neighbors. Hence, any new rings will be composed by a subset of the anterior and any two processes that shared a neighbor will be on the same resulting ring.

Such phenomenon yields for Bruck a higher locality potential, since if the number of machines m is a power of two smaller than g_{max}, then the cyclic mapping is able to optimally map more rings from earlier steps along with all subsequent ones. In contrast, the sequential mapping will in most cases not produce favourable rank distributions for Bruck, since if it was employed, the distribution would be the best for the ring of the first step, which contains mixed parity participant processes. However, the heaviest communicating rings are the ones in which all participants have the same parity and thus, employing this mapping would create a suboptimal distribution for them, hindering the improved potential for locality on Bruck.

5.3 QUALITY OF DEFAULT MAPPINGS AND STABILITY

The bottom line of the reasoning presented in Section 5.2 is that for Sparbit one of the optimal mappings is indeed the default sequential one and thus it should be preferred over cyclic.

On this case its heaviest communication ring is mapped in the most local way possible and earlier rings will not have their locality significantly affected, rendering the smallest communication costs. For Bruck, the cyclic mapping will be optimal on certain cases, allowing it to have an increased potential for communication locality over Sparbit, that extends for multiple steps due to the subset nature of later rings. For a given pair of m machines and c cores per machine, the cyclic mapping will be optimal for Bruck when m is equal to g_{max} (maximum number of rings of the execution), and since we are supposing that $p = mc$, when the number of cores c is equal to k_{min} (minimum number of processes on each ring). Benefits on its total communication time grow if m is a power of two smaller than g_{max}, since then the cyclic mapping will place rings of multiple steps inside the same hierarchical domain. In both cases, m must assume the value of a power of two for the cyclic mapping to be favourable.

M0	M1	M2	M3
0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

(a) 4 machines with 5 cores.

M0	M1	M2	M3	M4
0	4	8	12	16
1	5	9	13	17
2	6	10	14	18
3	7	11	15	19

(b) 5 machines with 4 cores.

Figure 17 – Sequential mapping on different infrastructure configurations of machines and cores.

M0	M1	M2	M3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

(a) 4 machines with 5 cores.

M0	M1	M2	M3	M4
0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

(b) 5 machines with 4 cores.

Figure 18 – Cyclic mapping on different infrastructure configurations of machines and cores.

When comparing the sequential mapping for Sparbit against cyclic for Bruck with the focus on stability, or the ability to maintain favourable mappings under different parameter scenarios of p, m and c, consistent differences appear. For Sparbit, the sequential mapping will

remain a favourable option on virtually any value of m and c, since as can be seen on Figures 17a and 17b, by varying either the number of cores or machines, the heaviest communication ring will still be mapped in the most local way possible. Hence, although these changes might affect the quality of mapping for earlier rings, the bulk of communication will be held on the machine's neighborhood, rendering this mapping a much stable choice for Sparbit. For Bruck under the cyclic mapping, varying c has no negative effect on communication costs, and if m is fixed while the number of cores grow then it will produce very favourable mappings. However this will only hold if m is a power of two, such as portrayed on Figure 18a, and if it is not, such as on Figure 18b, then for all values of c the resulting rank distribution will not be favourable, as all ring neighbors of each process will be placed on different machines or hierarchical domains. If we temporarily abandon the $p = mc$ relation and suppose a number of processes which is odd, then even with a power of two amount of machines, the cyclic mapping will be suboptimal for Bruck. This happens because when p is odd the maximum amount of distribution rings at any step g_{max} will be 1, in which all processes of mixed parities participate. Therefore, since processes of the same parity will be mapped to the same machine, various exchanges will inevitably have to cross domains, severely hurting its potential for locality. Hence for Bruck, apart from cases where m is a power of two, a custom mapping would be necessary for the rank distribution to be favourable, and as discussed in Chapter 3, this poses its own challenges.

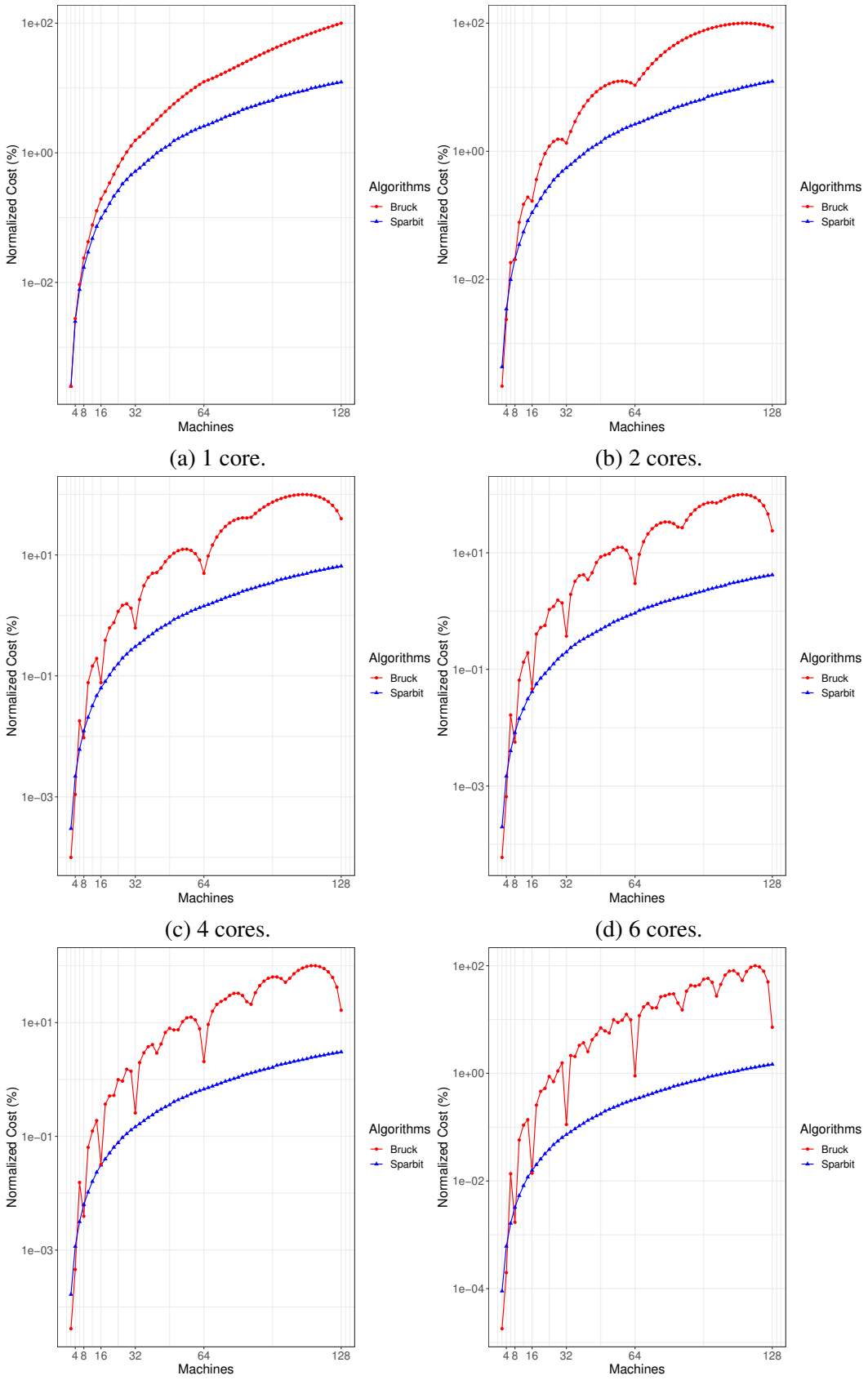
The unfavourable mapping of an algorithm has a great toll on the total communication cost of its execution. Considering the distance and the data size as two component factors of the communication cost, then under different combinations of algorithm and mapping, minor changes on parameters are able to make the cost skyrocket. Figures 19 and 20 present several plots for common numbers of cores on modern systems, with a varying amount of machines on the x axis and the total normalized communication cost of Bruck under cyclic and Sparbit under sequential on the y axis. The cost for sending each message is calculated as its size times the distance between the two machines in which the peers are located. This cost formulation is similar to the hop-byte metric, but since the analysis is not based on any topology, the machine distances are utilized as the hops.

The data size obviously contributes to the time taken to send a message and the further away two machines are, the higher the likelihood that they will be on different hierarchical domains and that the message will have to cross more levels to reach the destination, taking up more time. From this, the total cost of each algorithm is calculated by the sum of the cost of messages sent by all processes along all steps. Even though this is not a high fidelity representation of the real cost for the data exchanges, it serves as an useful simplification to understand the effects of unfavourable mappings on the communication time of an algorithm. Here, this model is employed instead of Hockney due to the latter's inability to account communication time variability when crossing hierarchical domains with different depths. Nonetheless, the results of this cost formulation can be related to what would be a two dimensional version of Hockney, where α and β would depend on the peers positioning on the topology. Since α represents a

fixed addition proportional to the distance and β a scalar derived from it, utilizing the distance itself yields overall similar behaviours and the two models can be considered asymptotically equivalent.

The scenario of only one core per machine, showcased on Figure 19a, implies that only one process will be mapped on each node and that all communications will inevitably not be local. On this situation doubling the rank distance also means doubling the distance to the destination machine, consistently degrading the cost with every new increase. This coupled with the rising data sizes makes the communication cost of Bruck to grow with a faster pace than Sparbit, as on the latter most of the heavy communications will happen within close machines and the larger the transported data, the shorter the distance travelled. With two cores on Figure 19b, some local minimums start to appear on Bruck's curve as the number of machines rise. These appear only on amounts of machines that are powers of two and are a direct exemplification of the cases with favourable cyclic mapping. The cost reduces on such points because as the amount of processes is a power of two, all the $p/2$ rings on the last step will have 2 participants that will be placed on the same machine, allowing only local communication to be needed on the final step. The total cost is yet higher than Sparbit's on the vast majority of cases since intermediary steps still cross long distances across the machines while transporting large data, which is only less expressive on very small amounts of machine, thus making Bruck better on those situations. When the amount of machines is not a power of two, Bruck's cost assumes an exponential-like growth powered by the unfavourable distributions – previously discussed and also exemplified on Figure 18a – where heavily communicating neighbors will be placed on different machines, potentially far away from each other.

As the number of cores per machine rises from Figure 19c to Figure 19f, what can be seen is that Bruck's total cost for power of two amounts of machines keeps reducing its growth factor in comparison to Sparbit, starting to reach smaller values than the latter for certain cases. This is the result of the aforementioned higher locality potential of Bruck under cyclic mapping, which as more cores become available is able to fit a higher number of rings from intermediary steps inside a single machine and thus expand the percentage of local communication throughout the execution. As expected, this phenomenon is also present on the plots for high number of cores from Figure 20 and with more cores available, the higher is the probability for Bruck to have a smaller cost on more power of two numbers of machines. For Sparbit, the growth is mainly a product of the increase in the number of communications and data size alone, since under sequential mapping it is not very sensible to either variations on the number of cores or machines, remaining favourable in all cases. On both the plots for low (Figure 19) and high (Figure 20) numbers of cores its cost naturally grows, but in a much smaller factor than Bruck on numbers of machines which are not powers of two.



(a) 1 core.

(b) 2 cores.

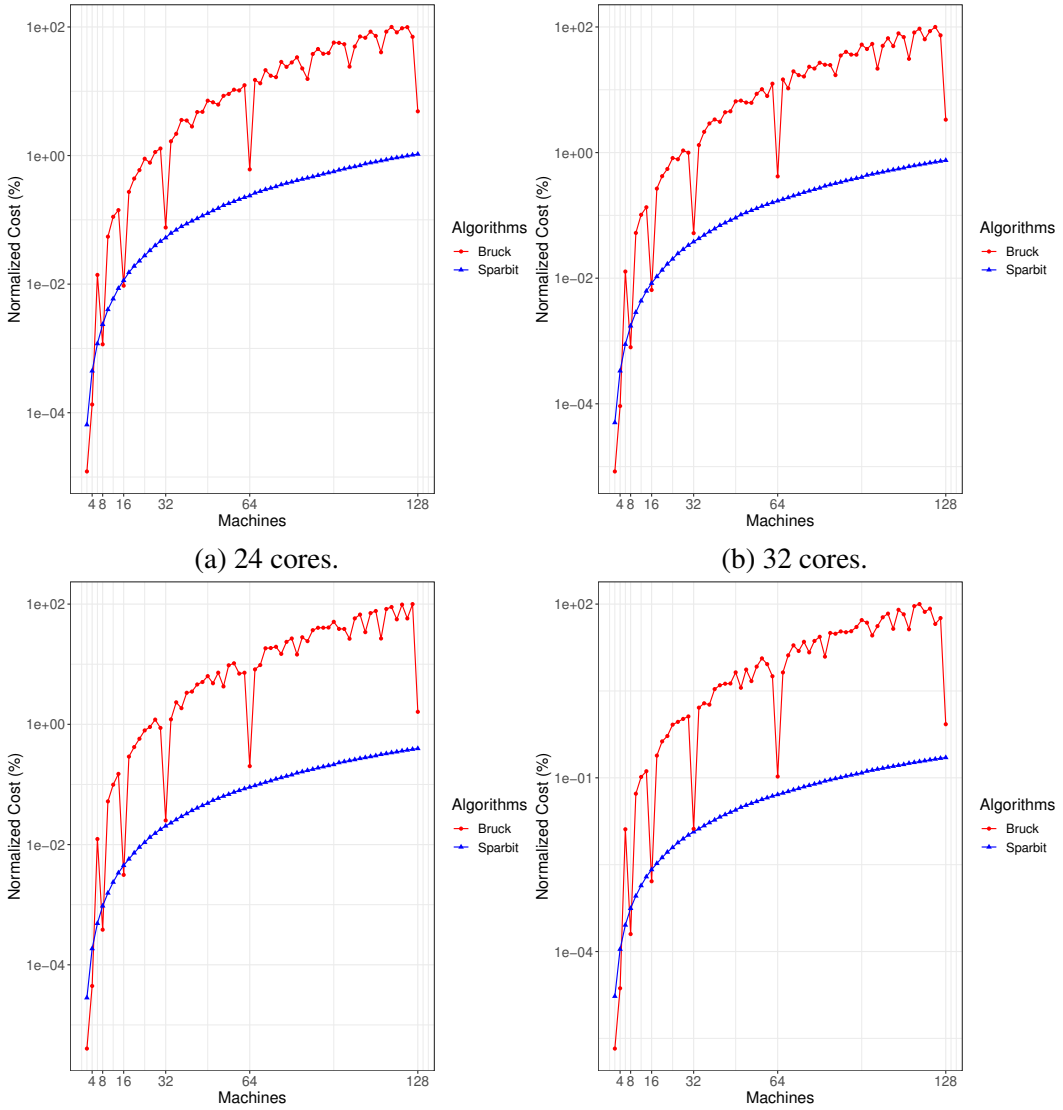
(c) 4 cores.

(d) 6 cores.

(e) 8 cores.

(f) 16 cores.

Figure 19 – Total communication cost of Sparbit under sequential mapping and Bruck under cyclic for small numbers of cores (≤ 16) and varying machines.



(a) 24 cores.

(b) 32 cores.

(c) 64 cores.

(d) 128 cores.

Figure 20 – Total communication cost of Sparbit under sequential mapping and Bruck under cyclic for high numbers of cores (> 16) and varying machines.

5.4 CONSIDERATIONS

This Chapter was dedicated for discussing the relation of the mappings and the algorithms, with focus on Sparbit and Bruck. Initially the two forms of default mapping employed by major MPI implementations were introduced to be the sequential and cyclic. The first is a best-fit approach that fills all slots of a machine before moving to the next, while the second employs a round-robin approach, assigning one rank to a machine and going to the next on a circular fashion. Next, both Sparbit and Bruck were modelled as if each step was composed of a set of disjoint rings that exert the communications for certain groups of processes. This abstraction allowed the mapping problem to be seen as a maximization of nearby placement for the rings with the heaviest communication and, if a mapping scheme was efficient at performing this task, it would also be efficient for the algorithm as a whole.

The modelling discussed that Sparbit's heaviest communication step will be always the last one, while on Bruck it can be either the last or second to last depending on the amount of processes. Furthermore, Sparbit on the last step has only one component ring, which can not be jointly optimized with rings from earlier steps due to process parity conflicts. Bruck has the maximum possible number of rings on the last step, which in general can be co-located with rings from earlier steps inside the same hierarchical domain, allowing it to have a greater locality potential on certain cases. From the complete reasoning presented, Sparbit will always have a considerably favourable rank distribution in all cases when employing the sequential mapping. Bruck on the other hand, will have a favourable rank distribution when employing the cyclic mapping only on numbers of machines that are powers of two with an even p. If none of these conditions are met, then its degree of quality will be highly variable depending on the combination of amounts of cores and machines. These conclusions serve as base for the discussion regarding the results on the experimental analysis.

6 EXPERIMENTAL VALIDATION

This Chapter presents all information related to the experimental tests performed in order to validate the proposal. Section 6.1 presents the methodology, experimental setup and detailed information regarding the execution for each of the Allgather calls. Section 6.2 presents the analysis of the data obtained for the Allgather collective, while Section 6.3 discusses the results and findings for Allgatherv. Section 6.4 briefly presents some works on MPI algorithm selection, which is fundamental for exploiting Sparbit's performance potential. Finally, Section 6.5 presents the considerations for this Chapter.

6.1 METHODOLOGY

Experiments were executed on two HPC infrastructures to measure the performance of Sparbit in comparison to the other Allgather and Allgatherv algorithms. This Section presents the methodology employed to execute the empirical tests along with details of machine specification, benchmarks, and configurations.

6.1.1 Testbed and benchmarks

Two sets of experiments were executed on the Yahoo and Cervino clusters from the University of Neuchâtel, where 16 and 5 dedicated homogeneous machines were employed, respectively. On the Yahoo tests, the network is organized in a two-tier tree topology with 5 machines connected to one Gigabit Ethernet leaf switch, 11 machines connected to another equal one, and both interconnected by a core switch through individual 10Gbps single links. Every machine has 8 processing cores provided by two Intel Xeon L5420 CPUs, each containing 4 cores and 1 thread per core. The available memory on each node is 8GiBs. On the Cervino tests, the network is organized in a flat topology with all nodes connected directly to the same switch via 40Gbps individual links. Every machine has 32 processing cores provided by two Intel Xeon E5-2683 v4 CPUs, each containing 16 cores and 2 threads per core. The available memory on each node is 128GiBs. For both experiments the employed operating system was Ubuntu 20.04.2; the chosen MPI implementation was Open MPI 4.0.3; and the benchmark employed was the popular OSU Micro benchmarks (Network-Based Computing Laboratory, 2021). In Section 6.1.2, we dedicate greater detail into explaining the functioning of the benchmarking suite, which is needed to more comprehensively and easily understand the experimental results.

6.1.2 OSU Micro Benchmarks

The OSU Micro Benchmarks are continuously developed by the Network-Based Computing Laboratory from the Ohio State University and provide several small applications for measuring diverse aspects of MPI, with a greater focus on latency and bandwidth tests. Since these applications are distributed through source code, they can be used to benchmark virtually

any MPI implementation by employing and compiling them with the chosen library. Another advantage of the source-based distribution is that modifications and additions for gathering more valuable data can be made in a simple manner, enriching the completeness of the tests when needed. Due to these characteristics and the ease of usage, the OSU suite has become a widespread choice for scientific works measuring communication performance of MPI.

On this work we employ two of the available applications from OSU on version 5.7, namely `osu_allgather` for measuring Allgather time and `osu_allgatherv` for measuring Allgatherv time. Each of the applications executes the MPI calls a repeated number of iterations in order to alleviate possible measurement skews and errors, consequently making the measures statistically significant. Natively, each of the processes participating on the collective operation locally records the execution time for each iteration, then the final results are created from the combination of all these data. For each individual iteration of the collective operation, either application gathers:

- Minimum time: Smallest time taken by any participating process to finalize that execution of the call.
- Average time: Mean of the times taken by all processes to finalize that execution of the call.
- Maximum time: Highest time taken by any participating process to finalize that execution of the call.

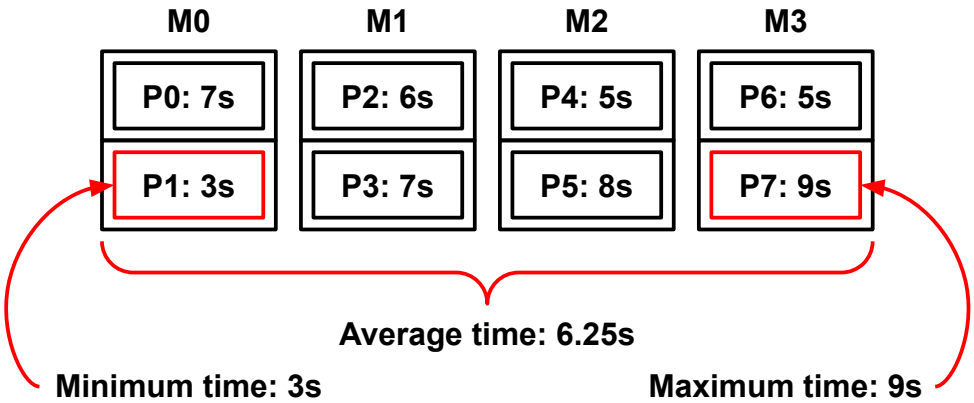


Figure 21 – Example of OSU measured times on one hypothetical iteration with 8 processes over 4 machines.

Figure 21 shows examples of minimum, average and maximum times resulting of an hypothetical iteration of either `osu_allgather` or `osu_allgatherv`. On each iteration the applications will execute the proper collective operation with a defined block size over all processes, here 8 portrayed spread equally over 4 machines. The time taken by each process to finish the operation will be recorded and the minimum, average and maximum times extracted. Once all iterations are over, there are 3 sets of measured times, one for each of the metrics and all

with a population equal to the number of iterations. OSU then outputs the individual unweighed mean of each set, or in other terms, the mean of minimum, average and maximum times across all iterations. Hereafter, whenever we discuss minimum, average or maximum times of a test it refers to the final mean value of all iterations and not to the individual value of one, as depicted on Figure 21. Table 5 presents an illustrative sample output of the benchmarking application on some selected block sizes.

Size	Avg Latency(us)	Min Latency(us)	Max Latency(us)	Iterations
1	45.16	12.99	67.77	50
64	42.31	20.71	58.90	50
4096	148.46	55.16	185.02	50
262144	1926.59	1277.40	2126.15	50

Table 5 – OSU's raw output for excerpt block sizes.

From an internal working perspective, OSU on its native version does not maintain all iteration's recordings on memory, rather each process sums the value of the current one into a dedicated variable and overwrites it with the value of the next one. This allows savings on memory consumption, specially with high number of iterations and processes per machines, with the drawback of not being able to compute the standard deviation for the final mean. Since this is an important information to understand the stability of the algorithms, we have modified the original and `osu_allgatherv` code so that each processes stores the measures of each iteration for the average time, allowing the output of the standard deviation of this metric on the final report of each test. Furthermore, although the Allgatherv call allows each process to send a block with different size, the original `osu_allgatherv` only measures the time with a regular distribution, where all blocks have the same size, limiting the analysis to a subset of possible use cases and forcing a resemblance to the original Allgather call functioning. Therefore, we have also modified this application and added several block size distributions as more thoroughly detailed in Section 6.1.3.

6.1.3 Metrics and configurations

We first present the general information valid for both of the employed OSU applications, which include aspects like the Sparbit's implementation and aggregation into the tests, experiments' parameters and execution as a whole. Then, the focus is shifted specifically to each of the calls, discussing their specific needs and modes of operation to acquire valuable data for each case.

All metrics were collected through the OSU reports, namely minimum, maximum and average time of the Allgather and Allgatherv calls over all runs. The Sparbit algorithm for both calls was developed in C and bundled into an Open MPI collective communication component. The compared algorithms were Bruck, Recursive Doubling, Ring and Neighbor Exchange. Each algorithm was selected for execution through command line arguments passed to the `mpirun`

executable, with native algorithms chosen through the dynamic rules of the `tuned` component, and Sparbit selected by raising its component priority. In order to reduce the probability and effect of any unexpected machine or network events on the time of a test, all algorithms were sequentially executed with both mappings for each number of processes. Since OSU does not output mean values for each of the iterations, we were not able to compute the exact sample size which would yield sufficiently precise 95% or 99% mean confidence intervals. Therefore its value was selected to be arbitrarily high and each test was executed 50 times for statistical significance, with 5 additional warm-up executions to reduce the effects of any possible start-up skews. Although both these parameters were not methodologically chosen, as will be further seen in Section 6.3 the means are overall stable with low coefficients of variation, attesting their validity.

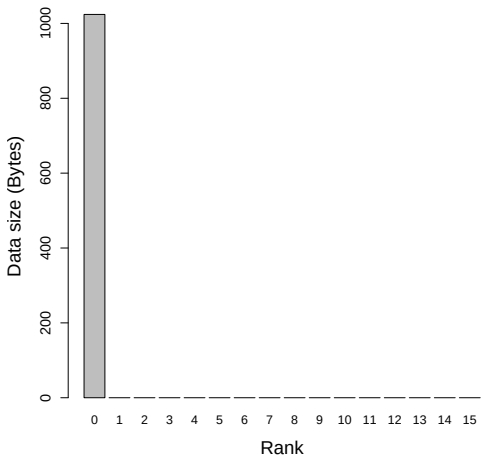
Block sizes were varied following the OSU's default limits, starting at 1B and raising up to 1MiB on successive multiplications by 2, totalling 24 tested data sizes. For Allgather, the maximum number of processes utilized was 256 on Yahoo and 320 on Cervino, respecting a limit of 2 processes per physical core of each machine on both infrastructures. For Allgatherv the same rule applies, however since only 4 Cervino machines were employed on this case, the maximum number of processes was 256 for these tests. The processes were varied following two arithmetic progressions: the first, starting at an even number and ending at 256 or 320 (depending on the case), was employed to represent cases of even and power of two numbers of processes. The second, starting at an odd number and ending at 253 and 317 for Allgather or 249 for Allgatherv, represented cases of odd numbers of processes, where there is asymmetry on the algorithms' execution and on the mapping of processes to the machines.

The step distance of the progressions, which determines what values between the inferior and superior limits will be part of it and also its starting point, was assigned differently on multiple experiment batteries. For the first set of tests, destined to measure the Allgather times on both Yahoo and Cervino, a step distance of 8 was employed and thus odd and even progressions started at 5 and 8, respectively. This choice results in a greater quantity of process numbers inside both progressions, allowing a more comprehensive analysis although significantly increasing the execution times. The tests' durations by themselves on each infrastructure clearly exemplify this negative aspect, totalling almost 20 days for Yahoo and 5 for Cervino. From this, when considering the second set of experiments, which collects data for Allgatherv, the step distance was set to 16 and so reduced by half the amount of tested process numbers, with the odd and even progressions starting at 9 and 16, respectively. The modification was made to alleviate the time needs, specially since the block size distributions add a new dimension when testing Allgatherv. The 8-stepped experiments generate 64 (80 for Cervino) different process numbers, resulting in 1344 (1680) individual tests from combinations with block sizes. The 16-stepped tests generate half the process numbers and therefore also half of the total tests. For both collectives, each singular combination of the previous parameters was executed employing both sequential and cyclic mappings, allowing the empirical validation of the discussion of Chapter 5. All

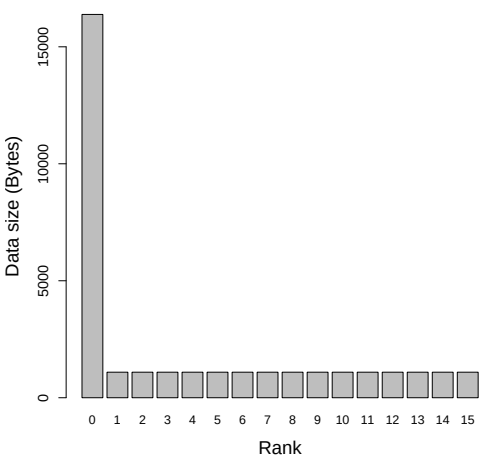
`osu_allgather` tests rely only on the parameters mentioned so far. The `osu_allgatherv` tests however also rely on the block size distribution, which allows each process to have a differently sized block to send to its peers. We employ 5 size distributions extracted from the work of Träff et al. (2008) but with some modifications. Considering c an arbitrary block size, m_i the size of the block on process with rank i and $m = \sum_{i=0}^{p-1} m_i$ the total amount of data to be exchanged, the distributions are defined as follows:

- **Broadcast:** Only one process has data, therefore $m_0 = c$, $m_i = 0$, with $0 < i < p$ and the total is $m = c$.
- **Spike:** A mix between Regular and Broadcast distributions, all processes contribute but one has more data. Therefore $m_0 = pc$, $m_i = \frac{pc}{(p-1)}$ and the total is $m = 2pc - \varepsilon$, with $0 \leq \varepsilon \leq pc \ mod \ (p-1)$.
- **Half full:** Even ranked processes have data and odd ones do not. Thus $m_{2\lfloor i/2 \rfloor} = 2c$, $m_{2\lfloor i/2 \rfloor + 1} = 0$ and the total is $m = 2\lceil p/2 \rceil c$.
- **Linearly decreasing:** Sizes decrease linearly with the growth of ranks. Thus $m_i = 2c\frac{p-1-i}{p-1}$ and the total is $m = pc - \varepsilon$, with $0 \leq \varepsilon \leq 2c \ mod \ (p-1)$.
- **Geometric curve:** Sizes decrease according to the harmonic series $(1/i)$. Thus $m_i = \frac{pc}{(i+3/2)logp}$ and the total is $m = pc - \varepsilon$, with $0 \leq \varepsilon$ and $\lim_{c \to +\infty} \varepsilon \ll pc$.

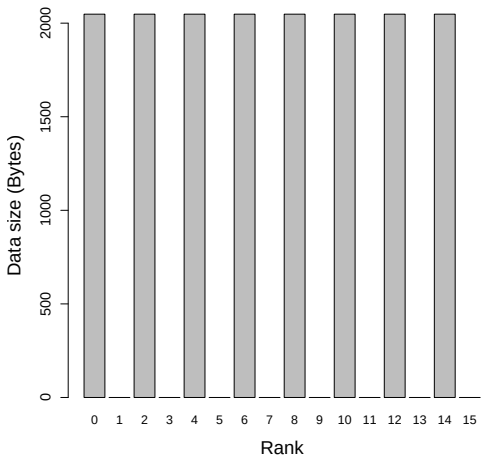
The ε *error* term that appears on some of the distributions is due to discretization losses along the process that add up and can reduce the total combined size that must be exchanged. This reduction however is bounded to a fixed maximum value which is far smaller than the individual sizes or the total, specially as data and process numbers rise. Hence, even with the presence of such small deviations, the general expected behaviour of the distributions is still maintained. Figure 22 presents a barplot for each of the employed Allgatherv distributions showing individual block sizes for all processes. Here, p is set to 16 and c to 1KiB for presentation reasons and to visually allow the comparison between all distributions.



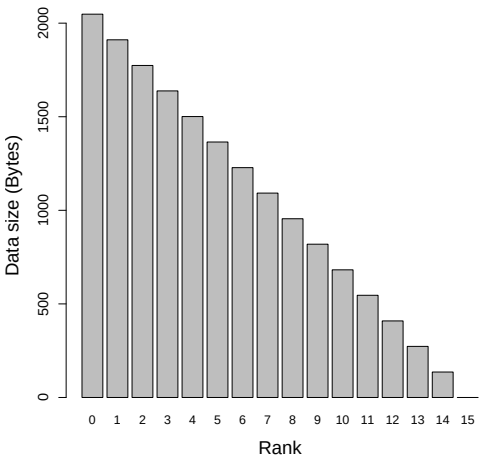
(a) Broadcast.



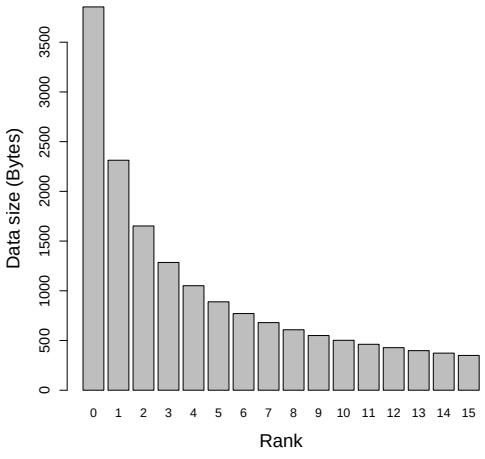
(b) Spike.



(c) Half full.



(d) Linearly decreasing.



(e) Geometric curve.

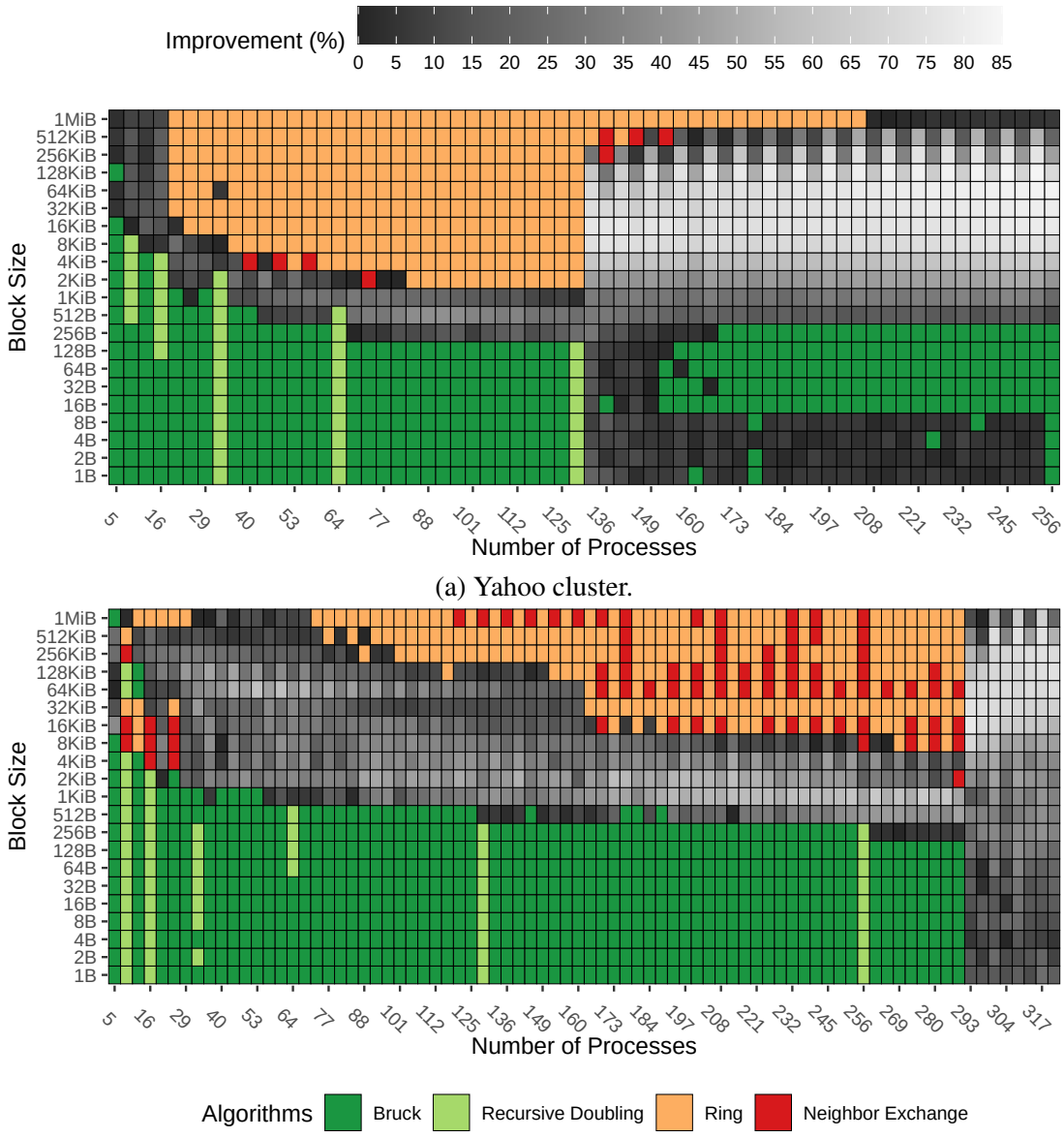
Figure 22 – Barplot of block size on each rank for 16 processes (p) and 1KiB base size (c) under different distributions.

6.2 ALLGATHER ANALYSIS

In this Section we present the results obtained from the experiments and the analysis of the data. Initially, four hybrid heat maps with a broad view of the experiments are provided on Figures 23 and 24, two for sequential and two for cyclic mapping, showing Yahoo and Cervino cases. Every cell of each heat map indicates a particular test configuration, comprising a unique combination of block size and number of processes. Each row indicates one block size from 1B to 1MiB and each column indicates one process amount, with both arithmetic progressions intertwined and starting from the odd one (as discussed in Section 6.1.3). For each individual combination, the average time of all algorithms on that mapping is sorted and the smallest one is taken to produce the result contained on the cell. The heat maps are said to be hybrid because they present both categorical and continuous values: if Sparbit does not have the smallest time on a test, the color of the cell indicates what traditional algorithm was the best (*e.g.*, the dark green color for 1B and 5 processes on Figure 23b indicates that Bruck was the best on this configuration). In turn, if Sparbit has the smallest time, a lighter is better greyscale color indicates its improvement percentage over the second best algorithm – *e.g.*, 23.83% improvement for 1MiB and 320 processes on Figure 23b. This percentage value of a cell is calculated by dividing Sparbit's time by the second smallest one, with disregard to which algorithm achieved it. So, if the best traditional MPI algorithm had concluded the communication with an average of 10 seconds while Sparbit did the same in only 5, then the portrayed improvement on the cell would be of 50%. All the heat map values are related to the smallest average time over 50 executions, as outlined in Section 6.1.2.

6.2.1 Network topology and data locality impact

Most of the results that can be seen on Figures 23 and 24 corroborate with the previous discussions on Hockney-based algorithm costs presented in Section 2.4. Under sequential mapping on both infrastructures, Sparbit has the smallest execution time on a large portion of the tests, reaching 46.43% (624/1344) on Yahoo and 39.64% (666/1680) on Cervino, as seen on Figures 23a and 23b, respectively. Under cyclic mapping there is a considerable drop on Sparbit's percentage, as expected from its worse communication pattern on this case, reaching 19.12% (257/1344) on Yahoo and 30.83% (518/1680) on Cervino, as seen on Figures 24a and 24b, respectively. The different topologies of the infrastructures give valuable insights on its impact over Sparbit and the results itself. First, Yahoo has a larger topology with more levels crossing switches, which imply on different communication costs among distinct pairs of machines. Cervino, on the other hand, has a flat topology which implies on equal theoretical communication costs among any two machines, leaving only the internal machine communication hierarchy and its difference to the external one. Therefore, on Yahoo with sequential mapping we can expect the highest benefit from employing Sparbit, which can be seen true by the largest fraction of the best times achieved on this experiment. The smaller topology of Cervino reduces



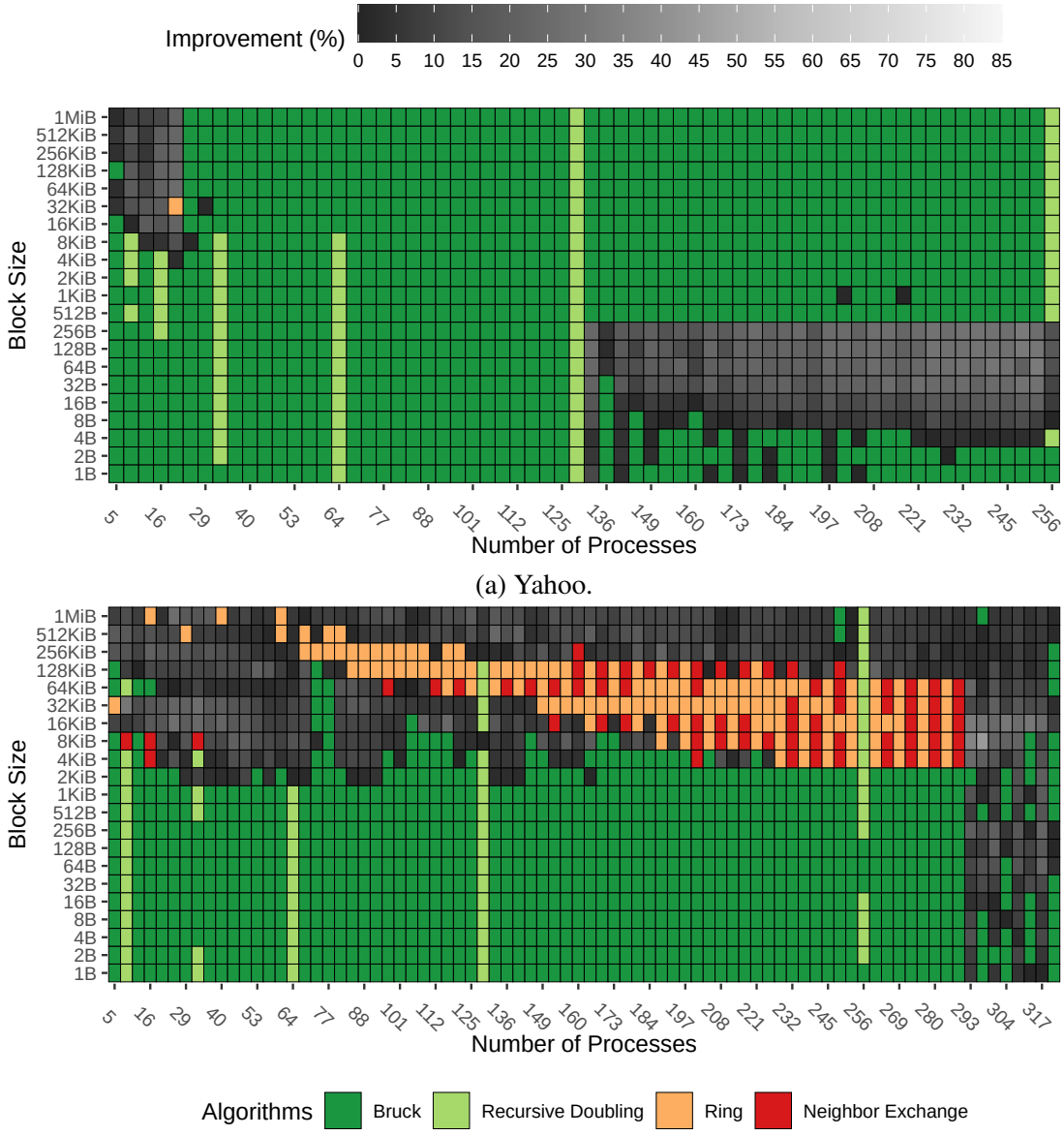
(a) Yahoo cluster.

(b) Cervino cluster.

Figure 23 – Hybrid heat maps for sequential mapping displaying a discrete color for best average time traditional algorithm or Sparbit's improvement over the second as a greyscale if it is the best.

the maximum potential benefit of Sparbit, which correspondingly reduces its fraction of the best times. Under cyclic mapping, Sparbit's drop in performance is expected to be proportional to the topology's size, since now there is an inverted communication pattern. This observation justifies the drop in relation to the sequential mapping seen on the Yahoo cluster to be larger than the one seen on Cervino, as the negative impact of the former's topology is higher than the latter. From all this we can state that the larger the topology's hierarchy, the larger the potential benefit of using Sparbit under sequential mapping, and the larger its benefit's reduction when switching to cyclic.

Another factor that complementarily justifies the higher drop on performance from sequential to cyclic on Yahoo is the relation of the algorithms with the mapping, discussed on



(a) Yahoo.

(b) Cervino.

Figure 24 – Hybrid heat maps for cyclic mapping displaying a discrete color for best average time traditional algorithm or Sparbit's improvement over the second as a greyscale if it is the best.

Chapter 5. From this infrastructure were employed 16 machines, which is a power of two amount and therefore maximizes the potential performance of Bruck under the cyclic mapping. When using the sequential distribution, where Bruck's locality is inescapably poor, Sparbit achieves a significant number of cases as the best. But when turning to cyclic, beside Sparbit's locality being degraded, Bruck achieves its higher potential due to the mapping and takes over several cases as the best. The same type of rationale also applies to Cervino, but in the opposite way, now the amount of employed machines was 5, which is not a power of two and is suboptimal to Bruck under cyclic. This coupled with the infrastructure characteristics allow the change on mapping to be less harsh for Sparbit, which is able to remain the best on a larger fraction of cases than on Yahoo.

6.2.2 Sparbit and other algorithms

The form of the area taken by the Sparbit's best cases (Figures 23 and 24), or the particular intervals where it is the best, assume these uncommon shapes due to the cropped nature of the heat map. If we individually compare the area of smaller groups of algorithms with Sparbit, like the ones with linear or logarithmic complexity, then more comprehensible forms appear. Through this rationale, Sparbit's form on Yahoo and Cervino sequential heat maps are the product of the same phenomenon. Considering Yahoo with sequential mapping and stripping Bruck and Recursive Doubling out of the plot, their area would be almost completely taken by Sparbit, absorbing 95.48% of the cases as the best, with Ring and Neighbor Exchange absorbing only the remaining percentage. If we instead stripped out Ring and Neighbor exchange while maintaining the rest, a similar pattern would emerge with Sparbit taking 100% of their area as the best. These observations show how there is a clear improvement pattern of Sparbit towards each of the algorithm groups. In relation to Ring and Neighbor Exchange, Sparbit is better for large data sizes (hereafter considered the ones greater than 1KiB) on large number of processes and on almost all tests (97.44%) for smaller sizes (hereafter considered the ones smaller or equal than 1KiB). In relation to Bruck and Recursive Doubling, Sparbit is better for large data sizes also on almost all tests (97.16%). Therefore on the merged final version, which is shown on Figure 23a, for 128 processes or less Sparbit excels on the verge sweet spot of both linear and logarithmic classes of algorithms, where the data size is to small to compensate the additional steps of the former and to large to compensate the non-local communication of the latter. Above 128 processes, which can be seen as a behavioural turning point, Sparbit takes over the majority of the large data sizes as the best - since it is the case on both the stripped versions - and a fraction of the small ones as well.

The same general pattern of Yahoo under sequential mapping can also be seen on Cervino. The difference is that the verge sweet spot for each number of processes is larger (*i.e.*, involves more test cases) and the behavioural turning point appears on 288 processes. We assume that theses abrupt changes of behaviour are due to the machines entering an overbooked scenario where there are more process preemptions. On Yahoo, the first load where there are more processes than cores is with 133, and Cervino by having two threads can suppress this problem until higher ratios, but also fails before reaching 2 processes per core. This negatively affects all algorithms but is less severe on Sparbit, particularly on large data sizes, suggesting that it has less trouble when operating on restricted or high utilization CPU environments.

Under cyclic mapping, Ring and Neighbor Exchange also suffer a drop in performance, since their neighbor ranks on the sequential mapping are now placed at other possibly far machines, which results in a very non-local communication pattern. Regarding Sparbit, on Cervino with cyclic mapping the area where it is the best results from a similar merging phenomenon as discussed on the sequential mapping. Here, the difference is that it is better on more cases of large data sizes previously taken by Ring and Neighbor Exchange, which are

surpassed due to their more expressive drop on performance. On the other hand, it also loses more cases of both small and large data sizes to Bruck, since the cyclic mapping for Sparbit has an even larger locality degrading effect then the unfavourable state of the latter. On Yahoo with cyclic mapping, the form taken by Sparbit as the best virtually becomes a product of its relation with Bruck and Recursive Doubling, as it would on some of the previously discussed stripped versions of the plot. Sparbit is the best on the small top left area due to its use of parallel sends, employing more available bandwidth sooner, thus resulting in smaller communication time.

6.2.3 Sparbit's time reduction

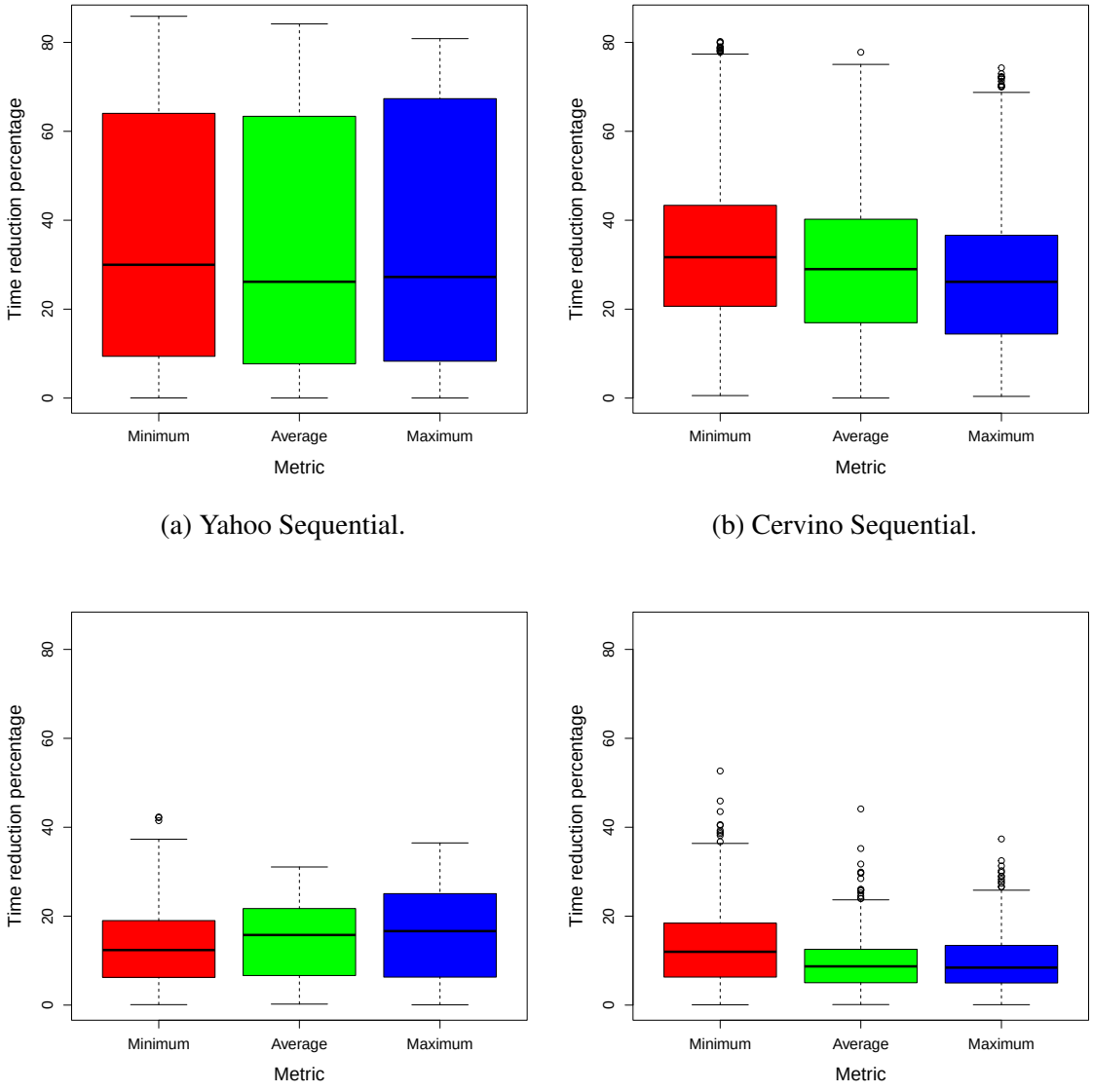
So far, the analysis has only focused on the number of cases in which Sparbit performed better than other algorithms, without discussing the actual improvement obtained on the tests. Table 6 presents the percentage of cases where Sparbit was the best and also summarizes the mean, median and highest percentage improvement values over the second algorithm. We also refer back to the heat maps of Figures 23 and 24 in order to discuss the patterns of improvement for the proposal.

Mapping	Cluster	Metric	Best in	Mean	Median	Highest
Sequential	Yahoo	Min	53.65	36.18	29.98	85.87
		Avg	46.43	34.7	26.16	84.16
		Max	38.17	35.48	27.24	80.85
	Cervino	Min	47.02	33.08	31.69	80.2
		Avg	39.64	30.23	29	77.78
		Max	27.62	27.67	26.14	74.27
Cyclic	Yahoo	Min	20.68	13.35	12.38	42.31
		Avg	19.12	14.89	15.77	31.07
		Max	21.80	16.07	16.64	36.47
	Cervino	Min	34.94	13.49	11.99	52.66
		Avg	30.83	9.6	8.71	44.12
		Max	25.24	9.71	8.43	37.36

Table 6 – Sparbit improvement of metrics over the second best algorithm on Cervino and Yahoo under sequential and cyclic mappings.

The first clear characteristic, as expected, is that the improvements under sequential mapping are much more significant and consistent than the ones under cyclic mapping. This is outlined by a greater occurrence of darker grey tones for the plots of the latter on both infrastructures and is once again due to its unfavourable process distribution. The same phenomenon can also be seen on the Boxplots of Figure 25 and numerically on Table 6, as all the reduction distributions are smaller and the metrics present the smallest relative values on the cyclic side when compared to the sequential tests. Focusing on the sequential mapping, another interesting feature is that the highest improvements appear generally on large number of processes and data sizes. For Yahoo, above 128 processes and between 1KiB and 256KiB (287 tests), the mean (median) improvement is 60.64% (69.16%). On Cervino, above 168 processes and on the same

data size interval, the mean (median) improvement is 43.86% (45.65%). The fact that on such cases the median is higher than the mean indicates a distribution that is asymmetric to the left, and thus has more larger improvements than lower ones. This difference is also higher on Yahoo, which corroborates the potential for growing benefits on larger topologies.



(a) Yahoo Sequential.

(b) Cervino Sequential.

(c) Yahoo Cyclic.

(d) Cervino Cyclic.

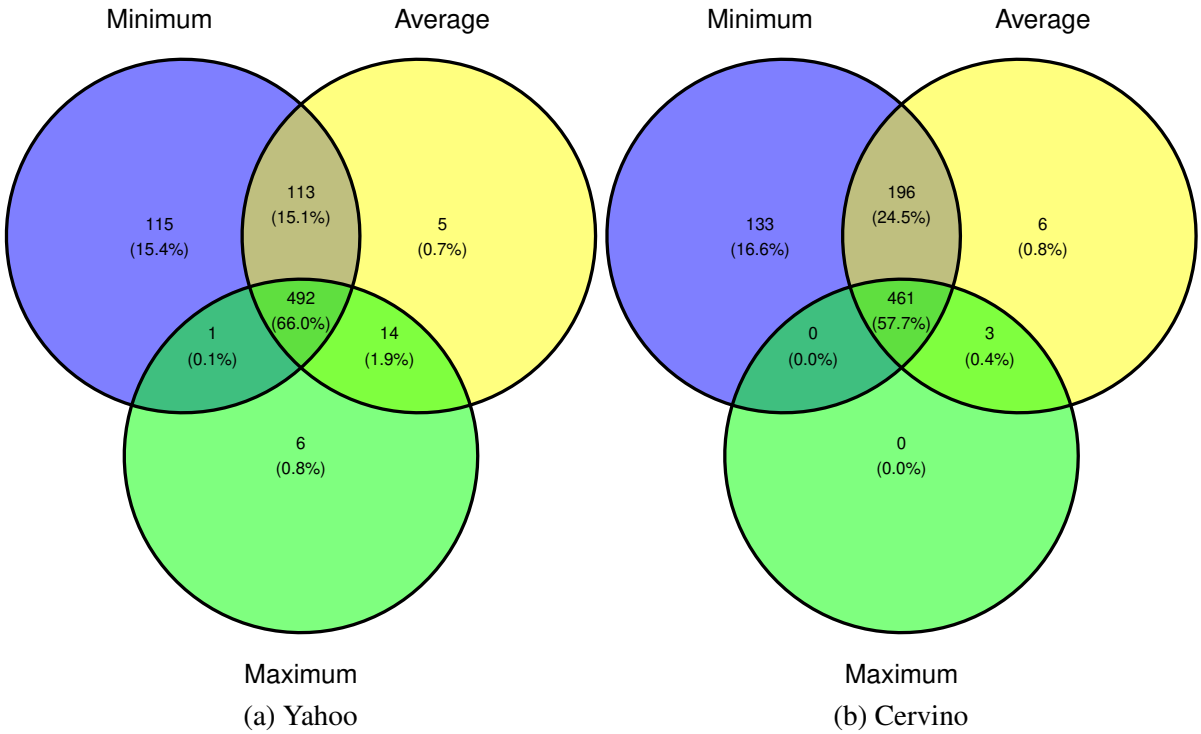
Figure 25 – Boxplots of Sparbit's Allgather time reduction over the second best algorithm for different infrastructures and mappings.

On a general picture the improvements are smaller but still very significant, as can be seen on Table 6. Under sequential mapping Yahoo and Cervino values are quite similar, with means for the average time improvement respectively residing on about 35% and 30%, largest near 80% on both scenarios and average minimum as well as maximum times following similar trends. Under cyclic mapping, the improvements for average time are reduced on both infrastructures, achieving around 15% on Yahoo and 10% on Cervino. On this case, minimum and maximum metrics also

follow similar behaviours, but the highest obtained improvements are more variable across all measurements, ranging from 31% to 52%. Returning to the Boxplots of Figure 25, it is visible that the data dispersion is high on all scenarios, specially under the sequential mapping, ranging from near zero to near 85% in some cases. This is nonetheless expected, as the performance of all algorithms is highly variable depending on the block size and amount of processes, so will be the differences between them, resulting in highly variable relative improvements.

6.2.4 Smallest minimum and maximum times

The additional metrics for smallest minimum and maximum times are discussed in relation to the previous considerations. The forms taken by Sparbit as the best on minimum and maximum heat maps are very similar to the ones taken on the already shown plots of average. Therefore, instead of presenting new heat maps we recur to Figures 26 and 27, which present for both mappings the relations amongst the minimum, average and maximum sets comprising cases where Sparbit has the best time.

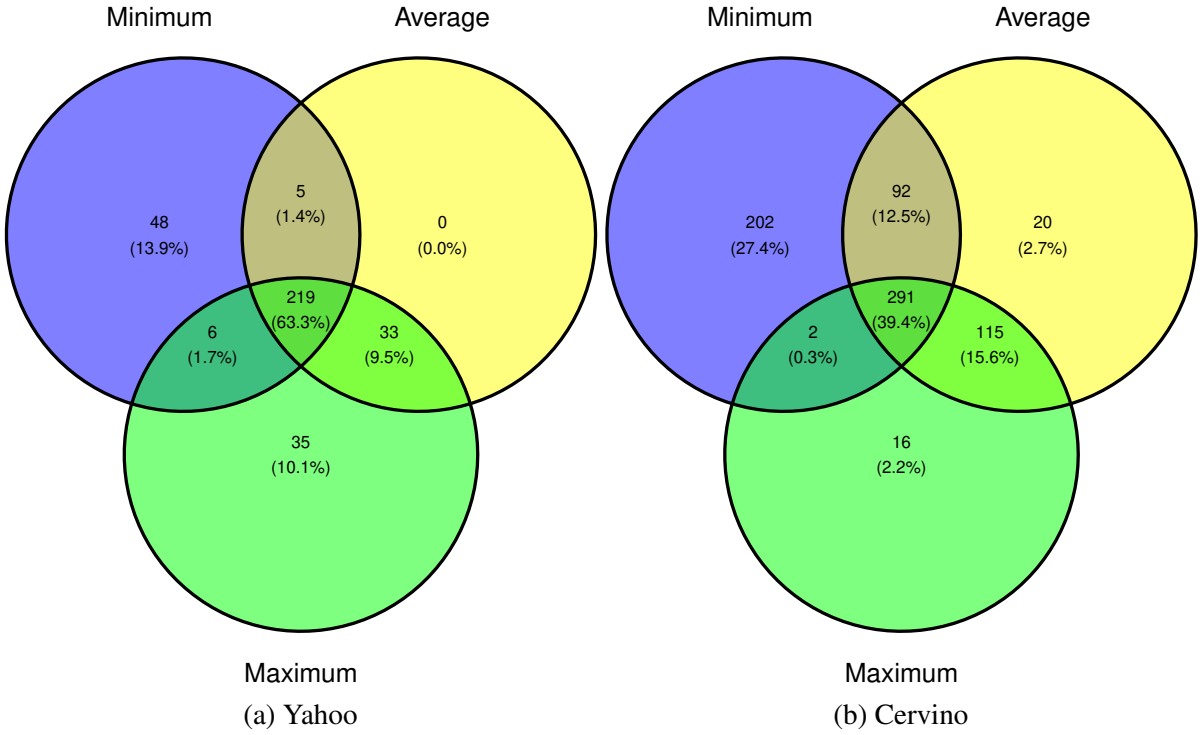


(a) Yahoo (b) Cervino

Figure 26 – Venn diagrams of the relation of Sparbit's cases as the best on minimum, average and maximum time sets for sequential mapping.

Initially, focusing only on the sequential mapping presented on Figure 26, Sparbit has either the minimum, average or maximum best time on 746 (55.51%) cases for Yahoo and 799 (47.56%) for Cervino. An important behaviour to notice is that for both infrastructures on this mapping there is little to no cases of exclusively average or maximum best times. This indicates that these metrics are somehow correlated and that on the majority of cases where it has the smallest maximum time, it also has the smallest average time – 98.64% (506/513) on

Yahoo and 100% on Cervino. On both infrastructures the largest set of Sparbit's best times is the minimum, which also accounts for the largest exclusive number of cases and intersections with the average. This indicates that under sequential mapping it is easier for Sparbit to lower the minimum and average times than it is to lower the maximum ones. Such observation is further based on the decreasing amount of best cases from minimum to average and then maximum sets. The intersection of the three sets accounts for 36.61% (492/1344) of the total tests on Yahoo and 27.44% (461/1680) on Cervino, indicating that on these fractions it performs better than the other algorithms in a consistent way.



(a) Yahoo (b) Cervino

Figure 27 – Venn diagrams of the relation of Sparbit's cases as the best on minimum, average and maximum time sets for cyclic mapping.

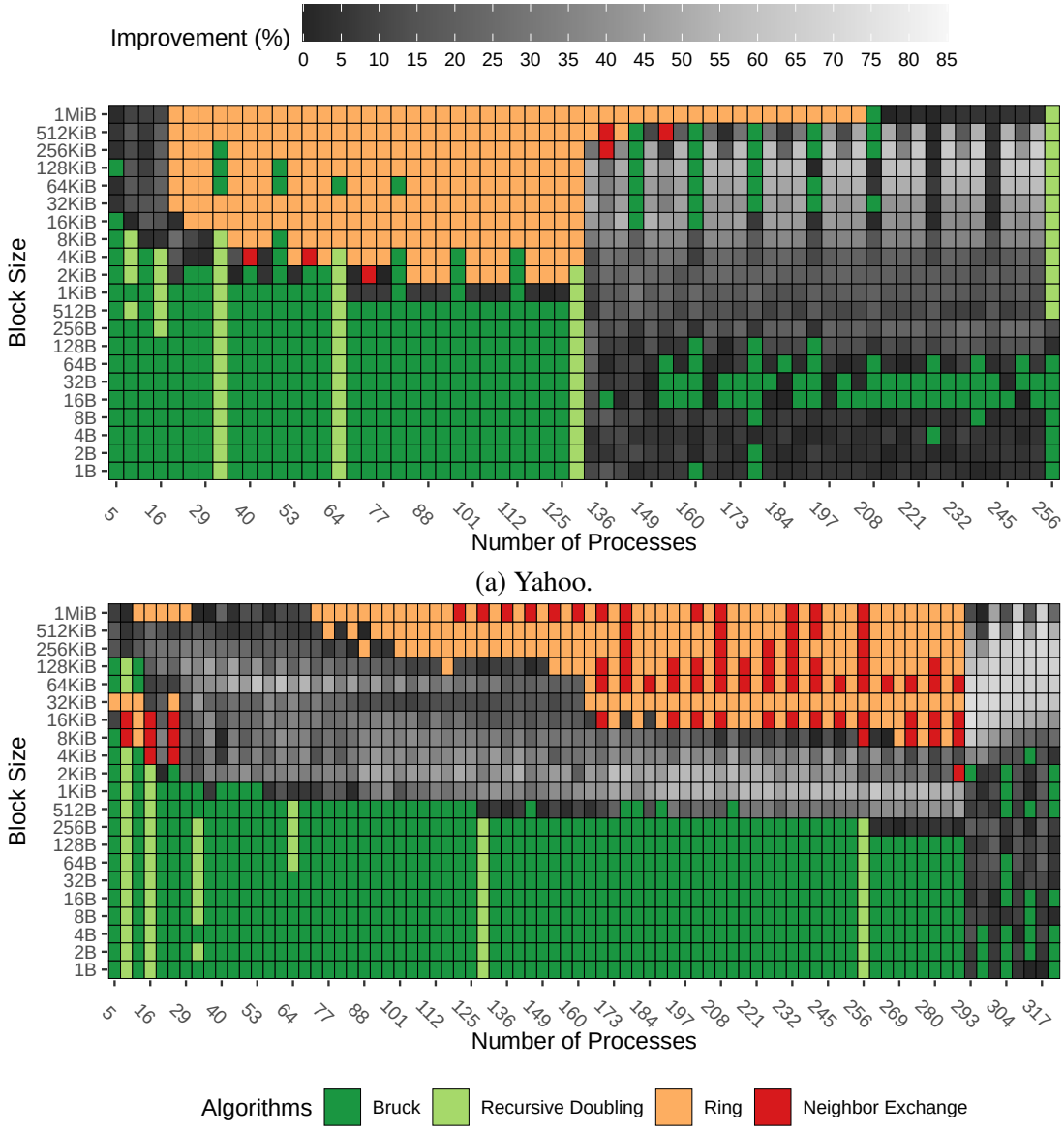
Under cyclic mapping, presented on Figure 27, the total number of cases where Sparbit is the best in any of the metrics drops to 346 (25.74%) for Yahoo and 738 (43.93%) for Cervino, which follows the decay pattern seen previously in detail for the average. Here, the cases are more evenly spread along exclusive and binary intersections, especially on Cervino. The average and maximum metrics correlation still exists but weaker, with 86% (252/293) and 95.75% (406/424) of the maximum best cases also being the best average ones for Yahoo and Cervino, respectively. The minimum exclusive cases are very expressive, probably due to the fastest and more capable network that amplifies Sparbit's ability to reduce the minimum metric. There is no clear ordering of set sizes that applies to both infrastructures under this mapping and the intersection of all metrics drops to 16.29% (219/1344) of the total cases for Yahoo and 17.32% (291/1680) for Cervino.

6.2.5 Overall best times

The results for best algorithm on each test have been presented and analysed under the division of the two default mappings. This allows a more detailed discussion of the relation between all the available options with the focus directed to each of the specific scenarios. On the flip side, this approach neglects the possibility of cross-comparing the results on an absolute basis. Thus, figuring out what is the smallest overall time for a test with disregard for the mapping employed is not possible. In order to allow this kind of analysis, Figure 28 provides two new heat maps, one for Yahoo and one for Cervino, following the same presentation rationale already outlined on the beginning of Section 6.2. The difference now is that instead of every cell presenting the algorithm with the smallest time for only one mapping, it shows the smallest time for the union of both sequential and cyclic. On the previous plots each cell was extracted from a unique set of 5 test records, one for each algorithm, while on the novel plots – which are half of the total of previous ones – each cell is the best result from a unique set of 10 test records, two for each algorithm comprising its executions on the two mappings. Again, the hybrid nature of the heat maps is maintained and a discrete color scale is employed to show the best algorithm if it is not Sparbit, otherwise a continuous greyscale in which lighter is better is employed to show Sparbit's improvement over the second best algorithm. The driving force behind such analysis is to verify in absolute terms which algorithm is the overall best for each test and whether it is better to adopt a sequential or cyclic mapping for obtaining the minimum achievable time on each parameter combination.

The first aspect to notice on the plots of Figure 28 is that their shape is very reminiscent of the plots on Figure 23, which portray the best times for the sequential mapping alone. In fact, from a best option perspective, algorithms which perform best under sequential mapping, such as Ring, Neighbor Exchange and Sparbit, will hardly enlarge their area when mixing with the cyclic side. With a few exceptions on Sparbit, in general the observed pattern is their original shape to be roughly maintained but being taken over in several tests by the algorithms that perform best under cyclic mapping, namely Bruck and Recursive Doubling. An example of such on Figure 28a is the area below 128 processes, where Sparbit was originally the best on the edge between traditional logarithmic and linear algorithms. This area is still maintained, but is a lot thinner and unstable specially from the bottom side, since now Bruck and Recursive Doubling have better times on this region due to their cyclic executions. Thanks to the same reason, both also take up cases previously dominated by linear algorithms and on large number of processes also surpass Sparbit in a couple of its original best cases. Even with these degradations, Sparbit is still the best on 44.49% (598/1344) of the total tests, achieving about 95.83% of the amount of tests where it was the best on the sequential side only (624 tests).

On the Cervino plot, presented on Figure 28b, the shape is even more similar to the one obtained for the sequential mapping displayed on Figure 23b. For this case, the area of the algorithms that are better on the sequential mapping is almost fully maintained, with Sparbit



(a) Yahoo.

(b) Cervino.

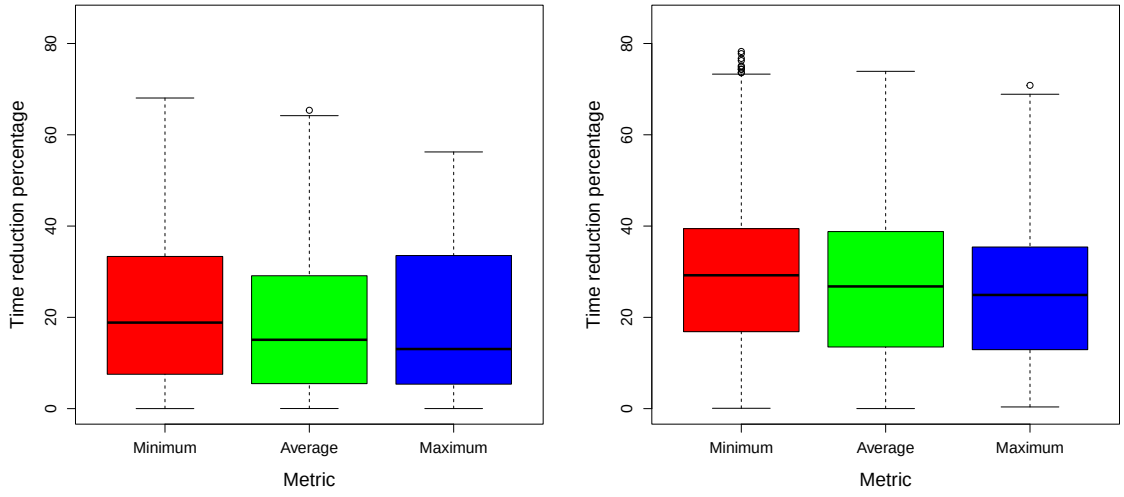
Figure 28 – Hybrid heat maps for the mappings united displaying a discrete color for best average time traditional algorithm or Sparbit's improvement over the second as a greyscale if it is the best.

reaching 38.04% (639/1680) of cases as the best and thus achieving 96.09% of the amount of cases that it had on the sequential mapping alone (665 tests). It loses few tests as the best only below 13 and above 288 processes, while Ring and Neighbor Exchange lose virtually none. Not only the shape is more similar to the previous sequential plot on Cervino, but also the shades of these areas and the improvement on each case. The reason that makes Sparbit's sequential time less prone to be surpassed or even reached on this case is its coupling with the sequential mapping and the stability achieved by their synergy. As detailed in Chapter 5, the rank distribution presented by the sequential mapping is very favourable to Sparbit and not significantly affected by changes on the numbers of cores or machines. Bruck under the cyclic mapping will fail to achieve comparable times due to its unfavourable setting of ranks, rendering

Sparbit's time practically untouched both on area and relative improvement. These results unfold from the current scenario because the number of machines is 5, possibly also extending similar characteristics for any amount which is not a power of two. Using the mapping reasoning on Yahoo, both the form and shades of Sparbit's final area were consistently more degraded than on Cervino, since now Bruck's rank distribution is more favourable and allows it to reach or get very close to the former's original sequential time.

Mapping	Cluster	Metric	Best in	Mean	Median	Highest
Overall	Yahoo	Min	44.79	22.12	18.85	68.06
		Avg	44.49	19.78	15.08	65.37
		Max	38.17	18.64	13.06	56.24
	Cervino	Min	45.58	30.04	29.22	78.26
		Avg	38.04	27.94	26.77	73.9
		Max	29.29	25.86	24.91	70.83

Table 7 – Sparbit improvement of metrics over the second best algorithm on Yahoo and Cervino for sequential and cyclic mappings united.



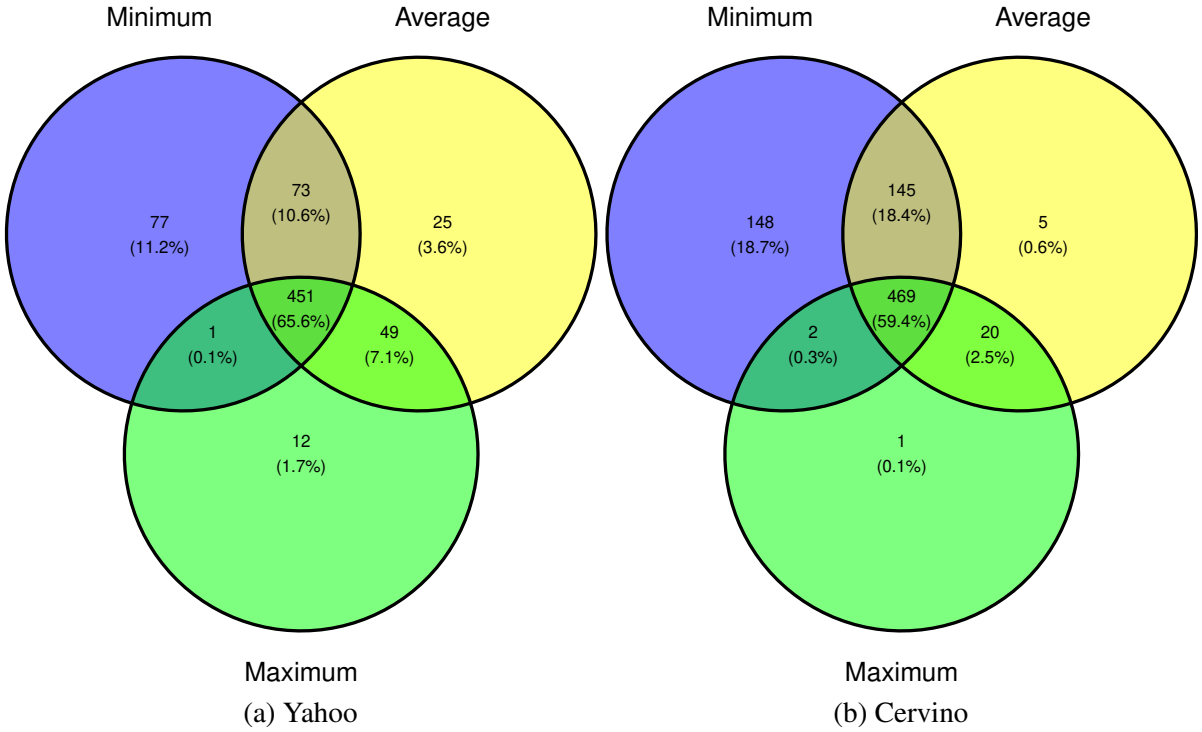
(a) Yahoo.

(b) Cervino.

Figure 29 – Boxplots of Sparbit's Allgather time reduction over the second best algorithm with sequential and cyclic mappings united for each infrastructure.

Regarding the execution time improvements, Figure 29 presents Boxplots of the time reduction percentage values, while Table 7 presents the percentage of cases where Sparbit was the best along with the mean, median and highest percentage improvements of Sparbit over the second best algorithm. Naturally, there is a drop in percentage improvement since now in many cases the second best time on a test, originating from the cyclic side, will be smaller than the one previously available only on the sequential mapping, thus reducing the performance gap towards Sparbit. On Yahoo the drop is very expressive, with the mean falling on all metrics from around 35% to about 20%, with the drops on the median following a similar pattern, still maintaining

the curve asymmetric to the right. On Cervino the drop is much smaller, with the mean values on all metrics still residing close to the original 30% improvement and median of around 25% to 30%, similar to the sequential mapping alone. The reductions on the highest achieved percentage accord to the general pattern of both infrastructures, falling more on Yahoo than on Cervino. The drops can also be clearly seen on the Boxplots of Figure 29, since all quantiles and the median are consistently smaller than the ones of the sequential mapping alone, presented on Figures 25a and 25b. The overall decreases are also in line with all the previous discussion presented in this Section and in Chapter 5, regarding the relation of the mappings and algorithms. Finally, the main conclusion that can be drawn from Table 7 is that on both infrastructures about 40% of all Allgather executions could currently benefit from about a 20% mean improvement just by employing Sparbit.



(a) Yahoo (b) Cervino

Figure 30 – Venn diagrams of the relation of Sparbit's cases as the best on minimum, average and maximum time sets for sequential and cyclic mappings united.

The heat maps for minimum and maximum best times have once again very similar forms to the ones presented for the average on Figure 28. Thus, Figure 30 presents a Venn diagram for the union of sequential and cyclic times, displaying the relation among minimum, average and maximum cases where Sparbit is the best. The first and most important characteristic is that the amount of cases where Sparbit jointly achieved the minimum, average and maximum best times has remained very close to the value for the sequential mapping alone, reaching about 91.67% on Yahoo while on Cervino it even increased a slight amount. As Sparbit's area of smallest average time changed very little from sequential to the united heat maps and most intersection cases of the three metrics have remained equal, we have indication that in general Sparbit also maintained a large area of minimum and maximum best times originating from the sequential mapping alone.

The higher improvement degradation effect seen for the united average on Yahoo in relation to Cervino can also be observed for these metrics, since the former presents a larger reduction in the number of both intersection and total cases with best time when compared to the latter.

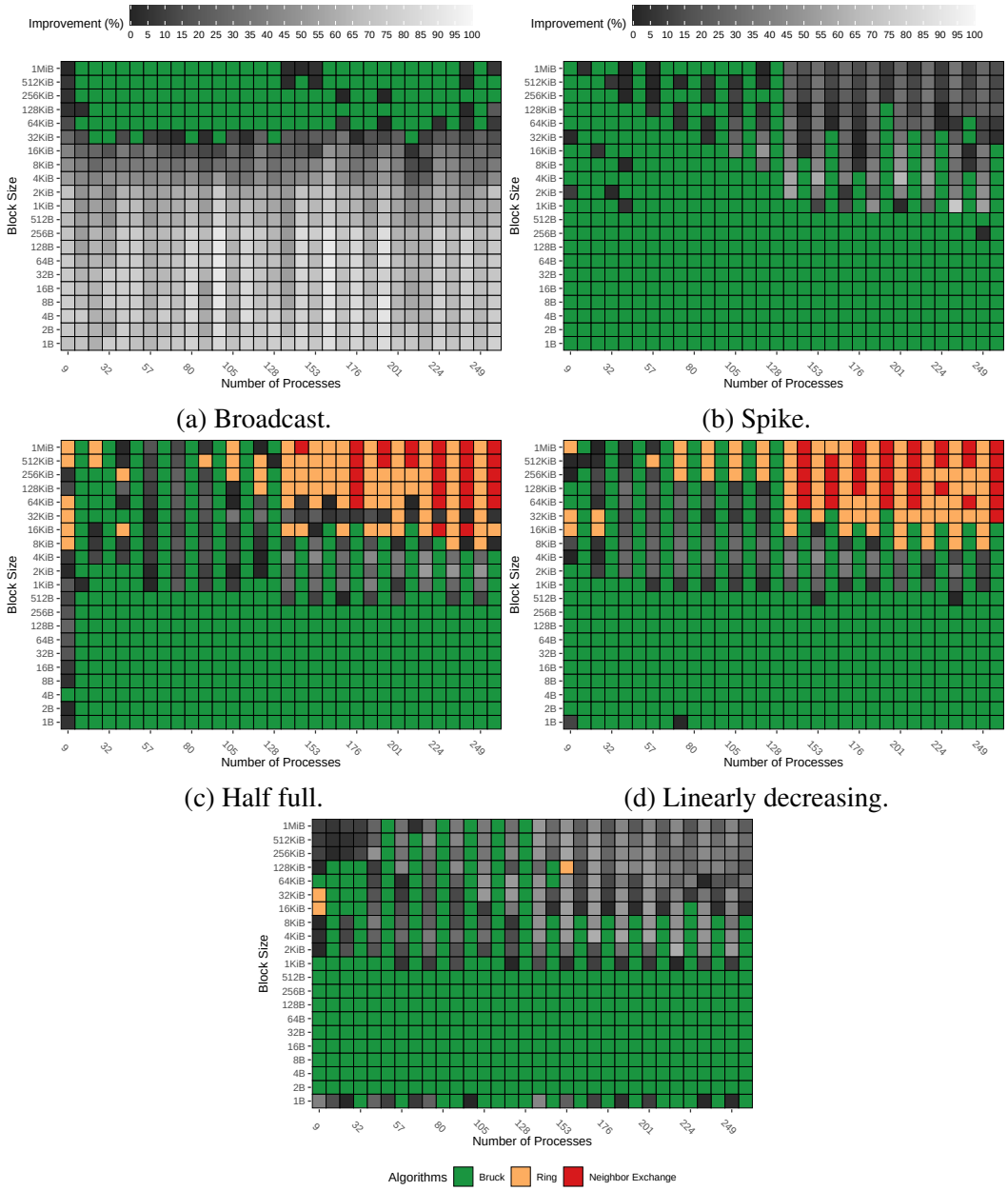
6.3 ALLGATHERV ANALYSIS

This Section devotes itself to presenting the results and analysis of the Allgatherv experiments, as previously described in Section 6.1.3. In comparison to the Allgather tests there is one additional dimension to Allgatherv, which is the block size distributions. There were employed five of such possible distributions adapted from the literature, namely Broadcast, Spike, Half Full, Linearly Decreasing and Geometric Curve. With the addition of this parameter the number of tests grows fivefold and to allow it to be feasible with regards to execution time and the analysis to not be overwhelmed by data, the granularity of tested process numbers has been reduced from one at every 8, utilized on Allgather, to one at every 16. Besides this deliberate action to reduce the total size of experiments, due to infrastructure demand and unavailability the execution of this set of tests was held only on Cervino.

Figure 31 provides 5 hybrid heat maps with results of average times, one for each block size distribution, similar to the ones employed for the Allgather discussion. If Sparbit is not the best, a discrete color scale indicates what traditional algorithm was the best. Otherwise, a continuous greyscale color where lighter is better indicates how much Sparbit improved over the second best algorithm. For brevity, all the portrayed plots are for sequential and cyclic mappings united, presenting for each case the smallest overall time achieved and following the same principles as the ones outlined for the Allgather united heat maps in Section 6.2.5.

The first remarkable aspect of the plots is the considerable behaviour difference between the results of the Broadcast distribution and all others. On this case, Sparbit was the best on 78.57% (528/672) of tests, with mean improvements of 56.82% and highest reaching 92.38%. These results outline how on this particular distribution Sparbit is vastly superior to other algorithms, both in the broadness and effectiveness of its action. The reason for Sparbit's large dominance on this configuration is its own architecture, since when there is only one block of data being exchanged the algorithm performs exactly like a regular broadcast binomial tree, which has both locality efficiency and optimal cost for this specific data distribution format. Below the 32 KiB block size mark, Sparbit is the best on all cases, while above it Bruck achieves the best times on most tests.

On all other distributions Sparbit achieves a much smaller percentage of cases as the best, revolving around 15% for Linearly Decreasing and up to 35% on the Geometric Curve, with Half Full and Spike in between. Sparbit's improvement percentage also reduced in comparison to Broadcast, with means of all metrics close to 15% on Half Full and Linearly Decreasing, 20% for Spike and 28% for Geometric Curve. The highest improvements on the Broadcast distribution are more significant than the ones achieved on the Allgather tests, but are smaller



(a) Broadcast.

(b) Spike.

(c) Half full.

(d) Linearly decreasing.

(e) Geometric curve.

Figure 31 – Hybrid heat maps for each distribution with the mappings united displaying a discrete color for best average time algorithm or Sparbit's improvement over the second as a greyscale if it is the best.

on all other distributions. Table 8 summarizes all the points discussed so far, presenting the percentage of cases where Sparbit achieves the best time along with its mean, median and highest percentage improvements over the second best algorithm. Conversely, Figure 32 presents Boxplots describing the behaviour of Sparbit's time reduction over the second best algorithm on each of the block size distributions and with detail for the minimum, average and maximum metrics.

The reason for Sparbit to have a significantly smaller performance on Allgatherv in comparison to Allgather is once again the relation between the algorithms and the mappings.

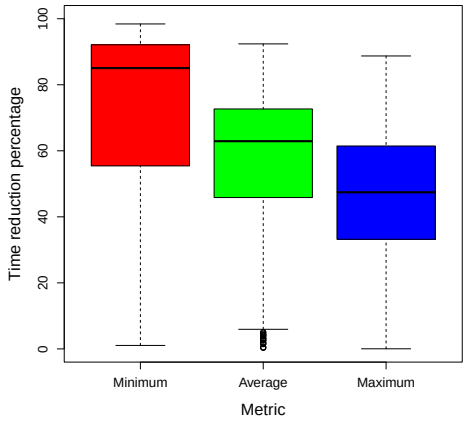
Distribution	Metric	Best in	Mean	Median	Highest
Broadcast	Min	87.20	73.03	85.05	98.42
	Avg	78.57	56.82	62.91	92.38
	Max	79.32	46.48	47.44	88.72
Spike	Min	25.30	20.55	17.74	59.26
	Avg	25.00	21.61	22.87	73.26
	Max	26.79	27.07	21.72	81.80
Half Full	Min	27.53	19.60	18.34	54.58
	Avg	18.90	15.44	12.84	51.71
	Max	09.08	13.32	09.49	43.93
Linearly Decreasing	Min	17.56	14.72	12.28	49.17
	Avg	14.58	17.86	18.18	45.07
	Max	10.57	16.89	13.98	37.84
Geometric Curve	Min	34.97	23.87	22.76	67.63
	Avg	35.57	28.37	28.61	59.83
	Max	32.29	34.14	33.82	71.00

Table 8 – Sparbit improvement of metrics over the second best algorithm for each of the block size distributions.

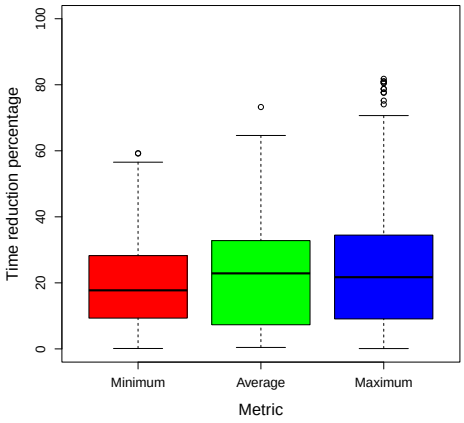
For these experiments 4 machines of the Cervino cluster were employed, which is a power of two amount and thus favourable for Bruck, allowing it to reach its higher locality potential. Therefore, Sparbit's worse presented results are largely influenced by the improved times of Bruck under cyclic mapping for this infrastructure configuration. Evidence of these claims can be found when looking on the improvements for the sequential side alone, specially for Spike and Geometric Curve, where Sparbit is respectively the best on 61.16% and 60.42% of cases, with mean (median) improvements of 57.75% (64.4%) and 27.74% (28.7%). Half Full and Linearly Decreasing are where Sparbit has the lowest performance, both on the sequential side alone and the united mappings, probably due to a surge on the performance of the linear algorithms, that only manifest on these two distributions, as seen on Figures 31c and 31d.

For all distributions except Broadcast, there are little to no cases where Sparbit achieves the best times on data sizes smaller than 1 KiB, while above it the tests where it is the best are more common on odd numbers of processes. The reason for this is that when the number of processes is odd, the maximum number of rings on a step (g_{max}) for Bruck will be 1 and thus its locality potential will be partially compromised on the last steps, as various sends will have to inevitably cross machines.

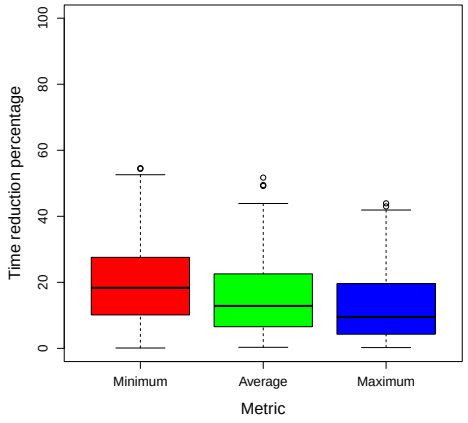
For the Allgatherv we have modified the OSU code to include both the block size distributions and to gather the standard deviation of the 50 iterations for each test. This latter data is presented through the 5 plots on Figure 33, which display cumulative distribution functions of the coefficient of variation for every algorithm with the two mappings united. Each of the plots represents one of the distributions and inside of them differently colored lines identify the curves of each algorithm. The value in the vertical axis indicates the fraction of tests which have a smaller or equal coefficient of variation than the one indicated in the horizontal axis. The



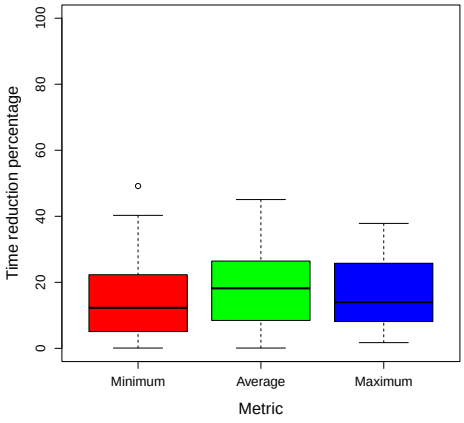
(a) Broadcast.



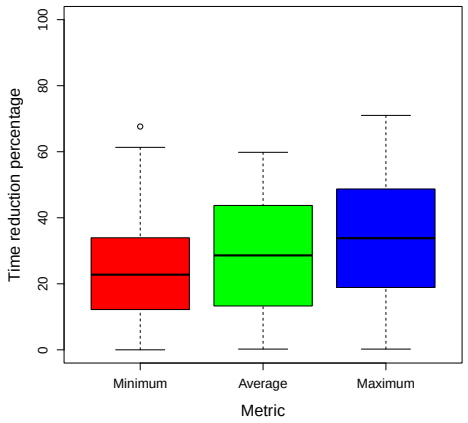
(b) Spike.



(c) Half full.



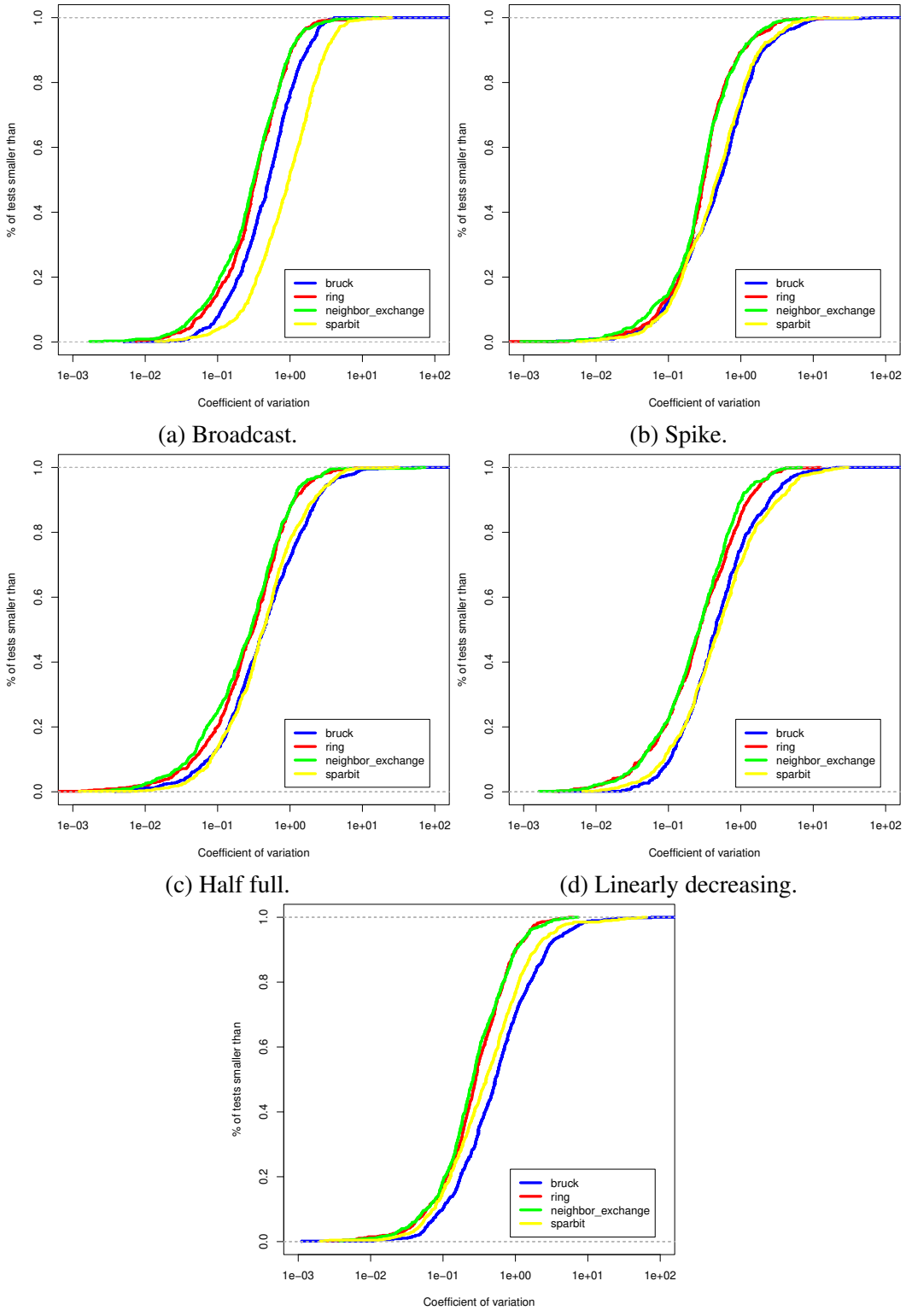
(d) Linearly decreasing.



(e) Geometric curve.

Figure 32 – Boxplots of Sparbit's Allgatherv time reduction over the second best algorithm on multiple distributions.

coefficient of variation is calculated by dividing the standard deviation of a test by its mean. From the plots, all the algorithms can be considered very stable on all distributions, since for



(a) Broadcast.

(b) Spike.

(c) Half full.

(d) Linearly decreasing.

(e) Geometric curve.

Figure 33 – Cumulative distribution function of the coefficient of variation for each algorithm's best times on each distribution.

every combination of the two, more than 90% of tests had a coefficient of variation smaller than 10%. Very high values are also much rarer on the remaining 10%, with the maximum for Broadcast and Linearly Decreasing being around 25% and for Spike, Half Full and Geometric

Curve around 70%. The distribution for the variation of Ring and Neighbor Exchange are very close to each other on all plots and both generally achieve smaller values than Bruck and Sparbit. This is probably due to their very stable communication patterns, which are less prone to diverse network fluctuations and slowdowns. Bruck and Sparbit also have very similar curves in relation to each other, specially on Spike, Half Full and Linearly Decreasing. On Broadcast Bruck has smaller deviations than Sparbit, while the opposite happens on the Geometric Curve. Although all algorithms have variations in relation to the others, they are all very small and only visible on the plots due to the logarithmic x scale. Therefore, from the findings of our experiments, the standard deviation is not significantly different among algorithms and thus not a meaningful criteria for selection or exclusion of any in relation to the others.

6.4 ALGORITHM SELECTION

The heat maps and other data presented throughout Sections 6.2 and 6.3 show that the areas on which each of the algorithms is the best assume various uneven forms that are hard to predict. This is a reflection of both the theoretical costs and practical limitations presented in Chapter 2. Additionally, they vary a great deal depending on the size and configuration of the infrastructure, as well as on the mapping employed to distribute the processes. Albeit the highest improvements of Sparbit are concentrated on large data sizes and process numbers, it still provides consistent benefits on almost every individual tested parameter value. These observations highlight the need for a precise tool for algorithm selection, able to choose the best option for each case as portrayed on the heat maps, otherwise there is a waste of potential performance. Solving this issue is naturally no trivial task and it is far from the scope of this work. Fortunately, there is an active research on this subject, with early works such as (FARAJ; YUAN, 2005; FARAJ; YUAN; LOWENTHAL, 2006; PJEšIVAC-GRBOVIć et al., 2007) along with more recent contributions in (HUNOLD; CARPEN-AMARIE, 2018; HUNOLD et al., 2020; NURIYEV; LASTOVETSKY, 2020). Moreover, there are also open source tuning tools like the Open Tool for Parameter Optimization (OTPO) developed for Open MPI (OpenMPI Team, 2021).

6.5 CONSIDERATIONS

This Chapter has been concentrated on providing information about the design and execution of our experiments and also on analysing the obtained results. Initially, the employed Yahoo and Cervino test infrastructures are described in detail under terms of machine configuration and topology. Next, the usage of Open MPI as the chosen implementation and OSU Micro Benchmarks as the workload are described, coupled with the list of the parameters and runtime routines. The analysis starts with the Allgather routine by presenting a set of heat maps which simultaneously show the best algorithm for each case and Sparbit's improvement when it has the best performance. The analysis progresses first by addressing the cases where our proposal is the

best and why these form such unusual shapes, which is in fact the result of a mixed individual relation between logarithmic and linear classes of algorithms with Sparbit. The latter is better than the first group mostly on large data sizes and better than the second on small ones, thus on the merged final version it is the best on the intersection of both behaviours. On overbooked scenarios Sparbit is the best for the majority of cases.

The discussion then proceeds into analysing the relation between the already presented average times with the minimum and maximum ones. The results indicate that there is a relation between the maximum and the average, since in virtually all cases where the proposal has the smallest maximum times it also has the smallest average times. The minimum is the metric with the most number of total and individual cases – where Sparbit was the best only on this metric – which indicates that it might be easier for the proposed algorithm to reduce the minimum time, specially on faster networks and machines. Next, the communication time reduction provided by our novel approach is also discussed, with the major improvements being on large data sizes and number of processes, reaching a mean of over 60% and 43% on Yahoo and Cervino respectively, both with sequential mapping. These improvements over state of the art collective algorithms on modern MPI implementations reinforce the validity of Sparbit, whose local communication pattern allows the potential of its logarithmic complexity to be fully exploited. Considering all the tests for each scenario, mean improvements range from 10% to 35%. On the last part of the Allgather analysis we discuss the cases as the best and respective improvements for both sequential and cyclic mappings united, outlining the smallest achieved overall times. Sparbit maintains above 95% of the amount of cases where it was the best under sequential mapping on the two infrastructures. The mean improvements are naturally reduced but still remain around 20% for Yahoo and 27% for Cervino. The conclusion for the Allgather routine according to our experiments is that about 40% of all executions could be benefiting from around a 20% improvement on communication time by simply employing Sparbit.

The next and final analysis Section is dedicated to the Allgatherv, which presents an additional dimension regarding the block size distributions. Due to the larger amount of data, the analysis is presented directly by the union of times for sequential and cyclic mappings. On the Broadcast distribution Sparbit is the best on the vast majority of cases, achieving the smallest time on near to 80% of tests. The improvements for this distribution are also higher, with means above 55%. This happens due to its fall-back to a binomial tree broadcast distribution when there is only one block to be delivered. On all other distributions its improvements are much smaller, being the best from around 15% to 35% of cases with improvements close to 20%. Here it presents smaller percentages due to the amount of employed machines being very favourable to Bruck under cyclic, which reduces the latter's time to much better spots and thus diminishes Sparbit's overall improvements. For this collective routine the standard deviation of each iteration was also collected and all algorithms have presented overall very stable executions on all distributions, with the bulk of standard deviations representing no more than 10% of the mean value.

7 CONCLUSION

Throughout this thesis we have highlighted the existing dichotomy of the employed Allgather and Allgatherv algorithms for MPI, in which the best theoretical ones have physical limitations, mainly regarding communication locality, while the ones that present more local exchanges have a poor theoretical complexity. To address this current situation, the Stripe Parallel Binomial Trees (Sparbit) algorithm has been developed and presented, utilizing pipelined sends and binomial trees to deliver the data faster and with greater locality. It is also able to perform the operation semantics with optimal time costs of both bandwidth and latency without any usage restrictions. Sparbit was empirically compared against the available algorithms on Open MPI 4.0.3, namely Bruck, Recursive Doubling, Ring and Neighbor Exchange for Allgather and Bruck, Ring and Neighbor Exchange for Allgatherv. The experiments were executed on two HPC infrastructures with different topologies, network speeds and machine configurations to enrich the data sources for the analysis. A broad discussion was devised to address the relation of the algorithms and the mapping of processes to the infrastructure. It provided insights on the favourable and unfavourable possible mappings for Bruck and Sparbit, allowing theoretically backed predictions of the performance and solid understanding of the experiment's outcome.

For the Allgather collective, results of the tests indicate that Sparbit achieves the smallest minimum, average and maximum times on a large fraction of test cases in the two infrastructures. The sequential mapping is the one where it ameliorates more cases and does so more intensively, however under cyclic mapping a considerable amount of cases have also been improved. By merging the sequential and cyclic times of all algorithms we were able to find what option achieved the smallest overall time on each case. For this analysis, Sparbit achieved a large percentage of cases as the best, maintaining almost the total amount of smallest times it had on the sequential mapping alone. This indicates that it is not only able to consistently improve the execution times under sequential mapping, where other logarithmic algorithms struggle, but that its time also surpass their best executions on the cyclic mapping as well, thus improving over the state of the art up to near half of the tests.

For the Allgatherv tests, Sparbit's improvements on the Broadcast distribution were very broad and effective, achieving high percentages of cases as the best and with high improvements. On the remaining Spike, Half Full, Linearly Decreasing and Geometric Curve distributions the improvements were fewer and smaller. This latter drop is not a result of a deficiency from Sparbit but actually from a boost on other algorithms' performance, specially Bruck, due to the infrastructure configuration and relation with the mappings. Indeed, the whole discussion and results of the experiments outline how the relation of mapping, algorithm and infrastructure plays a significant role in defining the communication performance of each algorithm, both when executing Allgather and Allgatherv. Therefore, with a different infrastructure configuration and number of machines results could have drastically changed for all algorithms on the Allgatherv tests. For this round of experiments the standard deviation of executions was also collected, but

all algorithms had very stable runs with low variations, similar among each other.

7.1 LIMITATIONS

The first limitation of the proposal is its inability to handle dynamism on the network. In this sense, once a process distribution is assigned to the machines before the execution of the algorithm, its operation will be deterministic based on this organization. Therefore, the communication pattern employed will be the same with disregard for the current utilization, latency or general condition of the involved links. When the network state is unstable or the overall capacity of its performance aspects is variable, Sparbit and the other traditional algorithms will suffer from degradations in comparison to more sophisticate proposals aware to these phenomena.

The second point of limitation is the foreseen performance degradation of Sparbit as the number of communicating participants rises to extreme values. As discussed in Section 4.3, this might happen due to the large amount of pipelined sends employed by the communication, which grows linearly with the number of processes and can overwhelm either the network, by using to much bandwidth in a short period of time, or the nodes itself, due to potentially high processing resources required for issuing all connections. This latter factor might also be at play for smaller number of processes, which could partially hinder Sparbit's capacity to be employed on non-blocking communications, as our theoretical analysis suggests that it would require more CPU cycles to perform the exchanges, thus reducing the total potential for communication-computation overlap. Both these discussed limitations regarding scaling degradations stem solely from theoretical predictions of the algorithm's behaviour and, in the same fashion as positive predictions, can only be truly verified via the proper experiments.

Finally, the formal definition of Sparbit and the experiments executed throughout this work do not allow us to guarantee that the proposal is agnostic to the type of network technology employed and neither by how much its performance is portable to other types of interconnect. We do know however from the findings presented, that among the major factors that influence Sparbit's and other algorithms' effectiveness is the topology size and organization, as well as the relation between the mappings and the amount of machines employed. Once again, novel experiments are necessary to verify if such factors have the same roles on other types of networks and whether Sparbit could sustain its improvements under these diverse environments.

7.2 CONTRIBUTIONS

The main scientific contribution of this work is the proposal of a novel basic MPI Allgather algorithm entitled Stripe Parallel Binomial Trees (Sparbit). It employs the binomial tree one to many distribution routine with multiple instances executing in parallel to transfer the block of each process to all others. This communication architecture is the core of the algorithm's functioning and allows it to perform the complete data exchange in a logarithmic number of

steps, with no usage restrictions for particular process numbers and with improved locality. As the experimental results show that the proposed algorithm can deliver improvements in many execution scenarios, it is a valid contribution for aggregation on diverse MPI implementations, specially open source versions such as Open MPI and MPICH.

7.3 FUTURE WORK

The implementation employed here and geared towards the MPI Allgather and Allgatherv collectives served as a proof of concept for the potential benefits of the algorithm. As future work we aim at extending the implementation towards other more specific versions of the Allgather and Allgatherv calls. The first are the non-blocking variants, which allow the communication to take place in the background while the CPU can still be used for computation. As processor contention due to communication routines have high performance degradation effects, the usage of such class of calls is ever increasing on novel applications and thus a good target for Sparbit's improvement. The second group is the neighbor calls, which allow the communication to take place among only a subset of processes in the communicator. These variants were created to allow specific optimizations on lattice and neighborhood communication routines, common in many scientific applications. On this case however, there is still need for analysis regarding if and how Sparbit could present significant performance benefits as the communication patterns are different.

An additional future work direction is executing novel experiments on supercomputing environments, which have larger hierarchical topologies and other types of networks. Such infrastructures would allow two distinct analysis to take place: the first focused on how Sparbit would behave when superscaling the execution, understanding which is vital for its utilization on massive computing environments, akin of the exascale age; the second analysis is directed towards Sparbit's behaviour and performance on infrastructures with other types of networks, mainly InfiniBand which is prominent on many supercomputers and computing clusters and which we were unable to assess on the experimental scenarios employed throughout this work. This analysis is of paramount importance since it allows the quantification of the proposal's performance portability and how affected it is by the particular networking technology employed. Another possible scenario over which experiments could be executed are computational grids, as their geographically distributed nature could greatly benefit from communication algorithms with improved locality, such as Sparbit. On either environment, we plan to analyse a third important aspect that is the CPU usage and impact of different algorithms, specially towards understanding why Sparbit is less affected on CPU-constrained scenarios, phenomenon to which, given our current data, has no definitive explanation.

Tests are planned for the Allgather, Allgatherv and other variants which will receive an Sparbit implementation in the future. These experiments are to be held in the infrastructure of the Santos Dumont supercomputer from the National Laboratory of Scientific Computing, in which

a project for execution was approved. Finally, there is also the aim of integrating the already developed codes for distribution into the Open MPI and MPICH implementations, two widely employed options on production systems. The future work can be summarized as:

- Expand the implementation for the non-blocking Allgather and Allgatherv variants.
- Study the feasibility and possible benefits of extending the proposal to the neighbor Allgather and Allgatherv variants.
- Execute novel experiments on a supercomputer and possibly computational grid environments.
- Analyse and quantify the influence of the network technology on the performance of Sparbit.
- Collect data on the CPU utilization of each algorithm and its effect on their performance.
- Formally integrate the proposal into Open MPI and possibly MPICH.

BIBLIOGRAPHY

AL-FARES, M.; LOUKISSAS, A.; VAHDAT, A. A scalable, commodity data center network architecture. **ACM SIGCOMM computer communication review**, ACM New York, NY, USA, v. 38, n. 4, p. 63–74, 2008. Cited 2 times on pages 8 and 31.

ALVAREZ-LLORENTE, J. et al. Formal Modeling and Performance Evaluation of a Run-Time Rank Remapping Technique in Broadcast, Allgather and Allreduce MPI Collective Operations. In: **17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)**. [S.l.: s.n.], 2017. p. 963–972. Cited 6 times on pages 14, 15, 26, 31, 39, and 40.

ARAP, O. et al. Adaptive Recursive Doubling Algorithm for Collective Communication. In: **2015 IEEE International Parallel and Distributed Processing Symposium Workshop**. [S.l.: s.n.], 2015. p. 121–128. Cited 2 times on pages 36 and 37.

BENSON, G. D. et al. A Comparison of MPICH Allgather Algorithms on Switched Networks. In: DONGARRA, Jack; LAFORENZA, Domenico; ORLANDO, Salvatore (Ed.). **Recent Advances in Parallel Virtual Machine and Message Passing Interface**. [S.l.]: Springer, 2003. (Lecture Notes in Computer Science), p. 335–343. ISBN 978-3-540-39924-7. Cited 4 times on pages 14, 27, 49, and 50.

BERNHOLDT, D. E. et al. A survey of MPI usage in the US exascale computing project. **Concurrency and Computation: Practice and Experience**, v. 32, n. 3, p. e4851, 2020. ISSN 1532-0634. Available at:<https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4851>. Cited on page 14.

BIEDRON, R. T. et al. **FUN3D Manual: 13.6**. [S.l.]: National Aeronautics and Space Administration, Langley Research Center, 2019. Cited on page 19.

BOSILCA, G. et al. Online Dynamic Monitoring of MPI Communications. In: RIVERA, Francisco F.; PENA, Tomas F.; CABALEIRO, Jose C. (Ed.). **Euro-Par 2017: Parallel Processing**. Cham: Springer International Publishing, 2017. (Lecture Notes in Computer Science), p. 49–62. ISBN 978-3-319-64203-1. Cited on page 14.

BRUCK, J. et al. Efficient algorithms for all-to-all communications in multiport message-passing systems. **IEEE Transactions on Parallel and Distributed Systems**, v. 8, n. 11, p. 1143–1156, nov. 1997. ISSN 1558-2183. Conference Name: IEEE Transactions on Parallel and Distributed Systems. Cited 5 times on pages 14, 15, 29, 35, and 50.

CHAKRABORTY, S. et al. Cooperative Rendezvous Protocols for Improved Performance and Overlap. In: **SC18: International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.: s.n.], 2018. p. 361–373. Cited on page 14.

CHEN, J. et al. Performance evaluation of Allgather algorithms on terascale Linux cluster with fast Ethernet. In: **Eighth International Conference on High-Performance Computing in Asia-Pacific Region (HPCASIA'05)**. [S.l.: s.n.], 2005. p. 6 pp.–442. Cited 3 times on pages 15, 27, and 35.

CHUNDURI, S. et al. Characterization of MPI Usage on a Production Supercomputer. In: **SC18: International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.: s.n.], 2018. p. 386–400. Cited 2 times on pages 14 and 24.

DONGARRA, J.; LUSZCZEK, P. TOP500. In: PADUA, David (Ed.). **Encyclopedia of Parallel Computing**. Boston, MA: Springer US, 2011. p. 2055–2057. ISBN 978-0-387-09766-4. Available at:<https://doi.org/10.1007/978-0-387-09766-4_157. Cited on page 22.

FAGG, G. E.; DONGARRA, J. J. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In: DONGARRA, Jack; KACSUK, Peter; PODHORSZKI, Norbert (Ed.). **Recent Advances in Parallel Virtual Machine and Message Passing Interface**. Berlin, Heidelberg: Springer, 2000. (Lecture Notes in Computer Science), p. 346–353. ISBN 978-3-540-45255-3. Cited on page 22.

FARAJ, A.; YUAN, X. Automatic generation and tuning of MPI collective communication routines. In: **19th Annual Int. Conf. on Supercomputing**. New York, NY, USA: Association for Computing Machinery, 2005. (ICS '05), p. 393–402. ISBN 978-1-59593-167-2. Cited on page 88.

FARAJ, A.; YUAN, X.; LOWENTHAL, D. STAR-MPI: self tuned adaptive routines for MPI collective operations. In: **20th Annual Int. Conf. on Supercomputing**. New York, NY, USA: Association for Computing Machinery, 2006. (ICS '06), p. 199–208. ISBN 978-1-59593-282-2. Cited on page 88.

FRIGO, M.; JOHNSON, S. G. Fftw user's manual. **Massachusetts Institute of Technology**, Citeseer, 1999. Cited on page 19.

GABRIEL, E. et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: KRANZLMüLLER, Dieter; KACSUK, Péter; DONGARRA, Jack (Ed.). **Recent Advances in Parallel Virtual Machine and Message Passing Interface**. Berlin, Heidelberg: Springer, 2004. (Lecture Notes in Computer Science), p. 97–104. ISBN 978-3-540-30218-6. Cited 3 times on pages 22, 23, and 24.

GONG, Y.; HE, B.; ZHONG, J. Network Performance Aware MPI Collective Communication Operations in the Cloud. **IEEE Transactions on Parallel and Distributed Systems**, v. 26, n. 11, p. 3079–3089, nov. 2015. ISSN 1558-2183. Cited 2 times on pages 14 and 18.

HOCKNEY, R. W. The communication challenge for MPP: Intel Paragon and Meiko CS-2. **Parallel Computing**, v. 20, n. 3, p. 389–398, mar. 1994. ISSN 0167-8191. Available at:<http://www.sciencedirect.com/science/article/pii/S0167819106800219>. Cited on page 25.

HUNOLD, S. et al. Predicting MPI Collective Communication Performance Using Machine Learning. In: **2020 IEEE Int. Conf. on Cluster Computing (CLUSTER)**. [S.l.: s.n.], 2020. p. 259–269. ISSN: 2168-9253. Cited on page 88.

HUNOLD, S.; CARPEN-AMARIE, A. Algorithm Selection of MPI Collectives Using Machine Learning Techniques. In: **2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)**. [S.l.: s.n.], 2018. p. 45–50. Cited on page 88.

IBM Corporation. Ibm spectrum mpi: Accelerating high-performance application parallelization. 2016. Available at:<https://www.ibm.com/downloads/cas/BKQNLJWO>. Cited on page 22.

INOZEMTSEV, G.; AFSAHI, A. Designing an Offloaded Nonblocking MPI_allgather Collective Using CORE-Direct. In: **2012 IEEE International Conference on Cluster Computing**. [S.l.: s.n.], 2012. p. 477–485. ISSN: 2168-9253. Cited 2 times on pages 36 and 37.

Intel. **Intel MPI Library**. 2021. Available at:<https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/mpi-library.html>. Cited on page 22.

JEANNOT, E.; MERCIER, G. Near-Optimal Placement of MPI Processes on Hierarchical NUMA Architectures. In: D'AMBRA, Pasqua; GUARRACINO, Mario; TALIA, Domenico (Ed.). **Euro-Par 2010 - Parallel Processing**. Berlin, Heidelberg: Springer, 2010. (Lecture Notes in Computer Science), p. 199–210. ISBN 978-3-642-15291-7. Cited 2 times on pages 39 and 40.

JEANNOT, E.; SARTORI, R. Improving MPI Application Communication Time with an Introspection Monitoring Library. In: **IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**. [S.l.: s.n.], 2020. p. 691–700. Cited 4 times on pages 14, 18, 39, and 40.

KANDALLA, K. et al. Designing multi-leader-based Allgather algorithms for multi-core clusters. In: **2009 IEEE International Symposium on Parallel Distributed Processing**. [S.l.: s.n.], 2009. p. 1–8. ISSN: 1530-2075. Cited on page 38.

KIM, J. et al. Technology-Driven, Highly-Scalable Dragonfly Topology. In: **International Symposium on Computer Architecture**. [S.l.: s.n.], 2008. p. 77–88. ISSN: 1063-6897. Cited on page 31.

KOTHE, D.; LEE, S.; QUALTERS, I. Exascale Computing in the United States. **Computing in Science Engineering**, v. 21, n. 1, p. 17–29, jan. 2019. ISSN 1558-366X. Cited on page 14.

KUMAR, S.; SHARKAWI, S. S.; JAN, K. A. N. Optimization and Analysis of MPI Collective Communication on Fat-Tree Networks. In: **2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2016. p. 1031–1040. ISSN: 1530-2075. Cited on page 14.

KURNOSOV, M. G. Dynamic mapping of all-to-all collective operations into hierarchical computer clusters. In: **13th International Scientific-Technical Conference on Actual Problems of Electronics Instrument Engineering (APEIE)**. [S.l.: s.n.], 2016. v. 02, p. 475–478. Cited 4 times on pages 14, 31, 39, and 40.

LAGUNA, I. et al. A large-scale study of MPI usage in open-source HPC applications. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.]: ACM, 2019. p. 1–14. ISBN 978-1-4503-6229-0. Available at:<https://dl.acm.org/doi/10.1145/3295500.3356176>. Cited 2 times on pages 14 and 24.

LI, D.; WANG, Y.; ZHU, W. Topology-Aware Process Mapping on Clusters Featuring NUMA and Hierarchical Network. In: **2013 IEEE 12th International Symposium on Parallel and Distributed Computing**. [S.l.: s.n.], 2013. p. 74–81. ISSN: 2379-5352. Cited 2 times on pages 39 and 40.

LI, M. et al. High Performance MPI Datatype Support with User-Mode Memory Registration: Challenges, Designs, and Benefits. In: **2015 IEEE International Conference on Cluster Computing**. [S.l.: s.n.], 2015. p. 226–235. ISSN: 2168-9253. Cited on page 50.

MA, T. et al. HierKNEM: An Adaptive Framework for Kernel-Assisted and Topology-Aware Collective Communications on Many-core Clusters. In: **2012 IEEE 26th International Parallel and Distributed Processing Symposium**. [S.l.: s.n.], 2012. p. 970–982. ISSN: 1530-2075. Cited on page 38.

MA, T. et al. Process Distance-Aware Adaptive MPI Collective Communications. In: **2011 IEEE International Conference on Cluster Computing**. [S.l.: s.n.], 2011. p. 196–204. ISSN: 2168-9253. Cited 2 times on pages 36 and 37.

MESSINA, P. The exascale computing project. **Computing in Science & Engineering**, IEEE, v. 19, n. 3, p. 63–67, 2017. Cited on page 14.

MIRSADEGHI, S. H.; AFSAHI, A. Topology-Aware Rank Reordering for MPI Collectives. In: **IEEE International Parallel and Distributed Processing Symposium Workshops**. [S.l.: s.n.], 2016. p. 1759–1768. Cited 3 times on pages 28, 39, and 40.

MPI Forum. MPI: A Message-Passing Interface Standard. p. 868, 2015. Cited 4 times on pages 14, 18, 22, and 24.

MPICH Team. **MPICH: Version 3.3.2**. 2019. Available at:<https://www.mpich.org/static/downloads/3.3.2/mpich-3.3.2.tar.gz>. Cited 4 times on pages 15, 26, 28, and 29.

MPICH Team. **MPICH - High Performance Portable MPI**. 2020. Available at:<https://www.mpich.org/. Cited on page 22.

Network-Based Computing Laboratory. **MVAPICH benchmarks**. 2021. Cited on page 65.

NURIYEV, E.; LASTOVETSKY, A. Accurate runtime selection of optimal MPI collective algorithms using analytical performance modelling. **arXiv:2004.11062 [cs]**, abr. 2020. ArXiv: 2004.11062. Cited on page 88.

Open MPI Team. **OpenMPI: Version 4.0.3**. 2020. Available at:<https://download.open-mpi.org/release/open-mpi/v4.0/openmpi-4.0.3.tar.gz>. Cited 6 times on pages 15, 24, 26, 28, 29, and 43.

Open MPI Team. **Open MPI: General run-time tuning**. 2021. Available at:<https://www.open-mpi.org/faq/?category=tuning>. Cited on page 23.

OpenMPI Team. **Open Tool for Parameter Optimization (OTPO)**. 2021. Cited on page 88.

PARSONS, B. S.; PAI, V. S. Accelerating MPI Collective Communications through Hierarchical Algorithms Without Sacrificing Inter-Node Communication Flexibility. In: **2014 IEEE 28th International Parallel and Distributed Processing Symposium**. [S.l.: s.n.], 2014. p. 208–218. ISSN: 1530-2075. Cited on page 38.

PJEšIVAC-GRBOVIć, J. et al. MPI collective algorithm selection and quadtree encoding. **Parallel Computing**, v. 33, n. 9, p. 613–623, set. 2007. ISSN 0167-8191. Cited on page 88.

THAKUR, R.; RABENSEIFNER, R.; GROPP, W. Optimization of Collective Communication Operations in MPICH. **The International Journal of High Performance Computing Applications**, v. 19, n. 1, p. 49–66, fev. 2005. ISSN 1094-3420. Available at:<https://doi.org/10.1177/1094342005051521>. Cited 6 times on pages 15, 31, 32, 35, 37, and 50.

TOP500. **TOP500 LIST - NOVEMBER 2020**. 2020. Available at:<https://top500.org/lists/top500/list/2020/11/>. Cited on page 22.

TRäFF, J. L.; HUNOLD, S. Decomposing MPI Collectives for Exploiting Multi-lane Communication. In: **2020 IEEE International Conference on Cluster Computing (CLUSTER)**. [S.l.: s.n.], 2020. p. 270–280. ISSN: 2168-9253. Cited 2 times on pages 36 and 37.

TRäFF, J. L. et al. A Simple, Pipelined Algorithm for Large, Irregular All-gather Problems. In: LASTOVETSKY, Alexey; KECHADI, Tahar; DONGARRA, Jack (Ed.). **Recent Advances in Parallel Virtual Machine and Message Passing Interface**. Berlin, Heidelberg: Springer, 2008. (Lecture Notes in Computer Science), p. 84–93. ISBN 978-3-540-87475-1. Cited 3 times on pages 35, 37, and 69.

WANG, J. et al. A locality-aware shuffle optimization on fat-tree data centers. **Future Generation Computer Systems**, v. 89, p. 31–43, 2018. ISSN 0167-739X. Available at:<http://www.sciencedirect.com/science/article/pii/S0167739X17327310>. Cited 3 times on pages 8, 31, and 32.

YU, H. RMPI: Interface to MPI. 2021. Available at:<https://cran.r-project.org/web/packages/Rmpi/index.html>. Cited on page 19.

ZHOU, H. et al. A Bandwidth-Saving Optimization for MPI Broadcast Collective Operation. In: **2015 44th International Conference on Parallel Processing Workshops**. [S.l.: s.n.], 2015. p. 111–118. ISSN: 1530-2016. Cited on page 35.