



Vrije Universiteit Brussel

FACULTY OF ENGINEERING  
Department of Electronics and Informatics

# Distributed memory reduction operations in presence of process desynchronization

Thesis submitted in fulfilment of the requirements for the award of the degree of  
Doctor in de ingenieurwetenschappen (Doctor in Engineering) by

## Petar Marendić

March 2016

Advisor(s): Prof. Dr. Jan Lemeire  
Prof. Dr. Peter Schelkens





# Examining Committee

---

Prof. Dr. ir. Peter Schelkens – Vrije Universiteit Brussel – Promoter

Prof. Dr. ir. Jan Lemeire – Vrije Universiteit Brussel – Promoter

Prof. Dr. ir. Gerd Vandersteen – Vrije Universiteit Brussel – Committee chair

Prof. Dr. ir. Johan Deconinck – Vrije Universiteit Brussel – Committee vice-chair

Prof. Dr. ir. Nikolaos Deligiannis – Vrije Universiteit Brussel – Committee  
secretary

Prof. Dr. ir. Martin Timmerman – Vrije Universiteit Brussel – Member

Prof. Dr. ir. Koen De Bosschere – Universiteit Gent – Member

Dr. Sabela Ramos Garea, ETH Zürich – Member



*“Origin of the logical.— How did logic come into existence in man’s head? Certainly out of illogic, whose realm originally must have been immense. Innumerable beings who made inferences in a way different from ours perished; for all that, their ways might have been truer. Those, for example, who did not know how to find often enough what is “equal” as regards both nourishment and hostile animals—those, in other words, who subsumed things too slowly and cautiously—were favored with a lesser probability of survival than those who guessed immediately upon encountering similar instances that they must be equal. The dominant tendency, however, to treat as equal what is merely similar—an illogical tendency, for nothing is really equal—is what first created any basis for logic. In order that the concept of substance could originate—which is indispensable for logic although in the strictest sense nothing real corresponds to it—it was likewise necessary that for a long time one did not see or perceive the changes in things. The beings that did not see so precisely had an advantage over those who saw everything “in flux.” At bottom, every high degree of caution in making inferences and every sceptical tendency constitute a great danger for life. No living beings would have survived if the opposite tendency—to affirm rather than suspend judgement, to err and make up things rather than wait, to assent rather than negate, to pass judgement rather than be just—had not been bred to the point where it became extraordinarily strong.”*

— Friedrich W. Nietzsche



# Table of contents

---

<b>Acknowledgments</b>	<b>v</b>
<b>Synopsis</b>	<b>vii</b>
<b>Acronyms</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Challenges of Exascale Computing . . . . .	3
1.1.1 Exascale applications . . . . .	6
1.1.2 Space Weather and in-situ visualization . . . . .	8
1.1.3 Global reduction operations and process desynchronization . . . . .	11
1.1.4 State of the art & challenges . . . . .	12
1.2 Outline and summary of scientific contributions . . . . .	15
<b>2 Distributed memory computing and MPI</b>	<b>19</b>
2.1 Parallel programming models . . . . .	19
2.1.1 Distributed memory systems . . . . .	20
2.1.2 Message Passing . . . . .	24
2.1.3 Shared memory programming . . . . .	25
2.1.4 Current trends in programming models for HPC . . . . .	26
2.2 Message Passing Interface . . . . .	27
2.2.1 MPI implementations . . . . .	29
2.2.2 Point-to-point operations . . . . .	30
2.2.3 Collective operations . . . . .	34
2.2.4 Global Reduction Operations . . . . .	35
2.2.5 Optimization of collective operations . . . . .	36
2.3 Non-blocking collective operations . . . . .	37
2.4 Visualization as a global reduction . . . . .	38
2.5 The reduction problem . . . . .	39
2.5.1 Design constraints on reduction algorithms . . . . .	41

2.6	State-of-the-art reduction algorithms . . . . .	43
2.6.1	Binomial Tree . . . . .	43
2.6.2	Parallel Ring . . . . .	44
2.6.3	Butterfly . . . . .	44
2.6.4	Radix-k . . . . .	47
2.6.5	Linear Pipeline . . . . .	48
2.6.6	Process desynchronization after collective operation invocation	49
2.6.7	Related work on reduction algorithms . . . . .	50
2.7	Summary . . . . .	52
<b>3</b>	<b>Process arrival time, imbalance and collective operation runtime</b>	<b>55</b>
3.1	Load imbalance . . . . .	56
3.1.1	Load balancing strategies . . . . .	57
3.2	System noise . . . . .	58
3.3	Process Arrival Time and collective operation runtime . . . . .	60
3.3.1	Collective operation runtime . . . . .	61
3.3.2	Implications on relative algorithm performance . . . . .	63
3.3.3	Prior studies on imbalanced process arrival times . . . . .	64
3.3.4	Absorption time and synchronization delay . . . . .	65
3.4	Time complexity models . . . . .	66
3.4.1	Linear model . . . . .	70
3.4.2	Independent progress . . . . .	71
3.5	Algorithm Complexity . . . . .	72
3.5.1	Lower bounds . . . . .	73
3.6	Imbalance robust collective operations . . . . .	74
3.7	An argument for clairvoyancy: PAT cognizant reduction algorithms .	76
3.8	Summary . . . . .	79
<b>4</b>	<b>Benchmarking Parallel Computing Systems</b>	<b>81</b>
4.1	Hardware Platforms . . . . .	82
4.1.1	Lynx . . . . .	82
4.1.2	VSC muk . . . . .	82
4.1.3	PRACE CURIE . . . . .	82
4.2	Measurements of elapsed time . . . . .	83
4.2.1	Hardware clocks . . . . .	83
4.2.2	Synchronization . . . . .	84
4.2.3	MPI timer . . . . .	85
4.3	Benchmarking of Message Passing primitives . . . . .	85
4.3.1	Point-to-Point operations . . . . .	85
4.3.2	All pairs communication microbenchmark . . . . .	87
4.3.3	Collective Operations . . . . .	92

---

4.4	Experimental Method . . . . .	94
4.4.1	Microbenchmark rationale . . . . .	94
4.4.2	Experimental setup . . . . .	97
4.5	Statistical analysis . . . . .	97
4.5.1	Well behaved data . . . . .	97
4.5.2	Comparison of statistical analyses among microbenchmarks . . . . .	100
4.5.3	Bootstrapping and bootstrap plots . . . . .	101
4.5.4	Permutation tests . . . . .	102
4.6	Selecting the most appropriate measure of location . . . . .	105
4.7	Summary . . . . .	105
<b>5</b>	<b>Optimal reduction algorithm for atomic messages</b>	<b>107</b>
5.1	Optimal reduction algorithm for atomic messages . . . . .	108
5.1.1	Absorption potential . . . . .	111
5.2	Atomic message reduction algorithms . . . . .	112
5.2.1	Binomial Tree . . . . .	112
5.2.2	Clairvoyant Reduction . . . . .	113
5.2.3	Local Redirect . . . . .	114
5.2.4	Correctness of Local Redirect . . . . .	117
5.2.5	Time complexity of Local Redirect . . . . .	120
5.3	Experiment Setup . . . . .	121
5.4	Results and discussion . . . . .	123
5.4.1	Simulated performance using the Helsim trace file . . . . .	127
5.4.2	The problem of different communication library versions . . . . .	129
5.5	Summary . . . . .	130
<b>6</b>	<b>Clairvoyant reduction algorithm for non-atomic messages</b>	<b>133</b>
6.1	Absorption potential of a clairvoyant reduction algorithm . . . . .	134
6.2	Clairvoyant (non-atomic) schedule generation . . . . .	136
6.2.1	Implementation details . . . . .	139
6.2.2	Time and space complexity . . . . .	142
6.3	Time complexity of selected algorithms . . . . .	144
6.4	Analysis of experimental data . . . . .	144
6.5	Experiment design . . . . .	148
6.6	Results and discussion . . . . .	151
6.6.1	Catastrophic slowdown . . . . .	154
6.7	Summary . . . . .	155
<b>7</b>	<b>Comparative analysis with non-blocking reduction operations</b>	<b>159</b>
7.1	LibNBC . . . . .	160
7.2	Non-blocking collective operation microbenchmark . . . . .	160

7.3	Experimental results . . . . .	162
7.4	Summary . . . . .	168
<b>8</b>	<b>Epilogue</b>	<b>169</b>
8.1	Conclusions . . . . .	169
8.2	Future work . . . . .	171
<b>Appendix A</b>	<b>Supplementary figures</b>	<b>173</b>
A.1	Helsim trace file autocorrelograms . . . . .	174
A.2	Peer-to-peer transfer time experiment . . . . .	175
A.3	Distribution of p2p message transfer time . . . . .	177
A.4	Variance in time to complete one round of reduction . . . . .	179
A.5	Bootstrap plots . . . . .	180
A.6	Histograms of algorithm runtime for atomic input data . . . . .	182
A.7	Absorption time for the PAT pattern with a single delayed process . . . . .	183
A.8	Observed runtime on the VSC muk cluster . . . . .	184
A.9	Truncated gamma distributed PAT experiment . . . . .	185
	<b>List of publications</b>	<b>187</b>
	<b>References</b>	<b>189</b>
	<b>Index</b>	<b>205</b>

# Acknowledgments

---

There are many people I would like to thank. I would like to begin by offering my sincere thanks to my supervisor, Prof. Jan Lemeire for his support throughout this long and arduous PhD journey. He acted not only as a guiding and benevolent mentor, but also as a friend. He was there to express sincere happiness in my moments of jubilation and to offer his considerate support in my periods of despondency. In no less manner, I want to thank my supervisor, Prof. Peter Schelkens for his wise guidance and support. He was most instructive in helping me successfully traverse the subtleties of scientific publication. Finally, I owe much to his belief in my ability to successfully complete the Ph.D.

Second, I would like to thank all my colleagues at the Intel Exascience Lab in Leuven, chief of which Tom Haber and Roel Wuyts, whose insights and suggestions were a huge contribution to this work. Their passion for science and pursuit of excellence were and still are a continuous source of inspiration. I still look back to those days as perhaps the most enjoyable period of my Ph.D. study.

I also wish to thank my committee members for their helpful comments. In particular, I would like to thank the external committee member Sabela Ramos Garea, for her assistance and advice in completing the final revision of the manuscript. Without her input, this work would not be nearly as comprehensive. I would also like to credit my committee secretary Nikolaos Deligiannis for his selfless help and advice, both before and after the private defense.

This research was conducted at the department of ETRO at the Vrije Universiteit Brussel and the Intel Exascience Lab at IMEC, Leuven. The chief experimental results presented in this work were collected on the Partnership for Advanced Computing in Europe (PRACE) CURIE supercomputer and the Vlaams Supercomputer Centrum (VSC) muk cluster. I gratefully acknowledge the funding received towards my Ph.D. from the following grants: Promotion of Innovation through Science and Technology in Flanders (IWT) IWT speerpuntproject (Exascience) grant IWT503 (ETROWER151), the iMinds Institute and the European Research Council under the European unions Seventh Framework Programme (FP7/2007-2013)/ERC Grant Agreement n.617779 (INTERFERE).

## Acknowledgments

---

I would like to thank my office mates Robrecht Dewaele and Jan Cornellis for many stimulating discussions about languages both spoken and only typed. With you, my days in the office were never dull.

I owe a great deal of thanks to my dear friends Andrei Sechelea, Bruno Cornelis, Gabor Fodor and Frederik Verbist for all those joyous board gaming afternoons and evenings. Your energy and presence has profoundly influenced and shaped my person. I would like to offer special thanks to Gabor Fodor for helping me understand how the scientific method can be applied to fields beyond computational science.

It goes without saying that this Ph.D. would not have been possible without the undying support of my mother and brother Jagoda and Marko Marendić. They showed a great deal of understanding for my decision to move abroad and never flinched in their belief that I have what it takes to complete this undertaking.

The person to whom I am the most indebted is my girlfriend Ting Tang. This Ph.D. would not have been the same without your unconditional love and support.

Petar Marendić,  
March 2016, Brussels, Belgium

# Synopsis

---

Despite decades of exponential growth in computational power, humans continue to find new problems that eclipse available computational resources. This unrelenting pursuit for computational power has brought about supercomputers consisting of millions of individual computing units. Writing programs that would efficiently utilize the computational power of such complex machines has turned out to be a major challenge. As of today, most High Performance Computing (HPC) applications continue to be based on the distributed memory programming paradigm, through the use of Message Passing Interface (MPI).

One of the principal drivers behind the research in this dissertation was the coupling of multi-scale and multi-physics iPIC3D space weather simulation with in-situ raytraced visualization for real-time simulation steering. This application was developed by the Leuven Intel ExaScale Lab as a research prototype for the type of HPC applications projected to run on exascale machines of the 2018-2020 timeframe. Due to the nature of the coupling, load imbalance and process desynchronization arose as one of the leading causes of performance shortcomings. In particular, image compositing and other collective reduction operations were identified as most critical to visualization performance.

This dissertation proposes three new reduction algorithms that are resilient to process desynchronization. Two of the algorithms achieve this using side information to preconstruct reduction schedules, while the third algorithm constructs dynamic reduction schedules at runtime. These algorithms jointly address both the case where the input data is atomic and the case where it is can be arbitrarily segmented. Compared to non-blocking collective operation, imbalance robust collectives have the advantage that they require no changes to legacy code, and are fully transparent to the user.

Finally, a new benchmarking suite capable of injecting customized process desynchronization was developed. The extensive experimental assessments presented in this dissertation indicate that the proposed algorithms conclusively outperform the state of the art.



# Acronyms

---

<b>HPC</b>	High Performance Computing
<b>PET</b>	Process Exit Time
<b>SIMD</b>	Single Instruction Multiple Data
<b>SSE</b>	Streaming SIMD Extensions
<b>ANOVA</b>	Analysis of Variance
<b>NIST</b>	National Institute of Standards and Technology
<b>MPE</b>	MPI Parallel Environment
<b>CLT</b>	Central Limit Theorem
<b>GNU</b>	GNU's Not Unix
<b>MIQP</b>	Mixed Integer Quadratic Programming
<b>RTT</b>	Round Trip Time
<b>RDMA</b>	Remote Direct Memory Access
<b>SKaMPI</b>	Special Karlsruher MPI Benchmark
<b>PAPI</b>	Performance Application Programming Interface
<b>COTS</b>	Commercial off-the-shelf
<b>MPICH</b>	MPI over Chameleon
<b>TGCC</b>	Très Grand Centre de Calcul
<b>RTC</b>	Real Time Clock
<b>TSC</b>	Time Stamp Counter

<b>HPET</b>	High Precision Event Timer
<b>RMA</b>	Remote Memory Access
<b>ISA</b>	Instruction Set Architecture
<b>NPB</b>	NAS Parallel Benchmarks
<b>BSP</b>	Bulk Synchronous Parallel
<b>RTS</b>	Request-to-send
<b>CTS</b>	Clear-to-send
<b>PRAM</b>	Parallel Random Access Machine
<b>MPMD</b>	Multiple Program Multiple Data
<b>SPMD</b>	Single Program Multiple Data
<b>SLURM</b>	Simple Linux Utility for Resource Management
<b>STAR-MPI</b>	Self-Tuned Adaptive Routines for MPI collective operations
<b>NIC</b>	Network Interface Controller
<b>AEOS</b>	Automatic Empirical Optimization of Software
<b>CUDA</b>	Compute Unified Device Architecture
<b>OPAL</b>	Open Portable Access Layer
<b>ORTE</b>	Open Run-Time Environment
<b>CDC</b>	Control Data Corporation
<b>Pthreads</b>	POSIX Threads
<b>HDFS</b>	Hadoop Distributed File System
<b>CNK</b>	Compute Node Kernel
<b>RDD</b>	Resilient Distributed Datasets
<b>PRACE</b>	Partnership for Advanced Computing in Europe
<b>CNL</b>	Compute Node Linux
<b>CSP</b>	Communicating Sequential Processes
<b>MIN</b>	Multistage Interconnect Network

<b>Intel MIC</b>	Intel Many Integrated Core Architecture
<b>PARMACS</b>	PARallel MACroS
<b>PVM</b>	Parallel Virtual Machine
<b>NSCI</b>	National Strategic Computing Initiative
<b>FLOPS</b>	Floating Point Operations per Second
<b>PFLOPS</b>	Peta FLOPS
<b>PGAS</b>	Partitioned Global Address Space
<b>PAT</b>	Process Arrival Time
<b>OpenACC</b>	Open Accelerators
<b>ECMWF</b>	European Center for Medium range Weather Forecasting
<b>IFS</b>	Integrated Forecasting System
<b>AMR</b>	Adaptive Mesh Refinement
<b>MPI</b>	Message Passing Interface
<b>CORAL</b>	Collaboration of OakRidge, Argonne and Livermore
<b>VUB</b>	Vrije Universiteit Brussel
<b>CRESTA</b>	Collaborative Research into Exascale Systemware
<b>HPL</b>	High Performance Linpack
<b>AMPI</b>	Adaptive Message Passing Interface
<b>HCPG</b>	High Performance Conjugate Gradient
<b>ASIC</b>	Application Specific Integrated Circuit
<b>TH-2</b>	Tianhe-2
<b>UHasselt</b>	Universiteit Hasselt
<b>OpenMP</b>	Open Multi-Processing
<b>OpenCL</b>	Open Computing Language
<b>FPGA</b>	Field Programmable Gate Array
<b>GPU</b>	Graphics Processing Unit

<b>GPGPU</b>	General-Purpose Graphics Processing Unit
<b>DSP</b>	Digital Signal Processor
<b>EDA</b>	Exploratory Data Analysis
<b>NASA</b>	National Aeronautics and Space Administration
<b>API</b>	Application Programming Interface
<b>CAAR</b>	Center for Accelerated Application Readiness
<b>iPIC</b>	Implicit Particle-in-Cell
<b>GC</b>	Grand Challenges
<b>BOINC</b>	Berkeley Open Infrastructure for Network Computing
<b>p2p</b>	Point-to-point
<b>CFD</b>	Computational Fluid Dynamics
<b>CPU</b>	Central Processing Unit

# Chapter 1

## Introduction

---

Electronic computers have evolved tremendously since their humble beginnings some 65 years ago. Fueled by doubling of integration density every 18 months, modern microprocessors sport several billion transistors. This is an astonishing million times more than that of the first microprocessor introduced by Intel in 1971. Yet, while Moore’s law is still very much alive, it has been more than a decade since microprocessors have hit a power wall limiting their operating frequency to about 5 GHz. Despite this, microprocessor computational power has continued to increase unabated. This was made possible by mutual integration of microprocessors, to form miniaturized supercomputers consisting of dozens of individual units that work in parallel to solve ever larger problems. Despite decades of exponential growth in computational power, there still exist scores of fundamental problems that vastly surpass the computational resources of modern supercomputers.

First elaborated in the late 1980s, Grand Challenges (GC) were part of U.S. policy aimed at advancing High Performance Computing (HPC)<sup>1</sup> research in response to foreign competition. Defined by one U.S. agency as “fundamental problems of science and engineering, with broad applications, whose solution would be enabled by high-performance computing resources...” [135]. Originally, GC focused on complex simulations of natural and technological systems, such as large space bodies, weather and climate, molecular interactions, electric power grids, fluid dynamics with focus on aerofoils and hydrofoils, drug design and development and speech and image recognition. The strategic importance of HPC in maintaining a nation’s economic competitiveness and national security has recently been re-established by President Obama’s Executive Order (July 2015) that established the National Strategic Computing Initiative (NSCI) as a means of ensuring that nation’s current lead in HPC. This statement comes as a direct response to declarations of China and Japan to field their first exascale supercomputers by 2020.

---

<sup>1</sup>The term “High Performance Computing” refers to systems that, through a combination of processing capability and storage capacity, can solve computational problems that are beyond the capability of small to medium-scale systems.

HPC performance has traditionally been gauged in terms of numerical crunching power, expressed in Floating Point Operations per Second (FLOPS). In the coming five years, the goal is to build supercomputers capable of one EFLOPS ( $10^{18}$  FLOPS). This great advance in computational power is not only to be brought about by ever larger advances in integration but also by major breakthroughs in computational power efficiency (FLOPS/W). That advances in HPC might shift away from aiming at ever higher FLOPS, is implied by the conclusion of President Obama's Council of Advisors on Science and Technology that HPC "must now assume a broader meaning, encompassing not only FLOPS, but also the ability, for example, to efficiently manipulate vast and rapidly increasing quantities of both numerical and non-numerical data". In the last decade, a new class of HPC systems, sometimes referred to as hyperscalers has emerged to collect and analyze vast quantities of data arising from Internet web pages, social media, scientific instruments and video and audio sensors. The problem tackled by these systems has come to be known as "big data" and is now approaching scales measured in exabytes.

Programming these large computer systems has turned out to be a major challenge. The reasons for this are manifold. Established parallel machine programming models still remain relatively low-level and consequently limited in their expressibility and robustness to design changes. The development of each HPC application requires meticulous platform measurements and tuning to achieve maximum attainable performance on the target platform. Migrating the code to a new platform often mandates that much of this process be repeated, not only to ensure optimal performance but also that the same results are reproduced. This makes HPC application development expensive and accessible only to those with extensive parallel programming skills, something that many scientists and engineers currently lack.

The distributed memory programming model has emerged as the main paradigm in HPC application programming, perhaps best embodied in the Message Passing Interface (MPI) established in the 1990s [125]. Applications written in this style have been successfully scaled to hundreds of thousands of cores. The MPI interface has undergone several expansions, building upon two decades of extensive HPC community experience. It is considered today as the de-facto standard in modern HPC application development. To take better advantage of modern multicore hardware, MPI is often augmented with some form of shared memory programming paradigm, such as Open Multi-Processing (OpenMP). The recent prominence of accelerators in HPC systems such as Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs) and Digital Signal Processors (DSPs) has seen increased use of Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) Application Programming Interface (API)s in conjunction with MPI.

Shared memory programming models, while promising superior performance due to being better matched to current multicore hardware, are notoriously dif-

difficult to scale to levels required by HPC application requirements. This appears to be a problem of fundamental nature. Sharing large memory among scores of computational cores inevitably imposes an implicit linear order on the memory read and write operations. Synchronizing concurrent operations is a hard problem, making programming in such a model a very difficult undertaking. Asynchronous message-passing, as offered by MPI or the programming language Erlang are proven alternatives that scale, but are however very explicit and fairly complicated. There is currently much research undertaken in the design of simpler and more expressive parallel programming models.

One promising example is the implicit parallelism of propagation networks [154], where computation is intrinsically concurrent and time is conceptually avoided by computing partial information instead of classical information. In propagation networks, local and stateless machines are interconnected with stateful storage cells and the computation proceeds through the propagation of information across the network. The crucial novelty here is that the storage cells should not store values, but rather accumulate information about a value: hence the distinction between classical and partial information. It is in this manner that partial information supports multidirectional, i.e. non-sequential programming, which leads to natural support for parallel computation.

## 1.1 Challenges of Exascale Computing

Exascale computing is a lofty target, currently hotly pursued by the HPC community worldwide. A machine with the numerical computational power of  $10^{18}$  FLOPS, would roughly match the processing power of the human brain on the neural level and is established as the target power of the *Human Brain Project*. A recent study commissioned by National Aeronautics and Space Administration (NASA) determined that an exascale supercomputer could allow the incorporation of a complete turbulence model, as well as more dynamic flight conditions in their simulations.

To name just one more of the myriad computational challenges whose solution promises to revolutionize the world as we know it, let us direct our gaze at the problem of whole genome sequencing. Physical DNA sequencing has recently surpassed in scaling Moore's law and has brought the price of raw megabase of DNA sequenced under 0.1 USD and that of the human genome close to 1000 USD. This cost threshold is widely viewed as an economic tipping point that will lead to wide-spread use of whole genome sequencing in medicine. However, there is another aspect of the cost equation that has not been keeping up in pace.

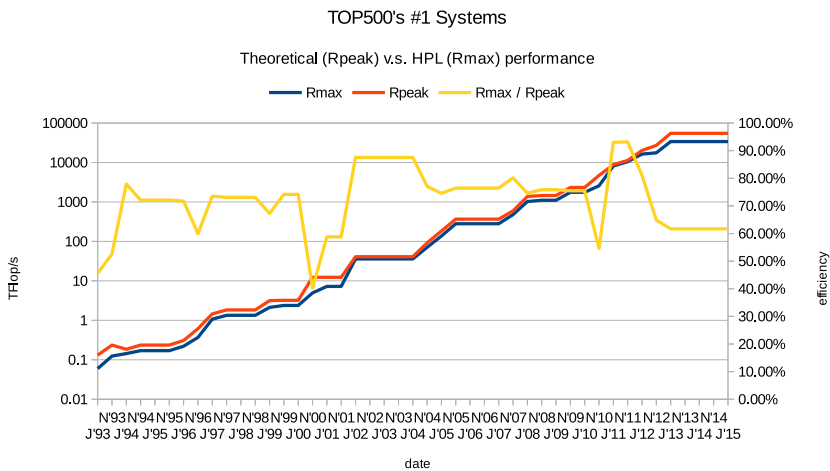
Genome analysis and assembly, genome and chromosome annotation, sequence search and clustering, function prediction, motif discovery all require, given the huge

data size, significant amounts of processing. Currently the energy cost involved in performing the computational analysis significantly outweighs the cost of genome sequencing. Development of exascale machines, together with the incorporation of new technologies necessary to increase the computational energy efficiency such as 3D chip-stacking, optical interconnects and near threshold logic, will drive down operational costs to enable the widespread use of the vast quantities of information gathered in metabolomics, proteomics and genomics.

The road towards exascale is beset by three major obstacles: scalability, power and reliability. An exascale system is projected to contain in the order of  $10^8$  cores. This is two orders of magnitudes more than the currently most powerful petascale system (Tianhe-2 (TH-2)). This unprecedented level of parallelism has to be supported not only by the applications themselves, but also by the entire software ecosystem. Current petascale systems have a power consumption approaching 20 MW (TH-2). Increasing the current performance to exascale levels will require substantial innovation in order to keep the power consumption within an acceptable budget. The availability of (cheap) power is increasingly determining the location where new supercomputers are installed and data centres are built. It is estimated that an exascale machine, consisting of millions of processing units will exhibit failures on an hourly basis. Hence, fault tolerance must be a primary design goal of such a system. Providing fault tolerance through hardware redundancy is a well-understood technique but it is hardly feasible due to the added power consumption. Therefore, software-based fault tolerance (possibly hardware assisted), is the most promising way forward. The goal is to guarantee completion of programs even in the presence of failing components. This is precisely one of the weak points of MPI and a crucial feature behind the adoption of MapReduce and HADOOP by hyperscalers [159].

With the slowing of Moore's law cadence for the doubling of integrated circuit density from 18 to 30 months (this is how long it took the semiconductor giant and fabrication leader Intel to get through the last process node transition from 22nm lithography to 14nm lithography - that this is not merely a fluke was asserted by the Intel CEO Brian Krzanich in July 2015: "The last two technology transitions have signaled that our cadence today is closer to two and a half years than two" [94]), it is becoming increasingly more difficult and expensive to increase the performance of modern supercomputers. Exascale computers were originally envisaged to come online by 2018, which was later pushed to 2020, but lately there are more and more voices murmuring that such a system may not appear before 2023. According to Jack Dongarra from University of Tennessee, and one of the curators of the Top500 list, slowing of Moore's law is not the principal problem: "It comes down to money. It's not a question of anything else but funding." Fewer and fewer labs are now able to afford the staggering costs of petascale machines. The world's fastest supercomputer, TH-2 cost 390 million USD to build, and several million USD a year just for the

electricity expenses. While the change in aggregate raw performance between Top500 list publications has doubled each year for much of its history, the development rate started tapering off around 2008. Between 2010 and 2013, the aggregate increases ranged between 26% and 66%. And between the last (June 2015) and penultimate list (November 2014) there was a mere 17% increase [174]. Heretofore unprecedented is the late plateauing of the #1 system in Top500 list for the last two years (Fig. 1.1).



**Figure 1.1:** Peak performance of #1 machines from the Top500 list. *Rpeak* denotes peak theoretical performance, while *Rmax* denotes maximum performance attained in the HPL benchmark. Adapted by permission from “Performance and Efficiency Analysis of Modern Accelerators – Fine-Grained Parallelism on the Intel Xeon Phi”, by Robrecht Dewaele. Master thesis, Vrije Universiteit Brussel, June 2014.

Another major challenge facing the deployment of exascale systems is overcoming the low performance utilization level of current machines: the highly optimized High Performance Linpack (HPL) benchmark [30] only succeeds to employ 61.7% of the world’s fastest Top500 supercomputer to date (June, 2015), TH-2. There are pessimists in the HPC community who predict that the first exascale machines are going to deliver only one or two per cent of their theoretical peak performance, when they run realworld applications. Jack Dongarra, from the University of Tennessee was quoted saying: “You’re not going to get anywhere close to peak performance ... Some people are shocked by the low percentage of theoretical peak, but I think we all understand that actual applications will not achieve their theoretical peak.”. This potentially very expensive waste of resources puts all the more emphasis on

developing better and more efficient software solutions.

Given the shortcomings of the HPL benchmark to predict realworld application performance, several different benchmarks have recently arisen. One attempt at creating a more relevant metric for ranking HPC systems is the High Performance Conjugate Gradient (HCPG) benchmark project. This benchmark was developed after observing that many modern HPC codes have diverged in their computation and data access patterns from HPL. Many modern HPC applications are governed by differential equations, which tend to exhibit higher bandwidth and low memory latency requirements, due to higher relative ratio of irregular data access patterns. According to Micheal Herroux from the U.S. Sandia National Laboratories, one of the three principal architects of HCPG, the HPL and HCPG form bookends of a performance spectrum: “Between the two is the likely speed of an application. The closer the two are, the more balanced the system”.

The Gordon Bell Prize was established to recognize and award each year outstanding software achievements in HPC, focusing on innovation in applying HPC to applications in science, engineering and large-scale data analytics. Prizes may be awarded for peak performance or special achievements in scalability and time-to-solution on important science and engineering problems. The winners in 2013 reached 55% efficiency (10.99 PFLOPS) in their high throughput simulation of cloud cavitation collapse on 1.6 million cores of the *Sequoia* supercomputer [163]. For comparison, the same system runs HPL at 85% efficiency. The 2014 year award was given to the designers of the Anton 2 special-purpose supercomputer consisting of a large number of Application Specific Integrated Circuit (ASIC)s [168] combined into a 512 node network, enabling the system to simulate a standard 23,558 atom benchmark system at a rate of 85 $\mu$ s/day, which is 180 times faster than any previously obtained supercomputer obtained results.

Emphasizing the crucial importance of energy efficiency of HPC systems, since November 2007, the Green500 list has been classifying top HPC systems based on their Top500 performance and published energy requirements. As of June 2015, the top system on the Green500 list is the Shoubu supercomputer from RIKEN, with the power efficiency of 7.031 GFLOPS/W. However, even if it was possible to somehow scale this system today to the level of an exascale machine while maintaining the same level of energy efficiency, such a machine would still require a whopping 142.8 MW of power.

### 1.1.1 Exascale applications

There is currently a lot activity in producing exascale workloads necessary to steer the design of an exascale supercomputer. To that end various initiatives have been established, some of which focus on porting existing codes to pre-exascale systems such as the last year announced Summit system, a 300 Peta FLOPS (PFLOPS)

supercomputer. The U.S. Department of Energy and Center for Accelerated Application Readiness (CAAR) program have selected 13 partnership programs to be redesigned, ported and optimized for Summit's hybrid CPU-GPU architecture. The selected codes cover a wide range of scientific domains including climate modelling, relativistic quantum chemistry, astrophysics, plasma physics, cosmology, biophysics, nuclear physics, materials sciences, seismology and combustion science.

Collaboration of OakRidge, Argonne and Livermore (CORAL) is a joint procurement activity among three of the U.S. Department of Energy's National Laboratories launched in 2014 to build state-of-the-art high-performance computing technologies, including the procurement of HPC systems such as Summit and Sierra built upon IBM Power9 CPUs, NVIDIA's Volta GPUs and Mellanox's Interconnected technologies.

Collaborative Research into Exascale Systemware (CRESTA) was established with funding from the European Union to help lay the groundwork for exascale applications. The project has selected six applications for exascale focused research from a broad range of domains including Computational Fluid Dynamics (CFD), numerical weather prediction, biomolecular systems, fusion energy and physiological flows. Through co-design it endeavours to develop the software required to support these applications at the exascale. Placing major focus into developing better software tools, CRESTA is exploring new programming models, such as Partitioned Global Address Space (PGAS) languages and Open Accelerators (OpenACC), as well as optimizing computational libraries for FFTs and sparse matrix operations. Another research vector is the introduction of fault tolerance both in the applications and in the communication libraries. The European Center for Medium range Weather Forecasting (ECMWF) uses the Integrated Forecasting System (IFS) model to provide medium-range weather forecasts to its 34 European member states. The global grid size for simulations, currently based on a 16 km resolution, is expected to be refined down to a 2.5 km global weather forecast model by 2030, when employed on an exascale system. This means that IFS needs to run efficiently on a thousand times more cores. Advances achieved by CRESTA have enabled IFS to harness 200,000 CPU cores on the Titan supercomputer at Oak Ridge National Laboratory. This is the most cores ever used for a weather model and it marks the first use of the 5 km resolution model that will be needed in medium range forecasts in 2023 [191].

As part of its major R&D investments in Europe, Intel established so-called Exascale Labs in Jülich, Leuven, and Paris during 2010. In 2011, an additional Exascale Lab was founded in Barcelona. These labs are collaborations with leading European HPC research organizations, and their goal is to address the challenges of exascale computing.

The Intel Exascale Lab in Leuven, Belgium is where I commenced the research behind this dissertation as part of the visualization work group formed by researchers from Vrije Universiteit Brussel (VUB) and Universiteit Hasselt (UHasselt). In the

following section, I will elaborate on the scientific problem that led up to the research problem addressed by this dissertation.

### 1.1.2 Space Weather and in-situ visualization

Space weather comprises Sun's surface conditions, the solar wind and the interplanetary space and the near Earth space environment, the magnetosphere, ionosphere and thermosphere [Definition adapted from the U.S. National Space Weather Plan [50]]. The main drivers of space weather are solar flares, coronal mass ejections and solar energy particle events [166]. When the solar wind arrives at Earth, the plasma is decelerated and deflected around the dipolar field centered in our planet's kernel. Where the pressure of the supersonic solar wind equals the magnetic pressure, the bow shock is generated and a substantial fraction of the kinetic energy of plasma particles is converted into thermal energy. Space weather can directly influence the performance and reliability of space-borne and ground-based technological systems such as satellite operations, communications, navigation and electric power distribution grids. Some of the space weather effects, can be dangerous and may cause direct damage like excessive electrical charging of spacecraft and intense geomagnetically induced currents in electric power networks and pipeline systems. Also, the quality of wireless communication and navigation may be affected, which can influence safety and security in certain cases [104].

Due to the severity of these effects, daily predictions of space weather effects on Earth are being published. Currently, the task of a space weather forecaster is to synthesize incoming information (satellite in-situ data, satellite images, ground-based images) and to produce a handful of parameters (geomagnetic indices, flare probabilities, EUV flux) that characterize the space weather effects on Earth in the coming days. Simulation runs are used as a validation, but the accuracy of today's predictions is low [37]. *The complexity of multi-scale and multi-physics models requires exascale computing for realistic simulations with a predictive value.*

Models of plasma system simulations broadly fall into two separate categories: kinetic (microscopic) and fluid (macroscopic). Currently, state-of-the-art kinetic models require two orders of magnitude more resolution to reach the scale at which they become relevant to macro simulations. Fluid models, that attempt to describe macroscopic behavior of plasmas, cannot be simulated at enough resolution to resolve those regions where small scale physics is important. At least two orders of magnitude more resolution is necessary than currently attainable. The complexity of these multi-scale and multi-physics models requires exascale computing power for realistic simulations with a predictive value [37].

Starting with 2010, an Intel Exascale Lab was established at IMEC, Leuven with the goal of producing an exascale proto-application that would simulate space weather. Through co-design this proto-app would bring about advancements in

programming methodologies such as failure resilient load balancing, a PGAS library, an architecture simulator and an in-situ tightly coupled visualization. The proto-application was to be based upon an Implicit Particle-in-Cell (iPIC) 3D model, originally developed by Stefano Markidis [115]. The advantage of the implicit method is the ability to select local resolution levels according to the needs of local processes, increasing the resolution only where absolutely needed. However, even with this method, the current petascale supercomputers can only model subsections of a typical space weather event. Assuming that a single core can fit at most  $16 \times 16 \times 16$  cells each with hundreds of particles for each species (electrons and ions) it is conjectured that approximately 2 million cores will be necessary for a fully predictive simulation.

The challenges of visualization in the exascale era are expected to be similar to those for petascale computing [198], but even more extreme. A major challenge will be to allow a scientist to observe a simulation as it proceeds, at the appropriate resolution required by the problems motivating exascale computing. Interactive on-the-fly visualization will be indispensable for simulation debugging and monitoring. Interactivity will allow run-time visualization steering: the selection of the visualization types and settings important for data discovery and analysis, either manually or through future cognitive methods, which reveal the information present intrinsically in the simulation data, but hidden to a human observer. Multiple, linked visualizations will be appropriate for the better understanding of the exascale simulation results. Remote, collaborative, visualization will need to be supported for the efficient use of exascale computer systems, which will be probably few and present in the remote locations.

Space weather prediction in particular needs the following types of visualization:

1. Particle phase space cuts: the visualization of particle density as a function of velocity and/or location components, by means of 2D or 3D histograms.
2. Magnetic field flux lines (frozen time), and particle trajectories (time-varying).
3. Volume visualization (iso-surface rendering being a special case) of average properties in Adaptive Mesh Refinement (AMR) cells, such as physical particle density, temperature, temperature gradients, velocity, intensity of the magnetic field, and the energy of the magnetic field, as functions of locations in the 3D solar system. AMR cells are expected to vary dramatically in size, being highly refined near shock waves for instance.
4. Virtual telescopes: volume visualization taking into account time lag, in order to compare the simulation results with the real telescope observations.

Visualization is typically performed upon simulation completion. A common practice is to down-sample simulation results to fit into the memory of a graphics

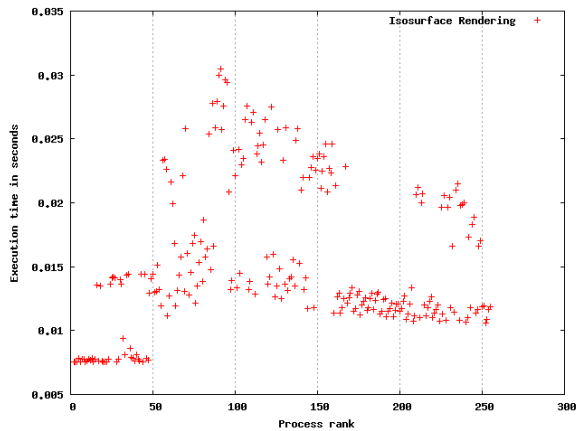
workstation used for the visualization task. Parallel visualization environments, in which the visualization calculations are performed on the supercomputer infrastructure itself are becoming more prominent. The visualization is however, still performed as a separate step after the simulation is done. It is obvious that such a step requires the movement of massive amounts of simulation data. Typically, this is done via a file interface, and it is considered to be cumbersome and inefficient. An alternative approach, that works around the data transfer bottleneck is in-situ visualization. In-situ visualization operates on-the-fly on the simulation data while it still is in memory.

We can distinguish two principal types of in-situ visualization:

**Tightly coupled [160]** : visualization has direct access to simulation data and the visualization code is run on the same nodes as the simulation. The visualization competes for the same resources available to the simulation and typically has to adapt to the data structures optimized for the simulation code.

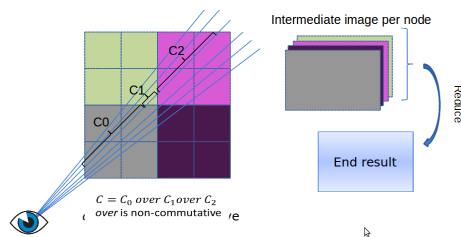
**Loosely coupled [198]** : visualization and simulation run concurrently and access data over the network. In a pull-driven approach, the visualization asks the simulation for a new time step when post-processing is finished, while in a push-driven approach a new time step produced by the simulation creates a new post-processing operation. Since this technique requires separate resources, memory copies between simulation and post-processing hosts are necessary and data transfers are limited by the network capabilities.

Compared with traditional postprocessing visualization, in situ visualization has to overcome unique challenges. The principal difficulty stems from the intrinsic requirement that the visualization code interacts directly with the simulation code. To optimize memory usage, simulation and visualization codes must share the same data structures. Because of this, balancing the visualization workload becomes more difficult because the visualization is now tightly coupled to the simulation architecture and frequent data migrations are prohibitively expensive. For stand-alone processing, researchers can parallelize visualization algorithms by partitioning and distributing data to best suit their visualization needs. In contrast, for in situ visualization, the simulation code dictates data partitioning and distribution [198]. The great challenge then becomes to balance the visualization workload so that the visualization is as scalable as the simulation. Failing to do so will result with sub par performance that can further aggravate the performance of subsequent computational phases (Fig. 1.2). A typical parallel rendering solution is composed of several stages. These include data partition, distribution, rendering, final image composition, and image delivery. Among the three basic parallel rendering approaches, namely sort-first, sort-middle, and sort-last [129], sort-last parallel rendering has been widely used by visualization researchers due to its simple task decomposition



**Figure 1.2:** Isosurface render times on the Lynx cluster. Number of nodes  $N = 32$ , number of processes  $P = 256$ . Tightly coupled visualization, with no load balancing.

for achieving load balancing. With the sort-last approach, the rendering stage scales very well since no communication overhead is involved [199]. An important component in distributed volume rendering is image compositing, i.e. back-to-front alpha blending of partial images (Fig. 1.3). This however demands inter-process communication, and in very large systems can dominate render times, becoming the principal performance bottleneck in distributed rendering solutions.



**Figure 1.3:** An illustration of image compositing in sort-last distributed rendering. The associative non-commutative operator over (C) is successively applied to produce a back-to-front alpha blend of partial images.

### 1.1.3 Global reduction operations and process desynchronization

In a typical distributed rendering solution, image compositing immediately follows image rendering. Image compositing, together with many other visualization

algorithms such as radiosity, volume rendering, isosurface extraction, streamline algorithms, etc., may be characterized as a **reduction operation**. Reduction is a common operation in serial and parallel computing, where a combining associative operator is progressively applied on linearly iterable data structures such as arrays or lists. Its performance often plays a critical role in parallel applications. For example, CFD applications typically use some form of conjugate gradient methods to solve partial differential equations, and kernels of these solvers are often expressed through global reduction operations.

A five year long profiling in production mode at the University of Stuttgart showed that more than 40% of the execution time of MPI routines is spent in the collective communication routines `MPI_Allreduce` and `MPI_Reduce` [152]. Of the total communication volume of LeanMD (a molecular dynamics benchmark), 51.18% can be attributed to reduction [118]. Similar findings have been reported by [46] for several prominent HPC applications like CTH [102], SAGE [1] and POP. Widener et al. [188] report more than 80% of application time spent in `MPI_Allreduce` for HPCCG, LULESH and MiniFE HPC applications.

Therefore, a great deal of research has been conducted to design optimized algorithms and produce efficient implementations for the reduction operation. However, most state-of-the-art reduction algorithms remain optimized only for the case where processes call (arrive at) the collective operation simultaneously. Such Process Arrival Times (PATs) are said to be balanced. Yet, balanced PATs are rare and generally only occur immediately after calls to synchronization routines.

That process arrival times can have an impact on the performance of collective operations has been largely overlooked by the research community. This is supported by the general dearth of work on imbalanced PATs and their effects on application performance. One explanation for this is the still prevailing perception that imbalanced PATs and optimization of collective operations are disjunct problems that can be addressed independently. However, as we will proceed to elaborate, solving these two problems separately is not always possible.

### 1.1.4 State of the art & challenges

Algorithms that solve the global reduction problem can be classified into two categories: *atomic* and *non-atomic*. The former solve the reduction problem when input data is not divisible, while the latter solve the problem when the input data can be arbitrarily segmented. Because the latter allows for more opportunity to load balance the computational part of the reduction problem, algorithms in this class tend to be significantly faster than those in the first class, if the input data is large. However, the former play an indispensable role in the presence of custom user input data, where segmentation might not be safely performed. Thus all MPI libraries feature both atomic and non-atomic algorithms in their implementation of global reduction

operations.

This dissertation chiefly concerns itself with finding solutions to the *reduction problem*. A particular variant of the reduction problem is considered, where the network is assumed to be homogeneous, and where individual compute nodes are connected with single port full-duplex communication links. Furthermore, it is assumed that the system does not support overlap of communication with computation and that on any given node only one process is executing.

These assumptions place a strong constraint on the space of possible solutions to the reduction problem. Nonetheless, this assumption is often adopted in literature and is considered relevant to systems whose nodes feature multicore compute architectures. There, the multicore parallelism is exploited through shared memory programming techniques. Applications written in this style are said to be *hybrid*, i.e. MPI+X, where MPI is used to address the distributed memory parallelism and X is used for the on-node shared memory parallelism. As of late, the MPI+MPI approach, where the shared memory parallelism is also tapped through MPI, is becoming more prominent. Recent trends in HPC systems, where accelerator cards, typically in form of GPUs, are providing the bulk of the computational power of the highest ranking Top500 systems, indicate that the hybrid programming models such as MPI+X are and will likely continue to remain the de facto standard in HPC computing, largely vindicating the choice of the reduction problem.

When input data is atomic, and when PATs are balanced, optimal algorithms are known. For the variant of the reduction problem examined in this dissertation, the optimal algorithm is the Binomial Tree reduction [111]. For systems that support overlapping of computation with communication, Fibonacci tree algorithms are known to be optimal [111, 97].

For non-atomic input data, as of time of writing this dissertation, no optimal algorithm is known. The best performing algorithms present in literature are roughly 2-approximations of the optimal solution in two of the problem domains (network latency, network bandwidth, computation) and optimal in the remaining two. These algorithms will be discussed in more detail in Chapter 3 and will be compared against the algorithm proposed in this dissertation.

There are two principal causes of imbalanced PATs or process desynchronization: system noise and load imbalance. System noise has been shown to have a detrimental performance impact on applications written in bulk synchronous style [56, 38] as the process with the longest completion time dictates the runtime of each stage. This effect is particularly pronounced on large scale systems. However, it can be curtailed to a large degree by fine tuning operating system kernels and system architecture parameters [38, 46]. Another strategy proved viable is to use *non-blocking collectives* to mitigate the impact of system noise if the amount of overlap an application is actually able to achieve between computation and collective propagation is sufficiently large [47].

Non-blocking collectives originated from non-blocking point-to-point operations and were formally introduced to MPI in the MPI 3.0 standard [125]. Prior to that, non-blocking collectives were available to the user through third-party libraries, such as the NBC library developed by Hoefer et. al [74, 71]. The principal benefit of non-blocking collectives stems from the fact that in many applications the communication between processes presents an overhead, in the sense that it does not contribute to the progression of the solution to the problem. Non-blocking collectives enable the user to mitigate this overhead by performing some useful work concurrently with the communication. However, this approach requires that the application or the algorithm support overlapping of communication with computation and that this computation is independent of the communication taking place. This will often require the rewriting of the application or the core algorithms, and the benefit of this strategy will hinge on the ratio of the overlappable independent computation and the communication overhead. In this dissertation, we investigate the application of non-blocking collectives to mitigate the problem of process desynchronization manifested in imbalanced PATs at the invocation points of collective reduction operations. This strategy is compared with the approach presented in this dissertation.

Load imbalance results from suboptimal data partitioning across computational nodes. Load balancing is a well understood problem and many schemes are actively employed in HPC systems, such as the Adaptive Message Passing Interface (AMPI) virtualization based approach [85]. However, successful load balancing requires information that might not always be available upfront, if the load imbalance is dynamic in nature. Even when information to perform perfect partitioning exists it may still be prohibitive to do so: for example, when the cost of data movements required for rebalancing outweighs the potential benefit. This often occurs in tightly coupled in-situ visualizations. There, both the simulation and the visualization share the same data structures, but the workload is load balanced only for the needs of the simulation and the visualization remains imbalanced (Fig. 1.2). In this situation, an alternative solution would be to employ an **imbalance robust** collective operation. If load imbalance results with process arrival times that follow a consistent and therefore predictable pattern, it might be possible to adapt the collective operation communication so as to absorb or hide much of the imbalance. While there have been efforts to seamlessly integrate load balancing into MPI, such as AMPI [85], there has been some, but still insufficient effort in designing collective operations that would be more robust to imbalanced PATs [113, 6, 140, 151, 139].

Inspired by the pioneering work of Mamidala [113] and Faraj [6], and their successors [140, 151, 109, 139], this dissertation proposes several new imbalance robust reduction algorithms.

The principal idea remains the same: the static communication schedule of high-performance algorithms for balanced PATs is re-ordered to optimize the reduction operation in presence of imbalanced PATs. The re-ordering ensures that the pro-

cesses that are the first to arrive can start communicating immediately, out-of-order prescribed by the static communication schedule.

The only prior work on the imbalance robust reduction operation, known to the author, is that of Parsons et al. [139], where the authors present an imbalance robust reduction algorithm for hierarchical memory systems where MPI is the sole method of parallelization, i.e. the MPI+MPI approach previously elucidated. Due to this fundamental difference, it is not trivial to compare their results with the algorithms proposed by this dissertation. This is further compounded by the fact that they have adopted a different metric to report algorithm performance, one that is biased towards reporting larger speedups compared with the one adopted in this dissertation, as is discussed in more detail in Chapter 3.

In this dissertation, three new reduction algorithms are presented, analyzed and benchmarked. Two of the algorithms solve the reduction problem for atomic input data, while the third algorithm solves the reduction problem for non-atomic data. Two of the three presented algorithms use side-information, in the form of PAT prediction to construct optimized reduction schedules. These are what we will be calling *clairvoyant* reduction algorithms. Finally, one of the proposed algorithms is proven optimal.

A major challenge for imbalance robust algorithms is to remain competitive when PATs are balanced. When clairvoyant algorithms are employed, the cost of schedule construction can represent a substantial fraction of the total time to solve the reduction problem.

## 1.2 Outline and summary of scientific contributions

In Chapter 2, we provide a comparison of shared memory and distributed memory computing models. We then proceed to elaborate on distributed memory computing by outlining the salient features of the MPI interface. Following that, we define the *reduction problem* which represents the research crux of the dissertation. The chapter is concluded by a detailed survey of several state-of-the-art reduction algorithms that were included in the comparative performance survey conducted in latter chapters.

In Chapter 3 we elaborate further on the nature of load imbalances and define PATs. We establish an upper limit on imbalance resiliency of reduction algorithms. Several models of collective operation complexity analysis are investigated and the adoption of a linear cost model is argued. From Chapter 4 the contributions of this dissertation are detailed.

In Chapter 4 we introduce the benchmarking suite developed for the purpose of performance evaluation of MPI collective operations under various patterns of process desynchronization. Current approaches and shortcomings of MPI mi-

crobenchmarks are discussed in detail. The chapter is concluded with a discussion on statistical methods adopted in a posteriori time series analysis of the data produced by the performance microbenchmark.

In Chapter 5 we consider the reduction problem on atomic messages with process desynchronization. A new atomic reduction algorithm is introduced and proven optimal under the assumption of fully connected full duplex no communication-computation overlap network. This algorithm is contingent on pre-knowledge of PATs at the reduction operation call site, hence its name *Clairvoyant*. It preconstructs the reduction schedule according to this side information. Additionally, a dynamic process desynchronization reduction algorithm is introduced that does not require pre-knowledge of PATs, and its performance analyzed. The algorithms were evaluated on a 128 node subset of the Partnership for Advanced Computing in Europe (PRACE) Curie supercomputer. The implementation of the optimal algorithm reached a maximum measured speedup of 1.67 compared with Binomial Tree and outperformed it in virtually all tests. The dynamic algorithm measured a maximum speedup of 1.49 and converged in performance with the implementation of the optimal algorithm for large measures of imbalance.

In Chapter 6 we consider the reduction problem on non-atomic messages with process desynchronization. We introduce a new pipelined algorithm that using pre-knowledge of PATs preconstructs reduction schedules that are of minimal possible length given a particular PAT pattern. For balanced PATs, this algorithm has been empirically shown to produce schedules of optimal length  $N + n - 1$ , where  $N$  is the number of segments the input data is divided into and  $n = \log_2 P$ ,  $P$  being the number of processes participating in the collective operation. To evaluate its performance, we implement four commonly used algorithms: binomial tree reduction, butterfly, parallel ring and Radix-k. The performance of selected algorithms was empirically evaluated on a 128 node subset of the PRACE Curie supercomputer. The reported results show that the new imbalance robust algorithm universally outperforms all the selected algorithms, whenever the reduction schedule is precomputed. We find that when the cost of schedule construction is included in the total execution time, the new algorithm outperforms the selected algorithms for problem sizes greater than 1MiB.

In Chapter 7 we investigate how the new imbalance robust algorithms proposed in this dissertation compare to non-blocking collectives in mitigating the impact of imbalanced PATs. We conducted the experimental evaluation on a 32 node subset of the VSC muk cluster machine. In the experiment, we varied the magnitude of the imbalance and the length of the independent (overlapped) computation. We found that for atomic input data, non-blocking collectives can match or exceed the performance of imbalance robust algorithms. However, for non-atomic input data, the imbalance robust algorithms were found superior.

Finally, Chapter 8 concludes the major findings of this dissertation and discusses

several avenues of future work lying ahead.



## Chapter 2

# Distributed memory computing and MPI

---

This chapter gives a brief introduction to the fundamental principles of distributed memory computing and enumerates the most commonly used parallel programming models. It then proceeds to analyze in more detail the Message Passing Interface (MPI) interface that forms the backbone of the great majority of contemporary and legacy High Performance Computing (HPC) applications. Following that, it discusses several important features of the MPI interface, beginning with Point-to-point (p2p) primitives and how they relate to collective operations. The guiding principles behind collective operation design are investigated together with their prevalence in modern HPC applications. The chapter is concluded with a presentation of state-of-the-art reduction algorithms, all of whom have been included in the comparative performance analysis presented in Chapter 5 and Chapter 6.

## 2.1 Parallel programming models

Parallel programming models define the manner in which programmers write and compile parallel programs. Ideally, a parallel programming model should be expressive enough to concisely implement a wide range of problems, while simultaneously being efficient in execution.

A very broad classification of parallel programming models is that of process interaction and problem decomposition [165]. In the HPC domain, process interaction has long been the driving force behind the development of parallel programming models. The two dominant process interaction models are shared memory and message passing.

Problem decomposition models can be classified as either task or data parallelism. A task parallel model is one focused on processes, or units of execution. These tasks will have defined inputs and outputs and their combination will form

an abstract parallel computer that solves a given problem. A data parallel model is one focused on the decomposition of data across different processes. Data parallelism is strongly modelled after modern computer architectures capable of concurrently executing multiple instructions and data streams. Perhaps the best known example being the Single Instruction Multiple Data (SIMD) (after Flynn’s taxonomy [48]) instruction sets present in modern Central Processing Units (CPUs) like Intel’s Streaming SIMD Extensions (SSE) or ARM NEON. Computers with this capability can perform the same operation on multiple streams of data. Data decomposition is also the key feature behind General-Purpose Graphics Processing Unit (GPGPU) computing. A prominent example of a data-parallel programming model in this domain is Nvidia’s Compute Unified Device Architecture (CUDA).

Both the message passing and the shared memory programming models have their counterparts in the physical world, with the defining criteria being the computer’s memory system hierarchy.

### 2.1.1 Distributed memory systems

Distributed memory computing arose out of necessity to solve problems that eclipsed the memory capacity of any single machine. Connecting multiple machines within a network and pooling their combined memory and computational resources allowed researchers to scale the size of their problems by using the established combining principles of computer networks. The use of affordable and proven commodity hardware, such as PCs and fast LANs and standardized software components (such as UNIX and MPI) has led to an abrupt emergence of a family of scalable parallel computing machines in the 1990s known as *clusters*. These machines have paved way to the modern landscape of HPC systems worldwide. A majority of machines on the Top500 list today can be classified as clusters.

Distributed memory computing has as its foundation distributed systems. “A *distributed system* consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software. Distributed system software enables computers to coordinate their activities and to share the resources of the system - hardware, software and data. Users of a well-designed distributed system should perceive a single, integrated computing facility even though it may be implemented by many computers in different locations.” [55].

Traditionally, distributed systems were defined by concurrency of components, resource sharing, scalability, independent failure of components and absence of global clocks. They could be further divided into *opportunistic* (such as grid computing) or *quasi-opportunistic* (such as Berkeley Open Infrastructure for Network Computing (BOINC)) depending on the resource sharing quality of service. A disadvantage of grid-like parallel computing is the absence of high-speed connections, resulting in high costs whenever data needs to be moved around. Because of this, grid-like

distributed systems are typically employed to solve a collection of separate tasks, while supercomputers, such as computer clusters, are employed to solve large and complex single task problems.

Supercomputer architectures have seen dramatic shifts since the pioneering systems introduced in the 1960s by Seymour Cray at the Control Data Corporation (CDC). Using Flynn's taxonomy [48], the early Cray systems could be described as shared memory SIMD machines. While the early machines relied on local parallelism to achieve superior peak performance, the ever increasing demand for larger computational power ushered in the age of massively parallel systems. This has created unique challenges both for hardware and software designers.

Examining the road supercomputing systems have taken from gigascale to terascale and petascale, reveals that from gigascale to exascale transistor improvements led to a speedup of 32 times, while the increased parallelism contributed an equal 32 times speedup. However, in the period of 1996 to 2006, in the transition from terascale to petascale machines, the transistor improvements accounted for only 8 times the speedup, while the increased parallelism led to a speedup of 128 times. Current estimates are that in the transition from petascale to exascale, the transistor improvements will lead to a speedup of mere 1.5 times, while the remainder will have to be derived from increased parallelism.

It has become common for modern supercomputers to run different operating systems on different nodes, depending on the services they provide. A small and efficient lightweight kernel such as Compute Node Kernel (CNK) or Compute Node Linux (CNL) is run on compute nodes, while Linux-derived kernels are typically employed on servers and I/O nodes. In pursuit of larger computational densities and power efficiencies, new data parallel computation accelerators architectures such as Graphics Processing Units (GPUs) and Intel Many Integrated Core Architectures (Intel MICs) have come to play a prominent role in the makeup of modern supercomputers. The #1 system, Tianhe-2 (TH-2) at the National Supercomputer Center in Guangzhou, China, incorporates 48,000 Intel Xeon Phi accelerators. The #2 Top500 system, Titan built by Cray at Oak Ridge National Laboratory, incorporates 18,688 Nvidia Tesla K20X GPUs. Due to a recent technological export embargo, imposed by the U.S. government [95], the TH-2 supercomputer will not see a planned upgrade composed of new generation Xeon Phi accelerators to push it above 100 PFLOPS, but rather one made up of domestically designed and produced general purpose 64-bit Digital Signal Processors (DSPs), delivering 2.4/4.8 TFLOPS of double precision/single precision computational power per unit. This should help keep TH-2 at the top at least till 2017, when the 150 PFLOPS Summit system is slated to come online.

Job schedulers are an integral feature of modern supercomputer and cluster systems. One of the best known job schedulers, Simple Linux Utility for Resource Management (SLURM) [197], an open-source fault tolerant and highly scalable

cluster management and job scheduling system for Linux clusters, is present on approximately 60% of the Top500 supercomputers, including the #1 system TH-2. It provides three key functions. First, it allocates exclusive and/or non-exclusive access to resources (computer nodes) to users for some duration of time so they can perform work. Second, it provides a framework for starting, executing, and monitoring work on a set of allocated nodes. Finally, it arbitrates contention for resources by managing a queue of pending jobs. Users interact with SLURM via command line utilities.

Perhaps the most defining feature of cluster computer systems are not their core computational units, but rather the network interconnect technology. In fact, when assessing whether the sale of a supercomputing system represents a threat to national security, the U.S government considers the "red flag" to be the nature of the interconnect [149], as this can give strong hints as to what the potential use of such a system might be. Due to very large number of network nodes in modern supercomputers, it is prohibitively expensive to install fully-connected interconnects, such as crossbar switches. Instead, various restricted direct interconnects are in wide use today in form of *mesh* or *hypercube* topologies. Mesh networks commonly come in form of 3D-tori, where each compute node has six connections to its immediate neighbours. Tree networks, in form of *fat-tree* topologies [106] have become prominent with the increasing market share of Infiniband systems. The compute nodes of a fat-tree network are located at the leaves of a complete binary tree, and the internal nodes are switches. Going up the fat-tree, the number of wires connecting a node with its father increases and hence the communication bandwidth increases. Such network configurations are typically implemented as Multistage Interconnect Networks (MINs). The common characteristic of such networks is a number of levels of switches. Switches in one level are connected to switches in adjacent levels in various symmetrical ways such that a path can be made from one side of the network to the other side [190]. MINs have a very long history and were originally developed for telephone exchanges [27].

Tree structured MINs based on central switches are now ubiquitous in HPC. Compared to other interconnect network topologies, such as tori or rings, the central-switch based structure has the advantage of being able to efficiently embed structured communication patterns while scaling the network in size, as shown by Leiserson [106]. A network with a true crossbar as its central switch would have almost ideal network properties: constant latency between all pairs of endpoints, as well as full bisection bandwidth (any half of the endpoints can simultaneously communicate with the other half at full line rate) [76]. However, in practice central switches are typically implemented as MINs. This can have implications on many communication patterns, as static routing schemes, such as those of Infiniband, can lead to significant network congestion. New interconnect technologies, such as the Intel OmniPath fabric have recognized the severity of this issue as one of the

major impediments to scaling HPC applications to exascale levels with 100k nodes. To that end, they support adaptive routing that monitors congestion in the paths between fabric end-points and periodically alters the paths taken by packets to rebalance the load across available fabric links. Their implementation should allow up to hundreds of real-time adjustments per second per switch. Combined with dispersive routing, that instead of sending all packets from a source to a destination via a single path, disperses them across multiple paths should allow the network to offer performance closer to optimal for a much wider range of communication patterns, thereby increasing its *effective bisection bandwidth* [82]. Similar features are already present in the implementation of the TH Express-2 interconnect, used by the #1 Top500 system TH-2 [108].

In the last decade, a new stream of High Performance Computing (HPC) has emerged, focused on analyzing datasets in the multi terabyte range. The size and complexity of such datasets eclipsed the capabilities of traditional data processing applications and necessitated the invention of new techniques that can take advantage of massively parallel computational systems. The MapReduce framework, pioneered by Google, has ushered in a new era of distributed memory computing where addressable problem size and fault tolerance have taken primacy over raw performance, hitherto emphasized in the HPC computing. In 2004, Google published a paper on a process called MapReduce [31] that used a distributed file-sharing framework for data storage and query. The MapReduce framework provided a parallel processing model and associated implementation to process huge amounts of data. With MapReduce, queries are split and distributed across parallel nodes and processed in parallel (the Map step). The results are then gathered and delivered (the Reduce step). Google originally implemented MapReduce to execute very large matrix-vector multiplications needed for the PageRank calculations. Matrix operations such as matrix-vector and matrix-matrix calculations fit very well into the MapReduce style of computing. The success of the framework prompted others to replicate and further build upon it. Apache's open source software framework Hadoop was designed for distributed storage (Hadoop Distributed File System (HDFS)) and distributed processing of very large datasets.

However, Hadoop's mandatory data output to disk after every MapReduce stage created huge performance issues for any problem that required iteration, interactive data mining or streaming. This has sparked innovation in in-memory post-Hadoop frameworks. Come Spark, an open source framework that in contrast to Hadoop's two stage disk-based MapReduce paradigm performs much of the iterative processing in-memory, given enough RAM. This has resulted in performance improvements of up to two orders of magnitude. Crucially, Spark retains fault tolerance thanks to Resilient Distributed Datasets (RDDs) [201]. RDDs are a new abstraction for in-memory computing, consisting of parallel data structures that can persist and be partitioned across Big Data infrastructure. On failure, RDDs rebuild lost data using

lineage: each RDD remembers how it was built from other datasets to rebuild itself.

### 2.1.2 Message Passing

The message-passing model is based on the notion of a process. One can think of a process as an instance of a running program, together with the program's data. Parallelism is achieved by having many processes co-operate on the same task. In shared memory interprocess communication, memory locations can be accessed and modified by multiple processes (programs) with an intent to provide mutual communication, such as passing data between processes. In message passing, each process only has access to its personal private memory. All communication and data sharing between processes must occur through process-to-process messages.

“Computers that rely on message passing for communication rather than cache-coherent shared memory are much easier for hardware designers. The advantage for programmers is that communication is explicit, which means there are fewer performance surprises than with the implicit communication in cache-coherent shared memory computers. The downside for programmers is that it's harder to port a sequential program to a message-passing computer, since every communication must be identified in advance or the program doesn't work. Cache-coherent shared memory allows the hardware to figure out what data needs to be communicated, which makes porting easier. There are differences of opinion as to which is the shortest path to high performance, given the pros and cons of implicit communication.” [142].

Some of the leading figures in the industry are convinced that the future processors for exascale nodes will have thousands of cores with large caches, but without coherency. Similarly, the nodes will consist of multiple processors, but without coherency across the caches (an example of such a system is the Intel Single-chip Cloud Computer [64]). In such a system, message passing will be the way cores and processes manage the memory and mutually communicate. Message passing support will exist in hardware for increased performance, while some levels of memory coherency might be supported in software instead.

A wide variety of mathematical theories have been produced to understand and analyze message passing systems, such as the Actor model [67] and various process calculi (Communicating Sequential Processes (CSP) [68]).

In concurrent systems, two kinds of communication naturally occur: *persistent* and *transient* communication. Message passing is a form of persistent communication. “In a transient communication act, the medium's state is changed only for the duration of the act, immediately afterwards reverting to its normal state. A message sent on an Ethernet modifies the transmission medium's state only while the message is in transit; the altered state of the air lasts only while the speaker is talking. In a persistent communication act, the state change remains after the sender has finished

its communication. Setting a voltage level on a wire, writing on a blackboard, and raising a flag on a flagpole are all examples of persistent communication.” [103]

A crucial property of persistent communication is that it allows the sender and the receiver to communicate at different times. Persistent communication is a necessary condition for *mutual exclusion*, which is the most prevalent form of coordination in shared memory programming models.

“At a low level, message passing is often considered to be a form of transient communication between asynchronous processes. However, a closer examination of asynchronous message passing reveals that it involves a persistent communication. Messages are placed in a buffer that is periodically tested by the receiver. Viewed at a low level, message passing is typically accomplished by putting a message in a buffer and setting an interrupt bit that is tested on every machine instruction.” [103]

### 2.1.3 Shared memory programming

Shared memory computation typically consists of multiple threads, each of which may be thought of as a sequential program. These threads communicate by calling methods of objects that reside in a shared memory. Threads are by nature asynchronous, meaning that they run at different speeds, and any thread can halt for an unpredictable duration at an time [65]. On a lower level, threads communicate by accessing shared (read-write) registers. Registers can be classified according to the consistency condition they satisfy when accessed concurrently, the domain of possible values they can hold and the number of processes that can access the *Read* and *Write* operation, which leads to a total of 24 register types [100].

All parallel programming models for shared memory computers are built upon two orthogonal concepts: the source of concurrency and the synchronization primitives used to manage the concurrency. The management of the two concepts may be implemented in hardware, software, or both, and typically may be mixed freely [127].

One of the most common standardized shared memory programming models is the POSIX Threads (Pthreads) standard. Pthreads defines a set of C programming language types, functions and constants. In all, there are about 100 Pthreads procedures, that can be categorized in four groups: thread management, mutexes, condition variables and synchronization. Any ANSI/ISO-C conforming compiler may be used to compile programs that use Pthreads.

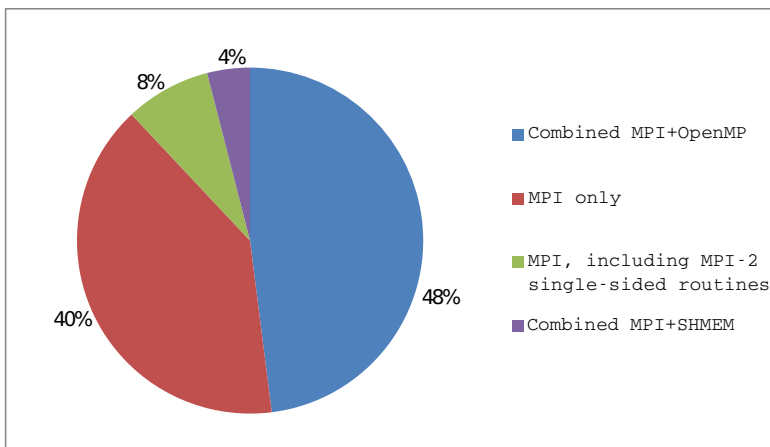
While Pthreads present a very explicit approach to parallel programming, Open Multi-Processing (OpenMP) was designed to perform a more implicit parallelization by extending programming languages (C, C++, Fortran) with directives. A programmer marks a section of code to be run in parallel with an appropriate directive and specifying the data sharing, synchronization and scheduling clauses.

Partitioned Global Address Space (PGAS) is a parallel programming model that assumes a global memory address space that is logically partitioned and a portion of

it is local to each process or a thread. In this model, variables and arrays can either be shared or local. Each process has private memory for local data items and shared memory for globally shared data items. A main tenet of PGAS languages is global view, the idea that data structures should be global and span the entire address space. This however comes at the cost of little control over data layout [196].

### 2.1.4 Current trends in programming models for HPC

The Partnership for Advanced Computing in Europe (PRACE) project is tasked with implementing a pan-European HPC service and the necessary infrastructure. A PRACE deliverable was published in May 2012 [49] presenting the survey results tasked to collect information on the active HPC systems in Europe, their usage profiles and application usage details. Each of the major European HPC service providers was surveyed on applications accounting for greater than 5% of system utilization. Information was gathered relating to a total of 57 distinct applications. Fig. 2.1 shows the distribution of application parallelization methods in PRACE Regular Access projects. Judging from the survey results, the vast majority of ap-



**Figure 2.1:** Summary of responses collected by the PRACE survey on the question: "Which parallelization method does your application use?". Adapted according to fair-use from "PRACE-11P Deliverable 7.4.1: Applications and user requirements for Tier-0 systems", by Xu Guo and Mark Bull. PRACE First Implementation Project, May 2012. [49]

plications still use the traditional programming methods such as MPI and OpenMP. However, the top machines in the Top500 lists all use accelerators to achieve their impressive performance levels. As of now, writing code for GPU accelerators is most

commonly done in CUDA or Open Computing Language (OpenCL).

Compute Unified Device Architecture (CUDA) is a proprietary interface created by NVIDIA which consists of extensions to C/C++/Fortran that allow the programmer to interface to the GPU hardware to leverage their general purpose computing facilities.

Open Computing Language (OpenCL) is a cross platform Application Programming Interface (API) that can be used to program both CPUs, GPUs and other accelerators such as DSPs and Field Programmable Gate Arrays (FPGAs). Compared to CUDA, OpenCL addresses the hardware at a lower level.

Finally, as of late directive-based programming of accelerators is becoming more prominent. Instead of rewriting application code to use CUDA or OpenCL directive-based approaches leverage the compiler to generate executables that can run on GPUs, FPGAs or DSPs. OpenMP and Open Accelerators (OpenACC) are two standards that have found wide-spread use in the HPC community.

## 2.2 Message Passing Interface

In the 1980s and early 1990s there emerged several message-passing systems with differing syntax but similar goals (Parallel Virtual Machine (PVM) [3], PARallel MACroS (PARMACS) [20]). MPI was the first effort to produce a message-passing interface standard across the whole parallel processing community. Eighty people representing forty different organisations, mainly in the U.S. and Europe, collectively formed the MPI forum throughout the period of 1992-1994. Jack Dongarra, Rolf Hempel, Tony Hey, and David W. Walker put forward a preliminary draft proposal in November 1992: this came to be known as MPI-1. A two-year process of proposals, meetings and review resulted in a document specifying a standard Message Passing Interface (MPI), version 1.0, that was released in May 1994[52].

The MPI standard has significantly evolved through time, producing a number of revisions and extensions. We will briefly enumerate them here.

- Version 1.1 in June 1995 [119]. This version constituted error corrections and clarifications to the original MPI document. Overall, the changes were minor.
- Version 1.2 in July 1997 [120]. Again, a minor change to the standard. A single new function was introduced, one for identifying which version of the MPI standard the implementation conforms to. The MPI-1.2 document is contained within the MPI-2.0 standard document.
- Version 2.0 in July 1997 [121]. This standard introduced substantial changes. The focus was on process creation and management, introduction of one-sided communications, extension of collective operations and importantly parallel I/O operations.

- Version 1.3 in May 2008 [122]. Minor corrections and consolidation of MPI-1.1 and MPI-1.2.
- Version 2.1 in June 2008 [123]. This document combined the previous MPI-1.3 and MPI-2.0 documents.
- Version 2.2 in September 2009 [124]. Mostly corrections of the previous document.
- Version 3.0 in September 2012 [125]. This was another major change, with significant extensions to the standard. Notable among these were non-blocking collective operations and new one-sided operations.
- Version 3.1 in June 2015 [126].

A major goal of MPI is portability across different machines comparable to that given by programming languages such as Fortran, but not at expense of efficiency. MPI endeavours to run transparently on heterogeneous systems, i.e. collections of processors with distinct architectures.

An MPI program consists of autonomous processes, executing their own code in either Single Program Multiple Data (SPMD) or Multiple Program Multiple Data (MPMD) fashion. The latter implies that program codes executed by each program need not be identical. The MPI runtime, typically through the use of *mpirun*, will instantiate instances of supplied executables on the supplied set of host machines. Processes are combined into process groups that can form a communication context (*communicator* in MPI parlance). Each process is assigned a unique rank in all of its communicators, ranging from 0 to  $P-1$ , where  $P$  is the number of processes forming a communicator. Each process can query its rank via communicator objects.

The primary facility that MPI provides to the programmer are mechanisms for transferring data between processes. The basic communication mechanism of MPI are point-to-point operations that transmit data between a pair of processes, one side sending, the other, receiving. In general, there exist two principal message passing strategies supported by MPI.

The first, *two-sided communication* requires the receiver to post a receive request to the MPI library in order to receive a message. Associated with each message is an *envelope* that specifies the message destination and contains distinguishing information (*tag*) that can be used by the receiving process to select a particular message. Thus, each message may be identified by its envelope, the tuple (sender, receiver, tag, communicator), and is guaranteed to be matched at the receiving side in the order of sending. This last property asserts that messages are *non-overtaking*. If a sender sends two messages in succession to the same destination, and both match the same receive operation, then the latter cannot receive the second message if the first one is still pending. This property guarantees that message-passing code is

deterministic. The tag may be used to distinguish logical communication channels between equal pairs of processes.

The second communication strategy, known as *one-sided communication* provides communication that is initiated by one side only (either the receiver or the sender). This is, in essence, Remote Memory Access (RMA). Message passing communication achieves two effects: *communication* of data from sender to receiver and *synchronization* of sender with receiver. The one-sided communication support allows these two functions to be separated.

### 2.2.1 MPI implementations

Today, there are more than a dozen different MPI implementations. Here we will list only two among the most used, both of which have influenced the MPI library implementations fielded on our test machines.

#### MPICH

MPICH started as an implementation of MPI-1 developed at Argonne National Laboratory. The original implementation was based on the Chameleon portability system, designed by William Gropp, to provide a light-weight implementation layer, hence the name MPICH. MPICH forms the basis for many open source and vendor projects such as MPICH-V [17] and MVAPICH. Due to its prominence, MPICH is a highly optimized implementation, where every collective operation is implemented using at least one algorithm [178]. MVAPICH2 [87] is a high-performance MPI implementation for Infiniband and is used on many Top500 systems. MPICH was the first MPI implementation to support the new MPI-3 standard.

#### Open MPI

Open MPI is a Message Passing Interface (MPI) library project that is developed and maintained by a consortium of academic, research and industry partners. It combines technologies and resources from several other projects (FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI). It has been used by many of the Top500 machines including the previous #1 machines such as the Roadrunner and the K computer.

The Open MPI implementation is composed of the following parts: the MPI layer (OMPI), the run-time environment (Open Run-Time Environment (ORTE)) and the portability layer (Open Portable Access Layer (OPAL)). The OMPI layer implements MPI semantics while ORTE provides a resource manager, global data store, messaging layer and a peer discovery system for parallel job start up. OPAL provides a number of useful functions and data structures implemented in a portable manner, such as high-resolution timers, fast atomic memory operations, and portable I/O functionality [54].

OpenMPI features full MPI 3.0 support, including non-blocking collectives, neighborhood collectives, RMA, shared memory, etc. It also includes OpenSHMEM, a one-sided shared memory communication library offering C and Fortran bindings to PGAS programming model. It comes with CUDA support and full MPI coverage in Java.

## 2.2.2 Point-to-point operations

Sending and receiving of messages by processes is the cornerstone of MPI communication. Disclaimer: the discussion in this section is based on the standard specification in [125].

The basic point-to-point (process to process) communication primitives are **send** and **receive**. Each of the two operations has as its arguments a *send buffer* or a *receive buffer*, residing in the sender or receiver's local memory, respectively. In addition to that, a message envelope is supplied that contains distinguishing information used by the receiving process to select a particular incoming message.

**Table 2.1:** Communication facilities of MPI

Comm. pattern	point-to-point		collective		remark
Comm. strategy	two-sided		one-sided		p2p only
Progress mode	blocking		non-blocking		two-sided and collective
Comm. mode	standard	buffered	synchronous	ready	only p2p two-sided

MPI offers two kinds of point-to-point operations: *blocking* and *non-blocking*. Blocking send calls do not return until the message data and the envelope have been safely stored away so that the sender is free to modify the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer. The purpose of message buffering is to decouple the send and receive operations. The advantage is that a blocking send can complete as soon as the message was buffered, even if no matching receive has been executed by the receiver. However, message buffering may incur a performance cost, as it entails additional memory-to-memory copying, and it requires the allocation of memory for buffering. MPI takes the path of maximum portability, offering to the user the choice of several communication modes, each implementing a different communication protocol [125].

MPI offers four communication modes: *standard*, *buffered*, *synchronous*, *ready*. In *standard* mode, which is the default mode, it is left up to the MPI implementation to decide whether outgoing messages will be buffered. If the message is buffered, then

the send call may complete before a matching receive has been posted. Otherwise, if the message is not buffered, the send call will not complete until a matching receive has been posted. It is said that the standard mode send is *non-local*: successful completion of the send operation may depend on the occurrence of a matching receive. [125] In *buffered* mode, the send operation may be initiated regardless of a matching receive being posted or not. Unlike the standard mode send, this operation is *local*, i.e. its completion does not depend on the occurrence of a matching receive. If a buffered mode send operation is posted while a receive operation has not been yet posted, the MPI implementation must buffer the outgoing message, to allow the send operation to complete. In *synchronous* mode, the send operation may commence regardless of matching receive being posted. However, the synchronous send operation will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send operation. In this context, the send operation provides synchronous send semantics: a communication does not complete at either end before both processes rendezvous at the communication. It follows then that synchronous mode sends are non-local. Finally, in *ready* mode, the send operation may be started only if the matching receive is already posted. Should the receive not be posted, the operation is considered erroneous and the outcome undefined. The motivation for this communication mode is the possibility of removing the hand-shake operation that is otherwise required and which might result in improved performance.

Unlike the four different modes for the blocking send operation, there is only one mode for the blocking receive operation. The receive operation returns only after the contents of the received message have been copied into the receive buffer. It is insightful to observe that a receive operation can complete before a matching send operation has completed - however, a receive cannot start before a matching send has started.

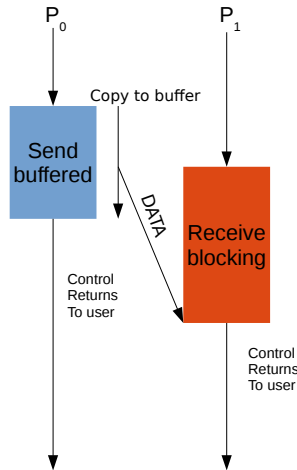
All the p2p communication mechanisms are enumerated in Table 2.1.

### **Semantics of point-to-point operations**

The MPI standard mandates that every MPI implementations adheres to a set of message passing semantics that we will shortly outline in this section.

Messages are to be *non-overtaking*. If a sender sends two messages in succession to the same destination, and both match the same receive operation (their envelopes are equal), then the receive operation cannot receive the second message if the first one has not been received. This requirement guarantees that the message-passing code is deterministic (at least in single-threaded mode).

All MPI implementations must ensure that whenever two matching send and receive operations are posted, at least one of them will complete regardless of other actions in the system. On the other hand, MPI implementations are not required



**Figure 2.2:** An illustration of a buffered send mode communication between two processes  $P_0$  and  $P_1$ . The sender, process  $P_0$  resumes control as soon as the message has been copied into a local buffer, while the receiving process  $P_1$  blocks until the contents of the message have been copied into the receive buffer.

to guarantee *fairness* in the handling of communication. This means that a process that is blocking on a send or receive operation might get resource starved, even if the destination process repeatedly posts a matching receive operation, because it is possible for it to be overtaken by another message, sent from a different source.

Lack of resources can prevent the execution of an MPI function call, leading to subtle errors. For example, a standard mode send operation can lead to a deadlock if both the sending and the receiving process are blocked due to insufficient buffer space being available. In contrast, a buffered mode send that cannot complete due to lack of buffer space will result with an error, which if not better in terms of program execution, is at least more predictable.

A program is considered “safe” if no message buffering is necessary for the program to complete. In such a program, all the send operations can be replaced with synchronous mode sends, and the program will still run correctly. This conservative programming style ensures best portability, as it comes with least requirements on the amount of buffer space available.

### Non-blocking point-to-point operations

The introduction of non-blocking communication was principally motivated by the potential of performance improvement via the overlap of communication and computation. Calling a non-blocking send operation will only start the send operation, but not complete it. A separate *send complete* call is required to complete the communication, i.e. to verify and guarantee that the data has been copied out of the send buffer. With suitable hardware, the transfer of data out of the sender memory may proceed concurrently with computations done at the sender after the send was initiated and before it completed [125]. The same applies to a non-blocking receive operation. Use of a non-blocking receive operation has the potential to avoid system buffering and memory-to-memory copying. This is because with a pre-posted receive operation, information is provided early to the system on the location of the receive buffer. Non-blocking send start calls are provided with the same communication modes as those of the blocking send operations with the same meaning. In all cases, the send call is local: it returns immediately, regardless of the status of other processes in the system.

The MPI standard motivates the use of non-blocking receives as follows: “The message-passing model implies that communication is initiated by the sender. The communication will generally have lower overhead if a receive is already posted when the sender initiates the communication (data can be moved directly to the receive buffer, and there is no need to queue a pending send request). However, a receive operation can complete only after the matching send has occurred. The use of non-blocking receives allows one to achieve lower communication overheads without blocking the receiver while it waits for the send.” [125]

### Communication protocols

Most MPI implementations use two different protocols to implement the transmission of messages: one protocol, called *eager* to transmit short messages, and the other called *rendezvous* to transmit long messages.

In the eager protocol, the sender process eagerly submits the message into the communication channel, relying on pressure from the flow-control mechanism to prevent loss of data and to throttle it if the receiver is not ready to receive the incoming data. It is now the receivers responsibility to buffer the message if a matching receive has not been yet posted. Upon receiving the message, the receiving process replies with an ACK control message.

The rendezvous protocol or the long message protocol is typically implemented by Request-to-send (RTS)/Clear-to-send (CTS) techniques. When a send is posted, it is only the message envelope that is sent to the receiving process in request-to-send message. When a matching receive is posted, a ready-to-receive message is

dispatched to the sender, who can now transmit the message. Finally, upon receiving the data, the receiving process sends an ACK control message to the sender.

Eager mode communication avoids the two additional transactions (RTS and CTS), by attempting to immediately send the message with little or no permission from the receiver. When latency is important, eager mode communication is preferred. Sending a 4-byte MPI message eagerly across Infiniband can take about 2  $\mu$ s. Adding the hand-shake of the rendezvous protocol would triple the latency to 6  $\mu$ s.

On the other hand, when messages are large, the cost of two extra control messages in the rendezvous protocol is easily outweighed by the performance gain of not having to perform memory-to-memory copies or extra buffer allocations when a send operation starts before a receive operation has been posted, as would be the case with the eager protocol.

### 2.2.3 Collective operations

Point-to-point communication can be used to build any complex communication pattern. However, different network topologies and parallel computer architectures often require different communication patterns to achieve optimal performance. MPI defines a set of *collective communication* patterns that attempt to cover most application communication requirements, while abstracting the underlying communication patterns used for their implementation. Key to a collective operation call is a communicator that defines the group or groups of participating processes.

Collective operations were primarily designed to take away the burden of implementing an optimized communication pattern using p2p communication from the application developer. An added benefit, is that they also enable performance portability, i.e., the same communication patterns expressed by collective operations can be expected to run reasonably well on a wide variety of different architectures and network topologies. Application developers are encouraged to use the predefined collective operations whenever possible. In a wider scope, one could argue that all point-to-point communications should be replaced by collective communications in order to produce well-structured parallel programs.

Gorlatchs [58] enumerates the following benefits that collective operations have in comparison with p2p operations:

**Simplicity** In MPI there is not one point-to-point communication primitive, but rather eight different kinds of send and two different kinds of receive. While allowing for very flexible programming, expressing complex programs in this style makes it very difficult to reason about the communication patterns if they are to be expressed in free-style.

**Programmability** Gorlatch argues that that programs structured as sequences of collective operations are amenable to several high-level program trans-

formations that can be used in systematic program design. For example, a Polynomial evaluation can be expressed as a series of (Bcast; Scan; Map; Reduce) operations.

**Expressiveness** A broad class of communication patterns found in HPC applications is covered by collective operations. The PLAPACK package for linear algebra, is entirely implemented without point-to-point communication operations.

**Performance** Collective operations typically offer better performance than what the average programmer can achieve on his own, because they are written by implementors more familiar with the parallel machine and its network parameters. It is becoming more common to see collective operations being implemented in hardware. Quadrics and Blue Gene/L interconnects provide NIC-based collective operations [8, 145] The performance gain by using hardware-based collectives can be orders of magnitude higher than using the most advanced software implementations based on point-to-point code. In addition, some reported data suggests that the performance of hardware-based operations is more consistent than the software-based approach [130]. A possible explanation lies in the fact that once the operation is delegated to the Network Interface Controller (NIC) it is no longer subject to CPU scheduling and will progress independently.

## 2.2.4 Global Reduction Operations

Disclaimer: the discussion in this section is based on the standard specification in [173], pages 173-174.

The Message Passing Interface (MPI) presents to the programmer several global reduction operations: a reduce that returns the result of a reduction to one member of a communicator (rooted operation), an all-reduce that returns this result to all members of a communicator and two different scan (parallel prefix) operations. Finally, a reduce-scatter operation is provided, combining the functionality of a reduce and of a scatter operation. The reduction operation applied on data belonging to each group member can either be one from a predefined list of operations, or user specified. The 12 predefined operators are: `MPI_MAX` (maximum), `MPI_MIN` (minimum), `MPI_SUM` (sum), `MPI_PROD` (product), `MPI_LAND` (logical and), `MPI_BAND` (bitwise and), `MPI_LOR` (logical or), `MPI_BOR` (bitwise or), `MPI_LXOR` (logical xor), `MPI_BXOR` (bitwise xor), `MPI_MAXLOC` (max value and location), `MPI_MINLOC` (min value and location).

This operation is always assumed to be associative. The predefined reduction operations are all commutative, while users are allowed to define associative, but non-commutative operations. The order of evaluation is not defined, and implementations

often take advantage of associativity or both associativity and commutativity to reorder the evaluation in order to optimize performance. This can result in different results for operations that are not strictly associative, such as operations on floating point data. The reduce operation is defined as follows:

```
MPI_REDUCE (sendbuf, recvbuf, count, datatype, op, root, comm)
IN  sendbuf  address of send buffer (choice)
OUT recvbuf  address of receive buffer (choice, significant only at root process)
IN  count    number of elements in send buffer (non-negative integer)
IN  datatype data type of elements of send buffer (handle)
IN  op       reduce operation (handle)
IN  root     rank of root process (integer)
IN  comm     communicator (handle)
```

Here, IN states that the procedure call may use the input value but does not update the argument from the perspective of the caller. OUT states that the call may update the argument but does not use its input value.

### 2.2.5 Optimization of collective operations

Early work on optimizing collective operation performance focused on developing optimized algorithms for specific architectures (hypercube, mesh, tree) with emphasis placed on minimizing link and node contention [16, 12, 8, 178, 23]. Using a combination of performance models and exhaustive testing, a best performing algorithm was selected for a range of problem sizes and communicator sizes.

Compiled communication was proposed to improve the performance of MPI collectives [200, 98]. In compiled communication, the compiler determines the communication requirements of a program. It does so by using the knowledge of the underlying network and architecture and the communication patterns present in the application.

With the advent of multicore architectures, hybrid programming paradigms combining message passing among nodes and multi-threading within, paved the way to higher throughput levels for HPC applications. Instrumental to this were hierarchical and scalable collective message passing algorithms [101, 185].

Automatic tuning (runtime or offline) of collective operations was recognized as just as essential [183, 42, 43]. In this context, experiments are performed to measure the performance of various algorithms for a given collective operations under different conditions (problem size, communicator size) and the best performing algorithm is selected for a given set of conditions. Vadhiyar et al [183] implement automatically tuned MPI operations. Using performance models coupled with gradient descent heuristics they reduce the search space for selecting the optimal algorithm. The Self-

Tuned Adaptive Routines for MPI collective operations (STAR-MPI) [43] analyzes both the system architecture and the application workload. As an application executes, STAR-MPI performs a runtime analysis based on the Automatic Empirical Optimization of Software (AEOS) technique [187] to select the best performing algorithm for that applications needs. This approach is known as *delayed finalization of MPI collective communication routines* [40]. However, STAR-MPI lacks flexibility in presence of rooted collective operations, where the root of a collective operation may change compared to the one that was used in the optimization process. A recent patent [44] on runtime optimization of parallel computer applications seeks to address that shortcoming.

However, most collective operations remain unoptimized for use cases where processes arrive at collective operation call sites in a desynchronized manner. In the following chapter, we discuss the implications of this in more detail.

## 2.3 Non-blocking collective operations

In September 2012, The MPI Forum released the MPI-3 standard [125], one of whose major novelties were non-blocking collectives. Non-blocking collective operations bring the combined benefit of avoiding synchronization and exploiting computation-communication overlap, together with the network optimized communication provided by blocking collective operations. Perhaps the first to strongly advocate for standardization of non-blocking collectives in MPI were Hoeffler et al in [80], building on their work in optimizing a conjugate gradient solver with non-blocking collectives [78]. Originally, the community consensus was that non-blocking collectives were not needed because the programmer could use a threaded MPI library and conduct the collective communication in a separate thread. This was, however, shown not be as simple and replete with drawbacks. Some operating systems designed for HPC (such as Catamount) did not even have support for spawning and managing threads.

Just like for non-blocking p2p operations, all calls to non-blocking collective operations are local and return immediately, regardless of the status of other processes. Other than allowing for overlap, non-blocking collective operations can be used to perform collective operations on overlapping communicators (i.e., those that share some of the processes), which would result with a deadlock if used with blocking collectives.

The performance benefits brought about by collective operations are best observed at scale. Many HPC applications extensively use collective operations, which as we will soon see, are in most cases implemented with static communication patterns. This makes them vulnerable to system noise, a major impediment to HPC application scaling.

Much research has recently been performed on the subject of performance improvement through usage of non-blocking collective operations. The authors in [57] present a modification of the GMRES algorithm where they overlap the dot-product global reduction communication with SpMV. They reported significant speedups compared to standard GMRES for strong scaling experiments and were mentioned in the "Report on the Workshop on Extreme-Scale Solvers: Transition to Future Architectures" by the U.S. Department of Energy as a "...new class of algorithms presenting significant opportunities for fundamental research". The algorithm is now included with the widely used PETSc library.

## 2.4 Visualization as a global reduction

Visualization is more than just producing images. A non-exhaustive list of common visualization operations includes rendering (isosurface/volume density), filtering, spectral analysis, feature extraction (extract isosurface), value computations (energy, satellite traces, isosurface volumes). All of the listed have one thing in common: global computation.

For example, the isosurface problem can be stated as follows. A *scalar volume dataset* consists of tuples  $(\mathbf{x}, \mathcal{F}(\mathbf{x}))$ , where  $\mathbf{x}$  is a 3D point and  $\mathcal{F}$  is a scalar function defined over 3D points. Given a scalar value  $q$  (the isovalue), isosurface extraction can be defined as computation and rendering of the isosurface  $C(q) = \mathbf{x} | \mathcal{F}(\mathbf{x}) = q$ . The computational process of isosurface extraction can be seen as consisting of two phases [26]. The first phase, *search phase* involves finding all *active* cells of the dataset intersected by the isosurface. The second phase, *generation phase*, uses an algorithm, such as *marching cubes* [110] to generate the isosurface out of active cells. The search phase is the more data intensive, as it searches the 3D dataset and produces 2D data. We can characterize the isosurface extraction as a reduction operation of light computational complexity and a small reduction factor (Table 2.2).

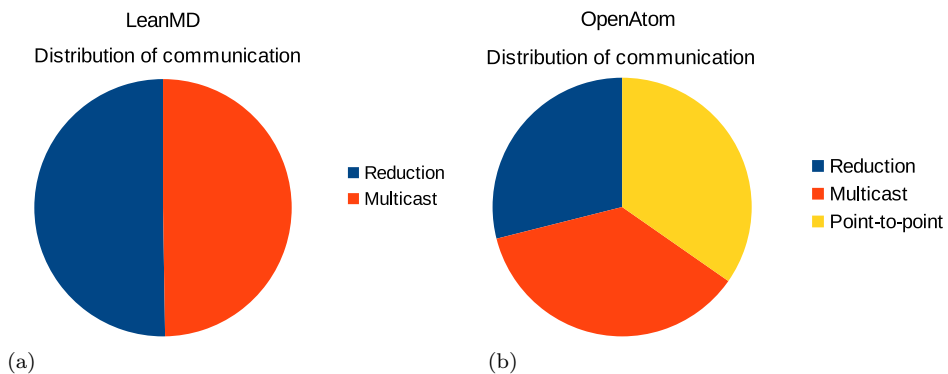
**Table 2.2:** Complexity of common visualization operations, viewed as global reduction.

Complexity of reduction	Light	Average	Heavy
Reduction factor			
Large	Max ( $ E $ )	Streamlines	Isosurface volume
Medium	Satellite trace	Phasespace	Rendering (isosurface/volume density)
Small	Isosurface extraction		

The essence of the rendering task is to calculate the effect of each data primitive

on each pixel, which can be seen as sorting primitives on the screen as first elucidated by Sutherland, Sproull and Schumacher [176]. In parallel systems, the sorting requires redistribution of data between processors. There are three principal distribution strategies employed in parallel rendering: sort-first, sort-middle and sort-last. The sort-last approach has high-scalability and good load balancing properties and is often used in distributed rendering systems [128]. The compositing stage is the key element of the sort-last parallel rendering pipeline. Each rendering unit generates a full-size partially complete frame and a global compositing operation is performed to produce the final image. The global compositing operation, also known as the *over operator* [15], is a per-pixel Z-buffer depth-visibility test (polygonal data) or a back-to-front alpha-blending (volume data) [36]. Thus, image compositing can also be classified as a global reduction operation, with complexity similar to isosurface extraction.

## 2.5 The reduction problem

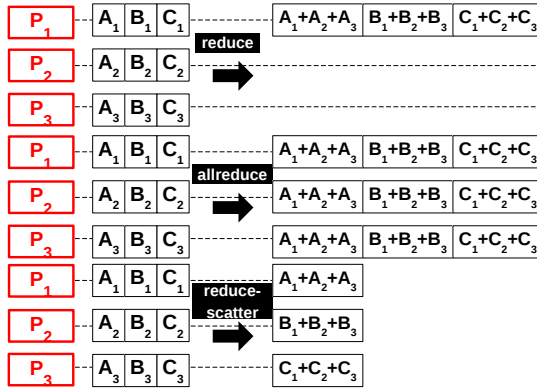


**Figure 2.3:** Breakdown of communication volume according to types of communication operations for two particle interaction applications. LeanMD [117] is a molecular dynamics benchmark, while OpenAtom [62] is a quantum chemistry application.

Reduction operations are among the most prevalent types of collective operations: a five year long profiling in production mode at the University of Stuttgart indicates that more than 40% of the execution time of MPI routines is spent in the collective communication routines `MPI_Allreduce` and `MPI_Reduce` [152]. Similar findings have been reported in [46] for several prominent HPC applications like CTH, SAGE and POP. Of the total communication volume of LeanMD (a molecular dynamics benchmark), 51.18% (Figure 2.3) can be attributed to reduction [118].

**Definition 2.5.1.** Assume a fully connected, single port homogeneous network of compute nodes (processes) with full duplex communication links. In such a network any process can at the same time send and receive one message, not necessarily from the same process. The homogeneity of the network ensures that the communication cost between any two pairs of processes is identical. It is further assumed that while a process is combining messages it cannot send nor receive other messages. We call this the **homogeneous simultaneous send/receive no-overlap model**.

This network model is often adopted in literature and considered reasonably appropriate for fully connected networks or modern interconnects like Myrinet and Infiniband [24, 22, 180, 164]. The no-overlap assumption in Definition 2.5.1 was introduced due to the absence of computation-communication overlap capability on the PRACE CURIE machine (discussed in more detail in Chapter 3).



**Figure 2.4:** The three principal types of global reduction operations: reduce, reduce-scatter and allreduce. In the illustration, A,B,C represent the three elements of the data input vector.

Of the three types of global reduction operations (Fig. 2.4), that are provided as part of MPI, in this dissertation we focus only on the former.

**Definition 2.5.2.** Consider a set of  $P$  processes numbered  $0 \dots P - 1$  distributed across a set of compute nodes, so that one and only one process is mapped onto each compute node. The compute nodes are spatially separated (no memory sharing)

and communicate with one another only by exchanging messages, as in the MPI programming model. Let each process  $i$  have a message  $m_i$ , where  $m_i$  is either a single value or an isotype vector of size  $m$ . We will refer to  $m$  as *problem size*. Consider now an associative commutative binary operator  $\star$ . We define the **P-way reduction problem** as the computation of the value  $M = m_0 \star m_1 \star \dots \star m_{p-1}$  that is made available at the root process 0 in the *shortest time*.

The requirement that the reduction problem be solved in shortest time is the key motivation behind the work in this dissertation. In principle, the collective operation can be said to have started when the first process in the communicator has arrived at the collective operation call site. The pattern of process arrivals at a collective operation call site is what we will call Process Arrival Time (PAT). When all the processes in the communicator arrive at the collective operation call site at roughly the same time, we say that such PATs are *balanced*. Otherwise, PATs are said to be *imbalanced*. The exact quantitative distinction between balanced and imbalanced PATs was first defined by Faraj et al [6]. There the authors consider PATs to be balanced if the range of the PAT is lower than the time required to send one message in the respective collective operation implementation. We will formally define this idea in the following chapter, where we discuss in more detail the implications it has on algorithm performance.

Surprisingly, the P-way reduction problem is one where multiple optimal solutions can be derived. We will briefly elaborate as to why this is the case. Chapter 3 discusses this issue in more detail.

First and foremost, there is no universal definition of collective operation runtime and thus different authors understand the *shortest time* in the definition differently. Secondly, by convention, the case where input data is a single atomic value, and the case where the data is a vector are treated as two distinct problems.

Thirdly, most work on the problem assumes that all processes should commence the reduction simultaneously. The authors go to great lengths to ensure this in their laboratory conditions. As of today, most library implementations of the global reduction operation can be said to be optimal only for the case when the processes begin the operation simultaneously.

However, in this dissertation we approach the problem from the standpoint that balanced PATs are *difficult to guarantee* in practice and have consequently endeavored to derive reduction algorithms that would be optimal for any pattern of PATs.

### 2.5.1 Design constraints on reduction algorithms

In some situations, the input data must be assumed *atomic*. This typically happens when the user supplies a custom defined combining operator  $\star$ . We will thus differentiate *atomic* and *non-atomic* P-way reduction problems. In this dissertation,

we present an optimal reduction algorithm for the atomic P-way reduction problem (Chapter 5) and a near optimal algorithm for the non-atomic P-way reduction problem (Chapter 6).

Another axis of reduction algorithm design (Table 2.3, orthogonal to atomicity of data, is fixity of communication patterns exhibited by algorithms. Static schedule algorithms are those algorithms that have an a priori determined communication graph that defines the order and direction in which processes receive and send data. On the other hand, dynamic algorithms will exhibit a different communication graph in each invocation. Static algorithms are typically preferred when the execution environment is in control and an optimal solution can be derived and implemented. Dynamic algorithms, however, might perform better in unpredictable environments, such as execution environments encumbered with load imbalance or system noise, outside the application programmer's control. Finally, reduction algorithms can be endowed with side information, such as the knowledge of PATs at their invocation. We will call algorithms that are supplied with such information *clairvoyant*, while those that are not agnostic.

**Table 2.3:** The three design axii in development of global reduction algorithms.

Design axis	Attribute	
Side information	Clairvoyant	Agnostic
Communication pattern	Static	Dynamic
Data constraint	Atomic	Non-atomic

The assumption of commutative and associative operator  $\star$  in Definition 2.5.2 is based on the fact that all 12 MPI combination operators, including sum, product, minimum, maximum, etc., are associative and commutative. Thus, all non-commutative combining operators are those defined by users. MPI allows the user to specify whether a given custom defined operator is commutative or not. If a custom combining operator is specified as non-commutative then the order of operands is fixed and is defined to be in ascending, process rank order, beginning with process zero. However, all combining operators are assumed associative. This property is essential for high performance, as it allows the evaluation to proceed in parallel. Problems can arise from using operators that are not "strictly" associative, such as most floating-point operations (as defined by the IEEE-754 standard). Parallel evaluation of such operators can lead to non-deterministic numerical error propagation. The MPI standard strongly encourages implementors to design algorithms such that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order.

## 2.6 State-of-the-art reduction algorithms

In this section, we present a selection of five algorithms that were included in the comparative study against the new algorithms proposed by this dissertation. As will become clear in the next section, these algorithms are representative of a much larger knowledge body of reduction algorithms.

It is assumed throughout this section that the root process resides at rank 0, and that the communicator size is a power-of-two. In the presented algorithm pseudocodes, *BLOCK* is a procedure of two parameters (iterator pointing to the beginning of a segment and segment size) that produces segments of input data.

### 2.6.1 Binomial Tree

This is the optimal reduction algorithm for atomic data, assuming that PATs are balanced. It is commonly found in implementations of many MPI libraries, where simple variations of the algorithm are used to also implement collective broadcast and gather operations. It belongs to a class of irregular tree topology algorithms, and is a subclass of  $k$ -nomial trees. For small messages, where latency dominates communication costs, this algorithm performs very well.  $k$ -nomial trees also form the basis of many hierarchical algorithms that are optimized for modern multicore shared memory cluster nodes [185]. We define the algorithm as Algorithm 1. Unlike the other algorithms that follow, in the execution of the binomial tree reduction algorithm, the number of outstanding processes is halved at each step. This is illustrated in Fig. 2.5.

---

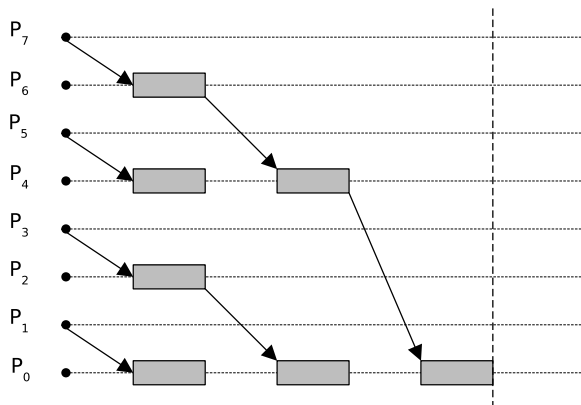
#### Algorithm 1: Binomial Reduction

---

**Require:**  $P = 2^x, x \in \mathbb{N} \wedge \text{root} = 0$

**Ensure:**  $R = m_0 \star m_1 \star \dots \star m_{P-1}$

- S1 Let  $x$  be the bit pattern of my rank.
  - S2 If the rightmost bit(lsb) of  $x$  is set to 1, send data message to the node with that bit set to 0
  - S3 If the rightmost bit(lsb) of  $x$  is set to 0, receive data message  $m_j$  from the node  $j$  with that bit set to 1 and combine.
  - S4 Shift the bit pattern  $x$  right by 1.
  - S5 If data was sent exit
  - S6 If the number of performed shifts  $n < \lceil \log_2 P \rceil$  goto 1, otherwise exit
-



**Figure 2.5:** Example of Binomial tree reduction algorithm execution for  $P = 8$  and root process at rank 0. Colored in gray is the time spent computing by each process.

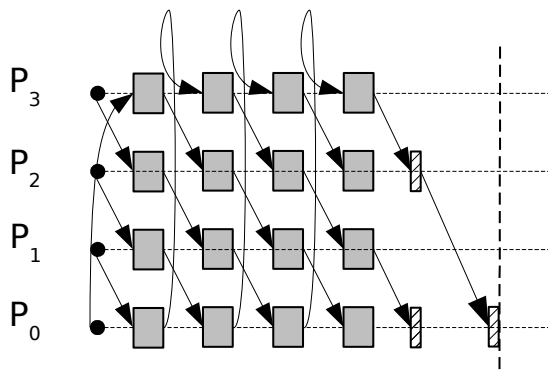
## 2.6.2 Parallel Ring

This is an algorithm best suited for large messages as discussed in [22] where the authors name it the bucket or cyclic algorithm. Our implementation is based on the algorithm explicated in that paper. We define Parallel Ring as Algorithm 2. The same algorithm forms the basis of the bandwidth optimal all-reduce algorithm discussed by [141]. The authors show, under the assumption that MPI processes with consecutive ranks are assigned to processors (cores) in each SMP node, the logical ring communication pattern of this algorithm is contention free (assuming full-duplex links on single ported nodes). This property makes the algorithm suitable for execution on fully subscribed SMP clusters. The execution of the algorithm for  $P = 4$  is illustrated in Fig. 2.6.

## 2.6.3 Butterfly

Many MPI implementations employ some version of the butterfly algorithm for reduction of large messages. The version of MVAPICH2 used for our experiments implemented the Rabenseifner's version of the butterfly algorithm [152, 178]. In the domain of computer graphics, this algorithm is also known as binary swap used for sort-last image compositing [112, 195]. For the purposes of this dissertation we have implemented the algorithm explicated in [22] as bidirectional exchange reduce-scatter followed by a call to library implemented `MPI.Gather`. Our implementation is written in iterative form, while that of [22] was written in recursive. We define Butterfly as Algorithm 3. This implementation necessitates that  $P$  is a power-of-2.

The algorithm operates in two phases. In the first phase a recursive vector-halving distance-doubling strategy is used wherein participating processes exchange



**Figure 2.6:** Example of Parallel Ring reduction algorithm execution for  $P = 4$ . Colored in gray is the time spent computing by each process. The algorithm proceeds in two phases. In both stages, the block or segment size is  $m/4$ . In the first stage the algorithm solves the reduce-scatter problem where each process ends up with  $1/4$  of the solution vector. Following that, the subsolutions are gathered to the root using a binomial tree communication pattern.

---

### Algorithm 2: Parallel Ring

---

**Require:** Input data is non-atomic

**Ensure:**

S1 Let  $prev = myRank - 1, size = \frac{m}{P}$

S2 If  $prev < 0$  then  $prev = P - 1$

S3  $next = myRank + 1$

S3 If  $next = P$  then  $next = 0$

S4  $index = next$

S5 For  $i = P - 2, \dots, 0$

S5.1 Send  $BLOCK(index, size)$  to  $prev$

S5.2  $index += 1$

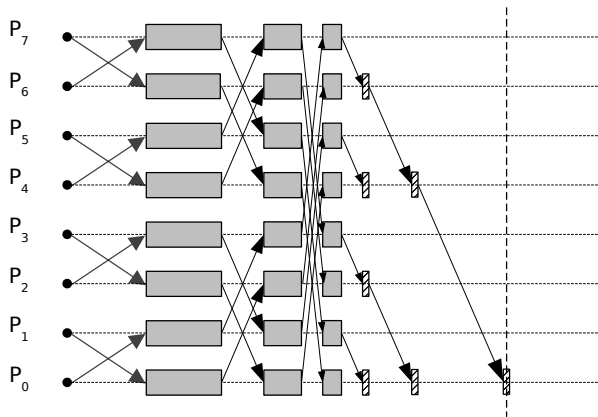
S5.3 If  $index = P$  then  $index = 0$

S5.4 Receive buffer from  $next$

S5.5  $BLOCK(index, size) += buffer$

G1  $COLLECTIVE\_GATHER(BLOCK(index, size))$  to ROOT

---



**Figure 2.7:** Example of Butterfly reduction algorithm execution for  $P = 8$  and root process at rank 0. Colored in gray is the time spent computing by each process. The algorithm proceeds in two phases. First it solves the reduce-scatter problem where each process ends up with  $1/8$  of the solution vector. In this stage the problem size is recursively halved so that segments of size  $m/2$ ,  $m/4$  and  $m/8$  are communicated, in that respective order. Following that, the subsolutions are gathered to the root using a binomial tree communication pattern wherein the message size is recursively doubled.

data pairwise (first with the neighbour one hop away, then with the one two hops away, four hops away, ...). In each step, only a half of the problem vector is exchanged and combined compared to that of the previous step. The gather operation in the final phase is implemented using a binomial schedule tree. The execution of the algorithm is illustrated in Fig. 2.7.

There exist optimizations of this algorithm for non power-of-2 process counts using a binary blocks [153, 178] scheme to reduce some of the load imbalance.

---

**Algorithm 3:** Butterfly
 

---

**Require:** Input data is non-atomic and  $P$  is a power-of-two

**Ensure:**

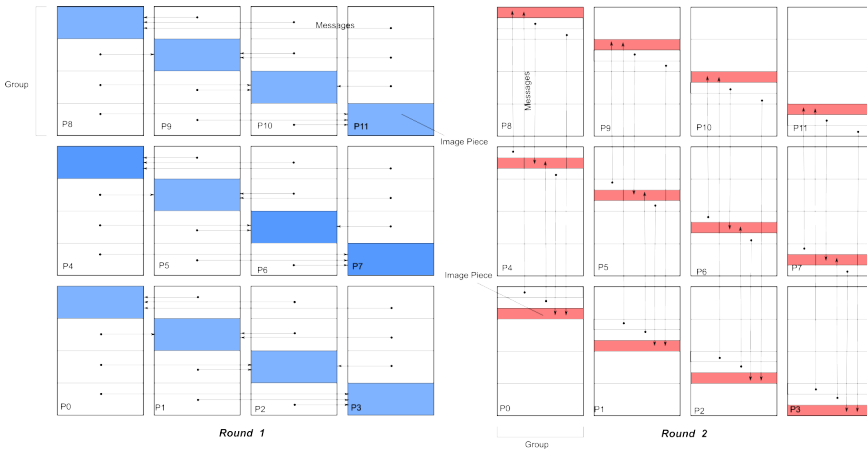
```

S1 Let size = m, k = 1, offRecv = 0
S2 If k < P
    S2.0 left += offRecv (vector halving)
    S2.1 size /= 2 (vector halving)
    S2.2 parity = myRank Bit-and k
        S2.2a If (parity == 0) then dir = 1 else dir = -1
    S2.3 k* = 2 (distance doubling)
    S2.4 peer = (myRank + (k/2) * dir + P) mod P
    S2.6 offSend = size * (not(parity))
    S2.7 offRecv = size * (parity)
    S2.8 SendRecv(peer, BLOCK(left + offSend, size), buffer, size)
    S2.9 BLOCK(offRecv, size) += buffer
LOOP Goto S2
G1 COLLECTIVE_GATHER(BLOCK(offRecv, size)) to ROOT
  
```

---

### 2.6.4 Radix-k

In the domain of image compositing a new algorithm has recently emerged, by the name of Radix-k, first described in [143] and later improved in [99]. Reduction operations in this domain are characterized by very large problem sizes ( $> 4\text{MiB}$ ) and non-commutative combination operators, disqualifying algorithms such as Parallel Ring that operate only on commutative operators. We define Radix-k as Algorithm 4. This algorithm operates by grouping  $P$  processes into  $r$  groups. These  $r$  groups form the radix vector  $k = [k_1, \dots, k_r]$  with the property that  $P = \prod_{i=1}^r k_i$ . The algorithm then proceeds in  $r$  rounds, where in each round  $i$  it performs  $k_i$  exchanges and reductions among  $\frac{P}{k_i}$  groups. In each round, the current slice is subdivided in  $k_i$  pieces, so that the size of the slice in round  $i$  is  $\frac{l}{\prod_{j=1}^i k_j}$ . Groups are formed in



**Figure 2.8:** Example of Radix-k execution for  $P=12$ , factored into 2 rounds  $R = \{4, 3\}$ . Reprinted from "A configurable algorithm for parallel image-compositing applications", by Peterka T., 2009., Proceedings of the Conference on High Performance Computing Networks, Storage and Analysis (SC '09). ACM, New York, NY, USA, Article 4. DOI=10.1145/1654059.1654064 [143]. Reprinted according to fair use.

the following way: in round 1, the  $k_1$  members of a group are nearest neighbours in rank order; in the next round, each member is now  $k_1$  apart, in the third round  $k_1 \cdot k_2$ , etc. An illustration of algorithm execution is given in Fig. 2.8. Radix-k has the potential to fine tune the amount of communication concurrency (multi-portness) to almost any given architecture by the appropriate selection of the k-values. When radix vector  $R = [2, 2, \dots, 2]$  then this algorithm becomes equivalent to Butterfly.

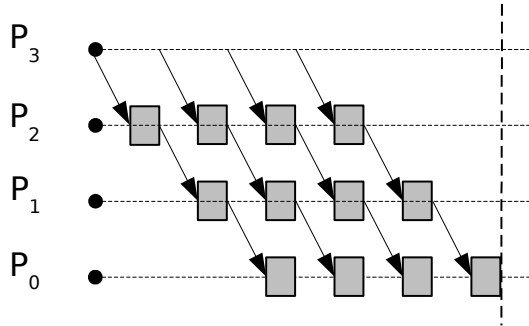
In the experimental evaluation of the algorithm's comparative performance we have determined the radix vectors empirically, by selecting for each problem size  $m$ , the radix vector that resulted in best performance. The empirically determined vector was identical for all problem sizes, and equalled  $R = \{4, 4, 8\}$ .

### 2.6.5 Linear Pipeline

This algorithm segments the input data  $N$  blocks and a linear pipeline is formed beginning with process rank  $P - 1$  and ending at the root (rank 0). At each step a process sends a block to its left neighbour and receives a block from its right neighbour. The received block is combined with the resident one before being sent to the left neighbour in the next step. If the underlying implementation allows independent progress, this algorithm can partially overlap the communication of subvectors with the time spent combining them. The large number of steps ( $P - 2$ ) for the first block to reach the root presents a scaling impediment on large systems, unless the messages are sufficiently large. The execution of the algorithm for  $P = 4$

**Algorithm 4:** Radix-k**Require:** Message  $m$  is non-atomic**Ensure:**S1 Let  $R$  be the radix vector  $R(r_0, r_1, \dots, r_k)$ S2 Let  $\text{start} = 0, \text{end} = 0$ S3 For  $i = 1$  to  $k$ S3.1  $\text{start} += \text{myRank} \frac{m}{\prod_{j=1}^{i-1} r_j}$ S3.2  $\text{end} = \text{start} + \text{prod}_{j=1}^i r_j$ S3.3  $g = \text{makeGroup}(r_i)$ S3.4  $\text{ReduceGroup}(\text{BLOCK}(\text{start}, \text{end}), g)$ G1  $\text{COLLECTIVE\_GATHER}(\text{BLOCK}(\text{start}, \text{end}))$  to ROOT

and  $N = 4$  is illustrated in Fig. 2.9.



**Figure 2.9:** Example of Linear Pipeline reduction algorithm execution for  $P = 4$ , number of segments  $N = 4$  and root process at rank 0. Colored in gray is the time spent computing by each process.

### 2.6.6 Process desynchronization after collective operation invocation

Some reduction algorithms, like Binomial Tree for example, can generate imbalanced process exit times even if when their respective arrival times are balanced.

---

**Algorithm 5:** Linear Pipeline

---

**Require:** Input data is non-atomic**Ensure:**S0 Let  $N$  be the number of segmentsS1 Let  $\text{next} = \text{myRank} - 1$ ,  $\text{size} = \frac{m}{N}$ S2  $\text{prev} = \text{myRank} + 1$ S3 For  $i = 0, \dots, N - 1$ S3.0  $\text{index} = i$ S3.1 If  $\text{next} \neq -1$  Send  $\text{BLOCK}(\text{index}, \text{size})$  to nextS3.2 If  $\text{prev} < P$  Receive buffer from nextS3.4  $\text{BLOCK}(\text{index}, \text{size}) += \text{buffer}$ 

---

In this algorithm, processes exit the collective operation as soon as they send their input data to their parent: after the first step of the algorithm, half of the processes will have exited the operation. If collective operations are invoked iteratively, without intermediate synchronization, this might have impact on the performance of subsequent collective operations.

### 2.6.7 Related work on reduction algorithms

Algorithm implementations of the collective reduction operation share many of the features seen in algorithm implementations of other collective operations, such as broadcast, gather, scatter, all-to-all, etc. In fact, some reduction algorithms implement the collective reduction as a combination of two other collective operations, typically reduce-scatter and gather. Common to the communication patterns of all algorithms is their mapping to a particular virtual topology, i.e. the communication can be described by a directed graph representing message propagation between processes. However, instead of classifying collective reduction algorithms according to their communication patterns, we have decided to classify them according to their input data preconditions.

One of the driving constraints in implementations of reduction operations is the atomicity of input data. An optimal reduction algorithm for atomic (non-segmentable) data was first presented by Karp in [97]. The authors assumed a fully connected homogeneous network and balanced process arrival times. The paper showed that the optimal algorithm for both operations is one that sends no redundant messages and has no unforced delays in sending or receiving messages. Such an algorithm thus sends messages as soon as it can and as often as it can. The authors in [111] present a greedy algorithm that preconstructs reduction schedules for homogeneous networks with communication-computation overlapping. Their algorithm constructs binomial tree reduction schedules if either the cost of

communication or the cost of computation is zero. When the two costs are equal, the algorithm constructs a Fibonacci tree reduction schedule.

When data is non-atomic, implementations are based either on pipelined tree reductions or composite algorithms based on reduce-scatter and gather operations. A prominent example of a composite algorithm is the butterfly algorithm elaborated in [152]. The original algorithm is limited to communicator sizes that are a power of two. Further improvements on this idea can be found in [153] with the additional focus on the non power-of-two number of processes. Another composite algorithm well suited for large input data is the bucket or parallel ring algorithm [141, 92, 22].

Pipeline tree algorithms are simple to implement and typically come in the form of linear tree pipelines or binary trees. Linear pipeline algorithms are known to perform well for large input data and small to moderate number of processes, but do not scale well with large number of processes. An improvement to the binary tree pipeline algorithm that exploits the full-duplex potential of modern networks was proposed in [164]. The authors report a near twofold speedup for their implementation compared with the pipelined binary tree reduction.

In the domain of parallel volume rendering, where input data is typically in the order of 4MiB-128MiB, several different reduction algorithms are prominent. A major performance impediment here is that the *over* operator in image compositing (Chapter 2, Section 2.4) is non-commutative. Direct Send is a simple algorithm that performs well on network interconnects with multi-port switches [136, 137]. In this algorithm each process is assigned a unique partition of the image to be composited (the problem size to be reduced). Then, each process sends each pixel fragment (or image partition) to the process responsible for compositing it. The total number of messages exchanged grows quadratically with  $P$ . This can be somewhat alleviated by the higher communication concurrency of the algorithm, but only if the network supports multi-port links. Otherwise, the algorithm will generate excessive network traffic, creating contention and subpar performance. Another algorithm, that conceptually combines direct send and butterfly algorithms, the 2-3 swap image compositing, has been shown to scale well on very large systems [199]. It relaxes the butterfly algorithm constraints such that processes can be grouped into pairs of two (butterfly) or sets of three. Using these groups of two or three, 2-3 swap can decompose any number of processes into groups, and in this way all processes can take part in image compositing throughout its entire time span. Another combination of direct send and binary swap, an algorithm called Radix- $k$  [143, 99] has been proposed as an improvement over the previous two algorithms. The algorithm first factors the communicator size  $P$  into a series of  $k$  numbers called the *radix vector*. The algorithm then proceeds in  $k$  rounds, where in each round  $i$  the communicator is partitioned into groups of size determined by the  $i$ -th value of the radix vector. Within each group, direct send is performed to composite the data. The next round recurses into processes with the same partition, until all

$k$  values are used and each process has a unique partition. The algorithm will be equal to direct send when it has one round with a  $k$  value equal to the size of the communicator. When the algorithm has  $\log_2 P$  rounds, each with group size of 2, it becomes equivalent to the butterfly algorithm [131].

A rather comprehensive treatise on the general problem addressing the performance of MPI collectives and their implementations can be found in [22]. Here the authors distinguish several different network topologies (linear, mesh, hypercubes, fully connected) and suggest an optimal solution for each collective operation over every considered topology. Another comprehensive work on the implementation of collectives in the MPICH library is that of Thakur et al. in [178].

Pjesivac et al. [148] present an in-depth analysis of collective operations performance using several frequently used models of parallel communication such as Hockney, LogP/LogGP, and PLog. Hoeﬂer and Moor [75] survey a large body of collective operations implementations and present models of their performance, energy and memory costs.

## 2.7 Summary

This chapter serves as a brief introduction to the message passing programming paradigm as embodied in the Message Passing Interface (MPI). The basic communication facilities such as p2p primitives and collective operations were described and their main attributes enumerated. The importance of collective operations was argued from the standpoint of code clarity and cross-platform performance.

The main problem of the thesis, the *reduction problem* was defined and discussed. Following that, a selection of five state-of-the-art reduction algorithms was presented: Binomial Tree, Butterfly, Parallel Ring, Linear Pipeline and Radix-k. Each of the algorithms was defined in pseudocode form. Their comparative performance will be presented in the following chapter, once an appropriate runtime cost model has been defined.

It is important to emphasize that this work does not attempt to provide a comprehensive survey of reduction algorithms known in literature. In particular, a high-performance tree pipeline algorithm, 2-Tree, was not included. While the experimental results of this algorithm might be interesting, it is an equi-segmenting algorithm like Linear Pipeline and Clairvoyant (Chapter 5) and in absence of imbalance its expected performance is strictly worse than that of Clairvoyant (Table 3.3). On the other hand, algorithms Butterfly and Radix-k operate with heterogeneous segment sizes and can in theory outperform algorithm Clairvoyant. This is discussed in more detail in Chapter 6.

The relevant results of this dissertation will be reflected on the comparative

performance of the new reduction algorithms introduced in Chapter 5 and Chapter 6 to the reduction algorithms defined in Section 2.6. If they are to be of practical benefit, their performance will have to surpass that of the state-of-the-art for imbalanced PATs and at least match them for balanced PATs.



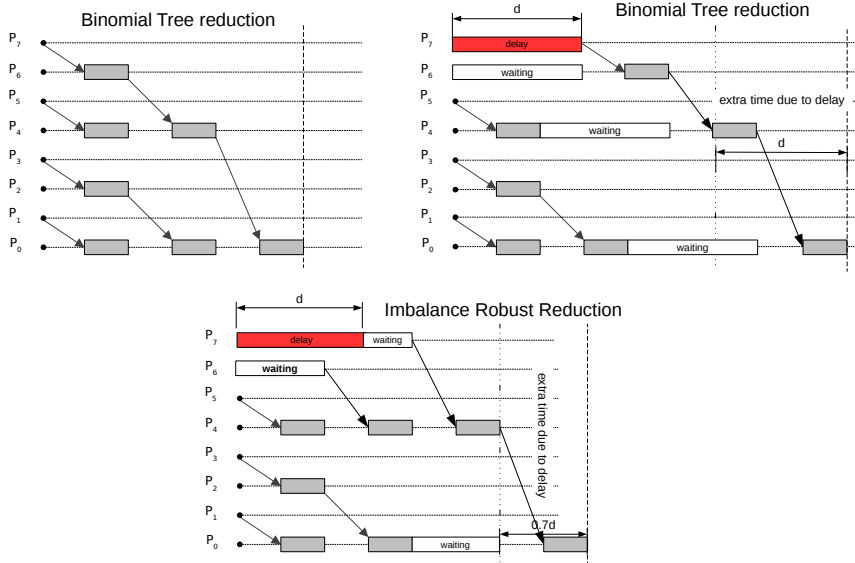
## Chapter 3

# Process arrival time, imbalance and collective operation runtime

---

Collective operation performance is critical to High Performance Computing (HPC) application performance. Therefore, a great deal of research has been conducted to design optimized algorithms and produce efficient implementations for various collective communication operations. However, state-of-the-art reduction algorithms are optimized only for the case where processes call (arrive at) the collective operation simultaneously. Such Process Arrival Times (PATs), where all the processes arrive at the call site at roughly the same time, are said to be *balanced*. Yet, balanced PATs are extremely rare and generally only occur immediately after synchronization routines. That PATs can have an impact on the performance of collective operations has been largely overlooked by the research community. The reason for this is the perception that imbalanced PATs and optimization of collective operations are disjunct problems that can be addressed independently. If we could only ensure that PATs are balanced before collective operation invocations, then all the work in optimizing the collective operations for balanced PATs would be justified. Yet, as many researchers have found, balanced PATs are difficult to ensure. This is due to two principal causes: load imbalance and system noise.

As we will show in this dissertation, it is possible to design collective operations that both mitigate the effects of imbalanced PATs and achieve optimal performance. Such collectives can perform work despite some of the processes in the communicator being belated. An illustration of one imbalance mitigation strategy is illustrated in Fig. 3.1. Compared to the binomial tree reduction algorithm, where the reduction schedule is a priori determined, the imbalance robust algorithm ensures that processes that are ready for communication do so out-of-order prescribed by the binomial tree schedule. As a result, a portion of the time process rank 7 spent waiting, was absorbed compared to no absorption of the binomial tree reduction strategy.



**Figure 3.1:** Impact of a single delayed process on the runtime of the binomial tree reduction algorithm. The imbalance robust algorithm uses a greedy strategy, similar to that of Algorithm 8 defined in Chapter 5. In this strategy, no process waits if there is another ready process. As a result, it manages to absorb nearly one third of the imbalance time  $d$ .

### 3.1 Load imbalance

Load imbalance is the bane of parallel computing. Amdahl’s law [9] stipulates that the maximum achievable speedup of some code in a parallel system is limited by the serial fraction in the code. The serial fraction of the code (or the problem) is the portion not amenable to parallelization. The presence of load imbalance in the execution of a parallel algorithm reduces the fraction of the problem that can be executed in parallel, resulting with smaller than desired/expected speedups.

Load imbalances can be roughly classified into three types:

**Type 1** imbalances in the phase preceding the collective operation

**Type 2** imbalance in the communication time of each step of the operation

**Type 3** imbalance in the completion time of the combination operator

The distributed image rendering problem, introduced in Chapter 2, is a good example where all three types of imbalances occur. In sort-last rendering, each

process first renders a local image (typically, in the multi megapixel range). In this stage imbalances occur due to uneven distribution of primitives in the volume, as seen from the view pane. In the parallel image rendering of a set of images from the Materials and Geometry Format database of the Lawrence Berkeley Laboratory, the mean number of floating point operations per pixel was measured as  $\mu = 10^6$ , while the variance  $\sigma^2 = 10^{12}$  [63]. The relative magnitude (to the mean) of the variance was shown to be persistent across different ray tracing algorithms, indicating that effective load balancing is a serious challenge in this domain.

After a local image has been rendered, each process then collaboratively composites the sub-images into the final render. Because image compositing is a global reduction operation, imbalances in local-image rendering times result in Type 1 imbalances. Due to the large size of sub-images, compression schemes are necessary to reduce network bandwidth load. This, however, results in Type 2 imbalances across the phases of the collective reduction operation. In the last, image compositing stage, each pixel of the final image is computed in a back-to-front alpha-blending fashion using the over operator [15]. The computational complexity of this stage is a linear function of the effective image size, measured in the number of non-black pixels. Because effective image size is inherently variable, Type 3 imbalances will regularly occur. Type 3 imbalances will also typically occur in heterogeneous systems.

Load imbalance has been determined as one of the major causes of performance deficit in petascale applications. In the domain of weather forecasting, one of the several prospective exascale applications, Xue et al [194] reported that sub-domains assigned to some processors may incur 20%-30% additional computational time due to non-uniformity of atmospheric processes distributed across the computational domain, such as active thunderstorms. Load balancing in atmospheric prediction models is challenging due to both the complexity of associated algorithms and the communication overhead associated with moving large amounts of data between processes. Due to this, most applications in the domain would forego load balancing. However, to scale applications to petascale and exascale levels, some form of load balancing is essential.

### 3.1.1 Load balancing strategies

Among the various forms of load balancing, the one that requires no changes to legacy code while still offering competitive performance is the most desirable. One such approach is *process virtualization*, exemplified by Charm++ [96]. Charm++ is a parallel programming system that builds on the idea of *over-decomposition*, where the programmer decomposes the application's data into a set of  $N$  objects that will execute on  $P$  processors, where  $N \gg P$ . The Charm++ runtime system handles the mapping of these objects onto the physical processors, including their migration

during application's execution. Overdecomposition allows for effective decoupling of the program data from the physical machine upon which it is to be executed.

Adaptive Message Passing Interface (AMPI) is a Message Passing Interface (MPI) implementation built on top of the Charm++ system [85, 86]. "In AMPI, each MPI rank is implemented as a user-level thread embedded in a Charm++ virtual processor. This approach ensures that the benefits of Charm++ are available to the broad class of applications written using MPI. Thus, an MPI program designed to be run on  $K$  processors is typically executed by AMPI with  $K$  virtual processors on  $P$  physical processors, where  $K \geq P$ " [162].

While the problem of balancing  $N$  communicating threads among  $M$  processors can be modeled and solved as a Mixed Integer Quadratic Programming (MIQP) problem [162], doing so is only feasible for small values of  $N$  and  $K$ . In almost all practical cases, a heuristic solver is required instead. Load balancing problems are closely related to scheduling problems, where dynamic algorithms are common [157]. Charm++ uses automatic instrumentation to perform load balancing on behalf of AMPI programs, by storing and analyzing at runtime the information on computational loads and communication patterns. This approach requires that users introduce in their code calls to a collective AMPI function `MPI_Migrate` that invokes the underlying load balancer. The authors in [162] evaluated different load balancing approaches for a BRAMS forecasting model using a grid of 512x512 horizontal points and 40 vertical levels, executed for 2400 time steps, where each time step required 6 seconds. They report a maximum attained speedup of 1.59 using a Hilbert curve based load balancer. However, merely using over decomposition without load balancing resulted in a speedup of 1.34.

An even less invasive approach in dealing with imbalance, is to design collective operations that are either inherently robust to imbalance or that can perform dynamic ad-hoc load balancing at each invocation. In this dissertation, we present two novel reduction algorithms that are robust to Type 1 imbalances (Chapter 5 and Chapter 6). Both algorithms are contingent on pre-knowledge of the PATs at the reduction operation call site to generate reduction schedule graphs, where early arrived process begin communication as soon as possible. However, the third algorithm herein presented (Local Redirect in Chapter 5) is robust to all three types of imbalances. A distinguishing feature of imbalance robust collective operations is that they require no changes to legacy code and if properly designed should be able to fully replace their non-robust counterparts, even in absence of load imbalances.

## 3.2 System noise

System noise, a result of operating system level interrupts and various other architectural overheads is another major source of performance degradations on large

---

systems running parallel applications. Petrini et al. [38] succeeded in putting this issue into spotlight by showing how interrupts from the system kernel and various daemons can lead to very big slowdowns of bulk synchronous (iterative compute-communicate phases) applications when run on a large number of processors. Their results have spurred much research on the topic: Agarwal et al. [5] performed a theoretical analysis of the potential impact of three distributions of system noise (exponential, heavy-tailed and Bernoulli) might have on the performance of MPI collectives. Their results show that most systems are expected to scale well under exponentially distributed noise while the heavy-tailed and Bernoulli distributed noise are expected to incur significant performance penalties. It has been shown that applications written in Bulk Synchronous Parallel (BSP) style are particularly sensitive to system noise [56, 38], as the process with the longest completion time dictates the runtime of each stage. This effect is particularly pronounced on large scale systems.

System noise can to a large degree be curtailed by fine tuning operating system kernels and system architecture parameters [38, 46]. Another strategy proved viable is to use non-blocking collectives to mitigate the impact of system noise if the amount of overlap an application is actually able to achieve between computation and collective propagation is sufficiently large [47]. The authors implemented a non-blocking `MPI_Allreduce` and showed that with sufficient overlap between application computation and collective communication, the effects of system noise can be almost entirely dampened. However, non-blocking collectives come with a disadvantage: computation needs to be independent of communication, or the core algorithms need to be redesigned accordingly. Consequently, most legacy code needs to be rewritten to use non-blocking collectives.

Hoefler et al. [84] introduced an OS noise measurement and simulation framework and analyzed the impact such noise might have on large-scale applications. The authors performed simulated runs with up to 1 million processes and have shown that the scale at which system noise becomes a bottleneck is system specific and primarily dependent on its distribution. However, there is a clear trend of increased noise amplification with increasing system size. This research also showed the effect system noise can have on various collective operations and identified that `allreduce` is a particularly sensitive one, due to its tendency to amplify system noise. That `allreduce` is a sensitive collective operation was reported earlier by [46] where the authors have implemented a kernel injection noise generation system. They have shown a slowdown of 2000% for a loaded schedule noise signature (high duration, low frequency) for the Parallel Ocean Program (POP) due to noise amplification. This particular application spends the majority of its runtime in the `allreduce` MPI collective. The authors have also conjectured and confirmed that the more hardware imbalanced a given system is (higher computation to communication ratio) the less susceptible it might be to OS noise.

### 3.3 Process Arrival Time and collective operation runtime

The previous two sections highlighted the gravity of process desynchronization on parallel application efficiency. In this section we proceed to formally define the notion of Process Arrival Time (PAT), which is inextricably linked to Type 1 load imbalances, defined in Section 3.1.

**Definition 3.3.1.** Let  $a_i$  denote the time when process  $i$  arrives at the collective operation call site, or in other words the time at which process  $i$  commences the collective operation. Let  $P$  denote the number of processes in the communicator. We define Process Arrival Time (PAT) as the vector  $a = (a_0, a_1, \dots, a_{P-1})$ .

The average arrival time is defined as  $\bar{a} = \frac{a_0 + a_1 + \dots + a_P}{P}$ . When all processes arrive at the same time, we say that the PATs are *balanced*. Otherwise, PATs are *imbalanced* (desynchronized).

**Definition 3.3.2.** Let  $e_i$  denote the time when process  $i$  exits the collective operation. We define the Process Exit Time (PET) as the vector  $e = (e_0, \dots, e_{P-1})$ .

Fig. 3.2 illustrates the concepts of process arrival ( $a$ ) and exit time ( $e$ ) patterns, our notion of collective operation execution time ( $t_A$ ), absolute imbalance  $I(a)$  and the average arrival time  $\bar{a}$ .

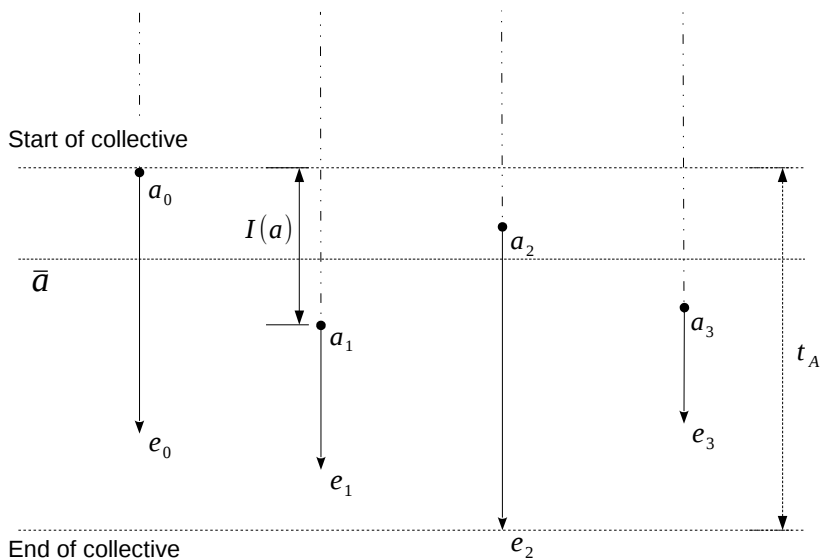
As heralded by Faraj et al. [6], we will use the term *imbalance* in PATs to signify the level of process desynchronization in collective operation calls. Let  $\delta_i$  be the time difference between process  $i$ 's arrival time  $a_i$  and the average arrival time  $\bar{a}$ ,  $\delta_i = |a_i - \bar{a}|$ . One way to quantify the imbalance in process arrival times is by the *average imbalance* time  $\bar{\delta} = \frac{\delta_0 + \delta_1 + \dots + \delta_P}{P}$ . Another approach, adopted in this work, is to consider the absolute difference (range) in arrival times, the *absolute imbalance* defined as follows.

**Definition 3.3.3.** Let  $a$  be the vector  $(a_0, a_1, \dots, a_{P-1})$ . We define  $I(a)$ , the **absolute imbalance** of the PAT  $a$  to be  $\max_{0 \leq i < P} a_i - \min_{0 \leq i < P} a_i$ , i.e. the time difference between the latest process to arrive and the earliest process to arrive.

The absolute imbalance defines the range of arrival times, but tells us little about their distribution. We introduce the idea of *slack* to formalize the important property of PAT patterns that will determine the absorption potential of imbalance robust algorithms:

**Definition 3.3.4.** Let  $a$  be a vector  $(a_0, a_1, \dots, a_{P-1})$ . Without loss of generality, let  $\min(a) = 0$ , i.e.  $I(a) = \max(a)$ . We define the **slack** of  $a$  to be

$$1 - \frac{\bar{a}}{I(a)}$$



**Figure 3.2:** An illustration of process arrival time (represented with vector  $a$ ), average arrival time  $\bar{a}$ , collective operation runtime  $t_A$  and absolute imbalance  $I(a)$

By definition, for balanced PATs (that we will henceforth denote with the Greek letter  $\pi$ ), the slack is zero.

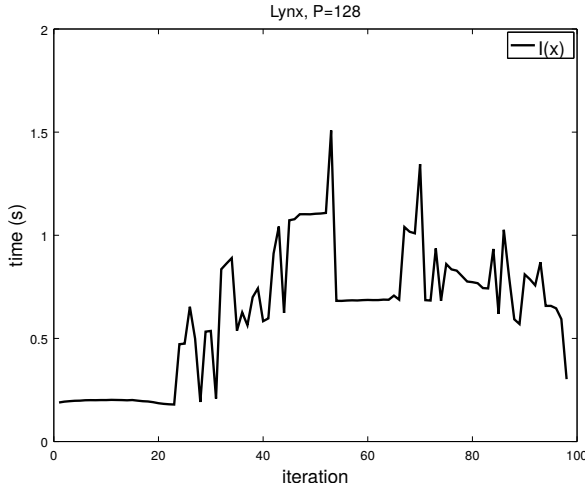
Thus defined, the slack is in the range  $[0, 1)$ , being minimal when PATs are balanced and maximal for PATs where a single process is delayed.

### 3.3.1 Collective operation runtime

There is no single definition of what constitutes collective operation runtime, nor how to measure it. Not all research papers define explicitly their understanding of collective operation runtime and often leave it implicit in their adopted runtime measure (estimation). This leads to measures and reported results that are sometimes incomparable. In all cases, estimation of collective operation runtime is performed by measuring the elapsed time between two distinguished events: the start and the end of the collective operation. These events either reside on the same process or on different processes. The former leads to local, while the latter to global measures.

Definition 3.3.5 presents the understanding of collective operation runtime adopted in this dissertation. This is a global measure.

**Definition 3.3.5.** Let  $\mathcal{A}$  be an algorithm implementing a  $P$ -way collective operation. A collective operation is said to be  $P$ -way if the number of participating processes or communicator size, is  $P$ . Let  $t_{\mathcal{A}}^i = e_i - \min_{0 \leq j < P} a_j$  be the runtime of



**Figure 3.3:** Absolute imbalance in image rendering times as measured on the Lynx cluster across 100 iterations of the Helsim plasma physics simulation.  $P=128$

process  $i$ . We define the runtime of  $\mathcal{A}$  denoted as  $t_{\mathcal{A}}$  to be:

$$t_{\mathcal{A}} = \max_{0 \leq i < P} t_{\mathcal{A}}^i(j) = \max(e) - \min(a)$$

In other words, we define the runtime of a collective operation as the time difference between the last process to exit and the first process to arrive at the collective operation. The same definition was adopted by the authors of MPIBlib [105]. This measure is also known as *time-to-solution*.

On the other hand, some of the research papers on the subject of imbalanced PATs [6, 140] adopt a local measure instead: mean or average process runtime.

**Definition 3.3.6.** Let  $\mathcal{A}$  be an algorithm implementing a  $P$ -way collective operation. Then the mean runtime of  $\mathcal{A}$  is defined as:

$$\bar{f}_{\mathcal{A}} = \frac{\sum_{i=0}^{P-1} (e_i - a_i)}{P}$$

Mean runtime is an appropriate measure for optimizing the energy consumption of a collective operation, an important consideration in large scale systems [75].

The principal motivation behind the limitation to solely commutative operators in Definition 2.5.2 stems from the observation that in presence of desynchronized or imbalanced process arrival times, the  $P$ -way reduction problem will always have an equal or smaller solution when the combining operator  $\star$  is commutative.

Let process  $i$  and process  $j$  be adjacent in the PAT vector  $\psi = a_0, \dots, a_i, a_j, \dots, a_{P-1}$ .

Then, if  $\star$  is non-commutative and  $|i - j| > 1$  the operator application  $\star(i, j)$  will have to be delayed leading to strictly equal or greater total evaluation time.

### 3.3.2 Implications on relative algorithm performance

Local and global runtime measures are not directly comparable. Importantly, they differ in the manner that they order algorithm runtimes. Let  $\mathcal{A}$  and  $\mathcal{B}$  be two *atomic P-way reduction* algorithms. It is not difficult to show that:

$$\bar{f}_{\mathcal{A}} \leq \bar{f}_{\mathcal{B}} \not\Rightarrow t_{\mathcal{A}} \leq t_{\mathcal{B}}.$$

The measure  $\bar{f}$  has a peculiar property: when used to compare algorithm performance, it will be biased towards larger values in presence of imbalanced PATs. We consider this an important distinction compared to the measure  $t_{\mathcal{A}}$  used in this paper and will further elaborate.

Let  $(s_0, s_1, \dots, s_{P-1})$  be  $P$  processes sorted so that their arrival times follow an ascending order  $(a'_0, a'_1, \dots, a'_{P-1})$ , where  $a'_i$  is the arrival time of  $s_i$  and  $a'_{P-1}$  the arrival time of the root process. Let  $d$  be the time to communicate and combine the input data of two processes. In addition, let  $a_i - a_{i-1} > d, i \geq 1$ . Let  $\mathcal{A}$  be the following algorithm: process  $s_0$  sends its value to process  $s_1$  and exits the operation. Process  $s_i, 2 \leq i \leq P - 2$  after receiving the message from  $s_{i-1}$ , sends to  $s_{i+1}$  and exits the operation. In the execution of the algorithm, process  $s_i$  has to wait until process  $s_{i+1}$  becomes ready. Thus  $e'_i = a'_{i+1} + d$  and the total time is equal to  $e'_0 - a'_0 + e'_1 - a'_1 + \dots + e'_{P-1} + a'_{P-1} = I(x) + Pd$ . This implies that  $\bar{f}_{\mathcal{A}} = \frac{I(x)}{P} + d$ . Due to the assumption that  $a_i - a_{i-1} > d, i \geq 1$  it follows that  $t_{\mathcal{A}} = I(x) + d$ .

Let  $\mathcal{B}$  be an algorithm identical to  $\mathcal{A}$  with the handicap of a collective barrier operation at its start: i.e. processes cannot begin combining their messages until all have arrived. Then the algorithm requires  $P - 1$  steps for the final message to be combined at the root process, leading to  $t_{\mathcal{B}} = I(x) + (P - 1)d$ . The total runtime is:  $I(x) + d + I(x) + 2d + \dots + I(x) + Pd$  which implies that  $\bar{f}_{\mathcal{B}} = I(x) + \frac{P-1}{2}d$ . Now if we look at how the two algorithms scale in regards to increasing absolute imbalance, we will observe that the two measures produce largely different relative performance:

$$\lim_{I(x) \rightarrow \infty} \frac{\bar{f}_{\mathcal{B}}}{\bar{f}_{\mathcal{A}}} = \frac{I(x) + \frac{P-1}{2}d}{d + \frac{I(x)}{P}} = P$$

compared to that derived by

$$\lim_{I(x) \rightarrow \infty} \frac{t_{\mathcal{B}}}{t_{\mathcal{A}}} = \frac{I(x) + Pd}{I(x) + d} = 1$$

In the presence of imbalanced PATs, reporting  $\bar{f}_{\mathcal{A}}$  might overestimate the

speedup of imbalance-tolerant algorithms by up to  $P$  compared to  $t_{\mathcal{A}}$  if the absolute imbalance dominates algorithm runtime.

The choice of runtime metrics is not clear-cut. Local time measures, such as  $\bar{f}_{\mathcal{A}}$ , are very much relevant when we are more concerned with the power consumption rather than the time-to-solution of collective operations. This might be relevant on very large systems, or on small battery-powered devices. Hoeffler and Moor [75] derive runtime, energy and memory consumption models for a wide range of collective operation algorithms, together with comparative analysis within each group. An interesting result of their study is that runtime-optimal algorithms universally have non-optimal dynamic energy consumption. This would provide an interesting optimization problem to find the best trade off between performance and energy consumption, when designing collective operations.

### 3.3.3 Prior studies on imbalanced process arrival times

While it would be reasonable to expect that imbalanced PATs would be caused by load imbalance prior to the collective operation call site, recent research has shown that even with perfect load balancing, the presence of system noise can still generate imbalanced PATs at collective operation call sites.

Faraj et al [6] studied the PAT patterns of a set of MPI benchmarks on two Commercial off-the-shelf (COTS) clusters: a high-end (at the time) Alphaserver and a low-end Beowulf cluster with a Gigabit Ethernet interconnect. They found that PATs at MPI collective operation call sites are usually imbalanced. Surprisingly, even in a contrived microbenchmark (Listing 3.1) where perfect load balancing was simulated, the PATs remained imbalanced.

This microbenchmark has every process performing an equal amount of work on private arrays. Parsons et Pai [139] attempted to investigate the causes of this imbalance by sampling performance counters using Performance Application Programming Interface (PAPI) [134]. The counters included such events as hardware interrupts, cache misses, stall cycles and the number of kernel instructions. They compared the sampled counter values to measured PATs. However, they failed to establish a correlation with any one event. They conclude “Instead, the imbalance seems to be caused by multiple system components, with no direct correlation to any one.”

**Listing 3.1:** Balanced workload micro-benchmark as defined by Faraj [6]

```
for (i=0;i<test_iters;i++) {
  for (j=0;j<xtime;j++) {
    for (k=1;k<1000;k++) {
      a[k]=b[k+1]-a[k-1]*2;
    }
  }
}
```

```

arrival[i]= MPI.Wtime();
//Evaluate collective operation
MPI_Collective();
exit[i]=MPI.Wtime();
}

```

An important result of their investigation is that MPI collective operation algorithms are sensitive to PAT imbalance, with algorithms that perform better with balanced PATs exhibiting a larger performance impact with imbalanced PATs. They showed that collective operations vary in the degree that they are susceptible to imbalanced PATs. For example, they found the broadcast operation to be less sensitive to imbalance than the all-to-all operation. In the broadcast operation, processes can complete the collective call before all processes in the communicator have arrived at the collective operation call site, unlike the all-to-all operation.

They conclude: “These findings indicate that for an MPI collective routine to be efficient in practice, it must be able to achieve high performance with different (balanced and imbalanced) process arrival patterns. Hence, MPI library implementers must take process arrival patterns into consideration in developing and optimizing MPI collective routines.”

### 3.3.4 Absorption time and synchronization delay

Examination of related literature reveals another global runtime measure used in the context of imbalanced PATs [35, 113], defined as *synchronization delay*:  $\delta = \max_{0 \leq i < P} (e_i) - \max_{0 \leq i < P} (a_i)$ , i.e. the time difference between the last process to exit and the last process to arrive.

For balanced PATs and some algorithm  $\mathcal{A}$ ,  $t_{\mathcal{A}} = \delta_{\mathcal{A}}$ . For imbalanced PATs, synchronization delay will ignore the segment of  $t_{\mathcal{A}}$  incurred by the absolute imbalance, leading to larger relative differences in the runtimes of different algorithms. The ordering of the algorithm runtimes remains the same, so these two measures are essentially comparable.

**Definition 3.3.7.** Let  $\mathcal{A}$  be an algorithm for  $P$ -way reduction with problem size  $m$ . Let  $\psi$  be some PAT vector. We will denote by  $t_{\mathcal{A}}(\psi, P)$  the runtime of algorithm  $\mathcal{A}$  for the  $P$ -way reduction problem and the PAT  $\psi$ . Whenever it does not create confusion, we will denote this time more succinctly as  $t_{\mathcal{A}}(\psi)$ . Let  $\pi$  be the PAT  $(0, 0, \dots, 0)$ . Then **absorption time** is defined as:

$$A(\psi, \mathcal{A}) = t_{\mathcal{A}}(\pi) - t_{\mathcal{A}}(\psi) + I(\psi)$$

It will be useful to normalize both the absolute imbalance and absorption time in relation to algorithm runtime for balanced PATs. This will allow us to characterize

the absorption capacity of reduction algorithms in terms of their nominal runtime (for balanced PATs).

**Definition 3.3.8.** We define the normalized absolute imbalance to be

$$I_N(\psi) = \frac{I(\psi)}{t_{\mathcal{A}}(\pi)}$$

**Definition 3.3.9.** We define the normalized absorption time to be

$$A_N(\psi, \mathcal{A}) = \frac{A(\psi, \mathcal{A})}{t_{\mathcal{A}}(\pi)}$$

If absorption time is equal to the absolute imbalance, then the algorithm will not exhibit any slowdown due to imbalance in process arrival times. However, there exists an upper bound on the absorption time.

**Proposition 3.3.10.** *Let  $\mathcal{A}$  be a  $P$ -way reduction algorithm. Then,*

$$A(\psi, \mathcal{A}) \leq t_{\mathcal{A}}(\pi) - t_{\mathcal{O}}(\pi, 2),$$

where  $t_{\mathcal{O}}(\pi, 2)$  is the optimal time to solve the 2-way reduction problem.

*Proof.* The largest absorption time will be attained when by the time the last process (let that be process  $i$ ) has become ready only the message  $m_i$  remains to be combined to derive the final result. Let us select the shortest absolute imbalance  $I(\psi)$  for which the assumption holds, i.e.  $I(\psi) = t_{\mathcal{A}}(\pi, P-1)$ . Then,  $t_{\mathcal{A}}(\psi) = I(\psi) + t_{\mathcal{O}}(\pi, 2)$ . From this it follows that

$$\begin{aligned} A(\psi, \mathcal{A}) &= t_{\mathcal{A}}(\pi) - t_{\mathcal{A}}(\psi) + I(\psi) \\ &= t_{\mathcal{A}}(\pi) - I(\psi) - t_{\mathcal{O}}(\pi, 2) + I(\psi) \\ &= t_{\mathcal{A}}(\pi) - t_{\mathcal{O}}(\pi, 2). \end{aligned}$$

□

## 3.4 Time complexity models

Modelling the runtime of HPC applications or parallel algorithms can be useful for estimating their performance both to guide their design and estimate their performance on various parallel system architectures. In developing such a model, a researcher will typically construct a dependency graph of communications and computations beginning with the input data and leading to the output data. This *algorithm model* can then be matched to a *machine model* in order to estimate the runtime of the algorithm on a particular architecture [79].

HPC application developers are typically guided by simple performance models on which they base their algorithmic decisions. For example, most would assume that transmitting a message of size  $m$  over the network has a cost linear in  $m$  and that gathering a small message in a communicator of size  $P$  would have cost logarithmic in  $P$ .

However, algorithm complexity analysis for parallel execution environments where  $P$  and  $m$  can take a wide range of values, is a non-trivial task. Attaining high degrees of accuracy can only be expected for a limited set of cases and even then with considerable effort. There are many parameters that are typically outside the algorithm designer's influence, such as process-to-node mappings and contention caused by traffic from concurrently launched jobs. For practical reasons many of these parameters have to be abstracted away using an abstraction known as the parallel machine model. The degree of abstraction present in a model ultimately limits its accuracy. We first examine a hierarchy of viable models as outlined in [79]:

**Asymptotic Model** is the most coarse and least involved of the possible communication performance models and boils down to making statements using the family of Bachmann-Landau notations such as  $O, \Theta, \Omega, o$ . While allowing one to reason about scalability this model precludes any absolute performance statements as the constants remain unspecified. For example an implementation of a broadcast algorithm could be imprecisely described as having a time complexity of  $T_{BC} = \Theta(m + \log(P))$ , where  $m$  is problem size.

**Dominant term exact model** This model comes into play when it is possible to specify some significant terms and disregard lower-order terms either due to their diminishing asymptotic role or because they are simply not known. A stronger statement about the broadcast algorithm is that for example fully exploits the interconnect bandwidth under this model would be  $T_{BC} = \Theta(\log(P)) + \beta m$  compared to  $T_{BC} = \Theta(m + \log(P))$  that in reality might behave like  $T_{BC} = \Theta(\log(P)) + 2\beta m$

**Bounded(Parametrized model)** Whenever it is possible to specify most terms of an asymptotic model one can arrive very close to making absolute performance statements. The remaining few unspecified terms can be abstracted away by making parameterized upper and lower bounds (e.g. worst possible and best possible process-to-node mappings).

**Exact(Parametrized model)** Finally, it might be possible in rare occasions to define an exact model for some particular usage scenarios of a given machine. This is of course most desirable from the stand point of an application developer. For example, the cost of a barrier on a BlueGene/P is  $T_{BAR} = 0.95\mu s$  [41], independent of  $P$ .

Among the existing body of parallel machine models some focus on particular architectures while others attempt to achieve architecture independence like PRAM [51], BSP [184] or LogP [29]. Of the three architecture independent models, the Parallel Random Access Machine (PRAM) model is the simplest. All processors in this model share a global memory while each retains a private local memory. In this model all processors are synchronized after executing each instruction, while the cost of memory accesses and computational steps are the same with no penalty to interprocess communication. While useful for coarse-grained algorithm classification PRAM cannot be used to identify potential performance bottlenecks in distributed memory machines since it assumes a perfect shared random access memory, i.e. infinite bandwidth and zero latency. This is a critical problem because PRAM will not discourage algorithms that have excessive amounts of interprocess communication.

On the other hand, Valiant's BSP model is of more immediate practical use, as it is an instance of a bounded parameterized model. In this model a distributed memory machine is described with the following three elements:

1. processor/memory modules
2. an interconnection network
3. a synchronizer which performs barrier synchronization

A BSP computer is thus a collection of processors each with its own local memory. Accessing local memory is fast compared to accessing remote memory, and all remote memory locations are accessible in uniform time. The communication network itself is modeled as a black box, so that algorithm designers unburdened of such details could focus on global performance. This makes BSP algorithms portable. In general, a BSP algorithm consists of a sequence of computation and communication supersteps, where each superstep is always followed by a BSP synchronization barrier. A computation superstep is a sequential program wherein the local computation can only depend on the local memory data present at the beginning of a superstep. A communication superstep consists of multiple communication operations between processors. It is assumed that in any communication superstep a processor can at most send  $h$  messages and receive  $h$  messages. Such a superstep is called a  $h$ -relation and the cost thereof is:

$$T_{comm} = hg + l \tag{3.1}$$

where  $g$  is the time per data word and  $l$  the global synchronization time. The cost of a computation superstep is expressed as:

$$T_{comp} = \frac{w}{r} + l \tag{3.2}$$

where  $w$  is the maximum number of floating point operations to perform per processor in a computation superstep and  $r$  is the processor speed in FLOPS. The cost of a BSP algorithm is thus the sum of the cost of its supersteps:

$$T_{comm} + T_{comp} = \sum_{i=1} \max\left(\frac{w}{r} + gh_i + 2l\right) \quad (3.3)$$

A disadvantage of BSP is that it enforces a programming model too strict to model general MPI algorithms. In particular it excludes communication patterns where not all processors receive or send an equal number of messages. In other words, it enforces synchronization points after each communication-computation superstep. Furthermore, the model does not allow the overlap of communication and computation. An alternative bounded parametrized model, known as the LogP model is more flexible.

According to the LogP model [29] the parallel machine is described as a distributed memory multiprocessor where processors communicate by point-to-point messages. The model describes the performance characteristics of the network while abstracting away the particulars of its structure. At any given time a processor in a LogP machine can be in one of two possible states: *operational* or *stalling*. When operational, a processor can do one of the following:

1. execute an operation on locally held data
2. receive a message
3. submit a message

The LogGP model uses the following four parameters to model the performance characteristics of a parallel machine:

1.  $L$  is an upper bound on the latency of sending a message containing a word (or a small number of words) from a source process to a target process
2.  $o$  is the overhead, defined as the time that the host processor is engaged in the transmission or reception of each message; during this time the processor cannot perform other operations.
3.  $g$  the gap between messages, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor
4.  $P$  number of participating processors; local operations take unit time per datum(word)

Furthermore, the model assumes that the network is of finite capacity, such that at most  $\lceil \frac{L}{g} \rceil$  messages can be in transit from or to any processor. Should a process

exceed this limit it will stall until such time another message can be sent without exceeding the network capacity. One can thus think of network links in this model as pipelines of depth  $L$  with initiation rate  $g$  and a process overhead  $o$  (or  $o_s$  for the sender and  $o_r$  for the receiver) on each end. It is important to emphasize that the model is *asynchronous*, i.e. processors work asynchronously and the latency of any one message is unpredictable but bounded by  $L$  in the absence of stalls. Because of variations in latency, small messages directed at the same target might not arrive in the same order. For this reason, as explicated by [14], one can conceptually associate with each processor an input and output buffer for asynchronous transmittal and receipt of messages. Thus under the model, the preparation of a message for submission to the output buffer requires  $o$  time units. Once submitted a message will be injected into the communication medium and eventually delivered to its destination. Upon arrival, messages are stored in the input buffer associated with the receiving processor. The actual acquisition can occur asynchronously, at some later time and will again cost the processor  $o$  time units. In this light we can view the parameter  $g$  as the minimum time between consecutive acquisitions or submissions by the same processor. The network can then instantly accept as many messages as allowed by the capacity. LogP has through time seen several refinements such as LogGP [7] that introduces a new parameter  $G$  to model the cost of sending large messages, LogGPC [132] that models network contention, LoOgGP from [116] and LogGPS [89] modeling synchronization overheads of sending large messages. Common to all these models is feature creep, a testament to the complexity of modern parallel computing machines.

### 3.4.1 Linear model

To model the time complexity of reduction algorithms, we will use a simple linear communication cost model consisting of three parameters:  $\alpha$  the message startup time, i.e. the time to prepare the message (copy to system buffer, etc) and the time to for the first byte to arrive at the receiver,  $\beta$  the per byte cost of message transmission, and  $\gamma$  the per-byte cost of message combination. This model originates from the work of Hockney [70]. He used the COMMS1 [4] ping-pong benchmark to measure the speed of communication between nodes of a distributed parallel computers, the Intel iPSC/860, Paragon XP/S and the Meiko CS-2. He identified two features of message passing communication as important: startup time, which determines the short-message communication and the asymptotic bandwidth which determines the long-message performance.

Using the linear model, the time to send or receive a message of size  $m$ , from one process to the other, is typically expressed as:

$$T(m) = \alpha + m\beta \tag{3.4}$$

The three parameters in this model are assumed to be message size and communicator size independent. As laid out by Definition 2.5.1 we assume that the network has no capability to overlap communication with computation, so the time to send a message is modeled as equal to the time to receive a message. In implementations of reduction algorithms, the receiving process typically combines the message with its local data upon receipt, which constitutes one complete round of the algorithm. In algorithm time complexity analysis, the runtime is typically expressed in the number of rounds required to solve the reduction problem. Using the linear model, the cost of one round where a message of size  $m$  is transmitted and combined is expressed as:

$$T(m) = \alpha + m\beta + m\gamma \quad (3.5)$$

Thakur et al [178] used the linear model to predict the performance of collective operations allgather, broadcast, all-to-all, reduce-scatter, reduce and allreduce. Rabenseifner et al [153] used the linear model to estimate the performance of tree-based reduction algorithms. Chan et al [24, 22] to evaluate the performance of different collective operation algorithms on a  $c \times r$  mesh topology. In [23] the same model is used to characterize the performance of algorithms on machines that support multiport communication links, such as the IBM Blue Gene/L supercomputer. Chan et al expand the linear model in [22] to include network contention, as well as to model the different costs of message transmission depending on whether the `MPI_Recv` was preposted or not.

### 3.4.2 Independent progress

Overlapping computation with communication is a powerful method of mitigating the effect of communication latency on the performance of parallel applications. Key to success of this method is the ability of the MPI library to make independent progress on outstanding communication operations and the presence of offloading capability on the Network Interface Controller (NIC) hardware. The overlap can then be accomplished by using non-blocking MPI communication primitives at the application level.

However, most MPI library implementation require subsequent library calls to trip the progress engine, i.e. to make progress on outstanding communication operations. To achieve overlap, the user typically performs a non-blocking call followed by an independent computation phase. The issue of communication progress becomes more important for the Rendezvous message communication protocol, where a negotiation exists prior to actual data transfer [155]. If a non-blocking communication call returns without completing the negotiation, then in absence of independent communication progress, the computation phase that follows the non-

**Table 3.1:** Evaluation of independent progress facilities on the Lynx cluster using the MVAPICH 1.9 MPI library. Measurements were conducted on an allocation of  $P = 8$  nodes, with one process per node and  $m = 4\text{MiB}$ .

Implementation	async progress	non-async progress
Native	0.024 s	0.023 s
Linear Pipeline (non-blocking)	0.0078 s	0.0071 s
Linear Pipeline (blocking)	0.0138 s	0.0092 s

blocking communication primitive will not overlap communication. Indeed, the MPI standard makes no guarantees about how the message progress is to occur.

In [155] the authors examine the potential of most commonly used network interconnects and MPI implementations to overlap communication with computation. They conclude that under the eager communication protocol (typically for small messages) all MPI implementations are able to overlap communication with computation. However to overlap the communication of large messages it is crucial for the MPI implementation to support independent progress. Modeling the exact degree of overlap in MPI programs is entirely possible as shown in [25], but the resulting expressions easily become complicated and unwieldy. However, the experiments we performed on our two platforms (Lynx, Partnership for Advanced Computing in Europe (PRACE) CURIE) have shown that overlapping communication with computation was not achievable. We have compared two variations of the Linear Pipeline algorithm (defined in Chapter 2). One version uses blocking sends and receives, while the other employs a non-blocking receive in an attempt to overlap communication with computation. If the messages are small enough for the eager protocol to be used, then the latter approach should see a performance improvement. However, if the messages have to be transmitted via the rendezvous protocol then only the presence of independent progress can guarantee that overlap indeed occurs. We have benchmarked the two implementations without and with presence of independent progress (MPICH\_ASYNC\_PROGRESS=1 MV2\_ENABLE\_AFFINITY=0). Table 3.1 shows the results of the experiment essentially confirming the older results of [155] that Mvapih even in version 1.9 still does not have proper support for asynchronous progress regardless of the subscription level of compute nodes. Increasing the subscription level significantly degrades the performance of both implementations.

## 3.5 Algorithm Complexity

In this section we present a short complexity analysis of collective reduction algorithms. First, we establish the lower bounds on time complexity of reduction

algorithms and then discuss how close the algorithms selected in Chapter 2 approximate this bound.

### 3.5.1 Lower bounds

It is illustrative to establish the lower bounds on the cost of the reduction operation and the two collective operations used as building blocks for composite algorithms: reduce-scatter and gather.

**Latency:** In reduction, every process must contribute its data by sending at least one message. These messages have to be successively combined until a fully combined data vector resides at the root process. In a single port network model at most two messages originating from different processes can be combined at the same time. This leads to a minimum of  $n$  steps, each of which costs time  $\alpha$ . Similar reasoning can be applied for the reduce-scatter and gather operations.

**Bandwidth:** The lower bound for bandwidth is derived by observing that the root node must at the very least receive a quantity of data amounting to problem size  $m$ , wherein all the information from other  $P - 1$  processes has been combined. For the reduce-scatter and gather operations the root node must receive or send  $P - 1$  blocks of size  $\frac{1}{P}m$ .

**Computation:** the lower bound on the computation can be derived by the observation that if all the computations were to be performed on a single node they would take time  $(P - 1)m\gamma$ . Assuming perfect load balancing, the computation time can be brought down by distribution to  $\frac{P-1}{P}m\gamma$ . The lower bounds are summarized in Table 3.2.

**Table 3.2:** Lower bounds for collective operations

Collective	Latency	Bandwidth	Computation
Reduce	$n\alpha$	$m\beta$	$m\frac{P-1}{P}\gamma$
Reduce-Scatter	$n\alpha$	$m\frac{P-1}{P}\beta$	$m\frac{P-1}{P}\gamma$
Gather	$n\alpha$	$m\frac{P-1}{P}\beta$	N/A

It is important to observe that it is not possible for any single algorithm to meet all three lower bounds. For example, perfect computation load balancing mandates a reduce-scatter operation in the first phase where each process is responsible for  $1/P$  of the data. In the linear cost model, this has the time complexity of  $n\alpha + \frac{P-1}{P}m\beta + \frac{P-1}{P}m\gamma$ . The result of this operation then needs to be gathered at the root for an additional time of  $n\alpha + \frac{P-1}{P}m\beta$ . This is a 2-approximation in latency and bandwidth. Since computational cost is typically lower than bandwidth by a

factor of three or more in modern systems, the requirement for meeting the lower computational bound can be slightly relaxed.

If we view the execution of an algorithm in terms of rounds, where in each round it can send, receive and combine one segment of size  $B = \frac{m}{N}$ , where  $N$  is the number of segments a message has been divided into, then the minimum number of rounds to complete the reduction (for balanced PATs) is  $n + N - 1$ . In a single port network (Definition 2.5.1),  $n$  rounds are necessary for the first fully reduced segment to reside at the root, followed by additional  $N - 1$  rounds for the remaining segments.

Using the linear cost model, we can derive the time complexity of this algorithm as follows. Assume that the per-process input data of size  $m$  is split into  $N$  segments of size  $B$ . Then the reduction time of an equi-segmenting algorithm  $\mathcal{A}$  for balanced PAT, that completes in  $R$  rounds is  $T_{\mathcal{A}}(\pi) = R \cdot (\alpha + B(\beta + \gamma))$ . Expanding  $R$ , gives us the following equation:

$$T_{\mathcal{O}}(\pi) = (n - 1)\alpha + (n - 1)B(\beta + \gamma) + N\alpha + m(\beta + \gamma) \quad (3.6)$$

Differentiating by  $\frac{d}{dN}$  and selecting the positive real root, we can determine the optimal number of segments:

$$N_{\text{opt}} = \sqrt{\frac{(n - 1)m(\beta + \gamma)}{\alpha}}$$

and the optimal segment size:

$$B_{\text{opt}} = \frac{m}{N_{\text{opt}}} = \sqrt{\frac{\alpha m}{(n - 1)(\beta + \gamma)}}$$

By substituting  $B_{\text{opt}}$  for  $B$  and  $N_{\text{opt}}$  for  $N$  in Eq. 3.6, we derive the runtime complexity of the optimal equi-segmenting reduction algorithm  $\mathcal{O}$  for balanced PATs:

$$T_{\mathcal{O}}(\pi) = (n - 1)\alpha + 2\sqrt{(n - 1)\alpha} \sqrt{m(\beta + \gamma)} + m(\beta + \gamma) \quad (3.7)$$

Table 3.3 presents the computed time complexity of some well-known reduction algorithms, including those implemented in this study and defined in Chapter 2, Chapter 5 and Chapter 6. The equations in Table 3.3 were derived using the linear cost model. For purposes of comparison, we include in the table the time complexity of the segmenting Clairvoyant algorithm defined in Chapter 6.

### 3.6 Imbalance robust collective operations

That imbalanced PATs can have adverse performance impacts on collective operations has been long known. While imbalance resilient algorithms for collective

**Table 3.3:** Time complexity of some reduction algorithms for homogeneous, fully connected full-duplex networks.

Algorithm	Communication cost (upper bound)	non-comm ops	Source
Binomial Tree	$n[\alpha + \beta m + \gamma m]$	yes	[111, 97]
Butterfly	$2\alpha n + 2\frac{(P-1)}{P}m\beta + \frac{P-1}{P}m\gamma$	yes	[22]
Parallel Ring	$(P-1+n)\alpha + \frac{(P-1)}{P}m(2\beta + \gamma)$	no	[141]
Radix-k	$\sum_{i=1}^r [(k_i - 1)]\alpha + n\alpha + \frac{(P-1)}{P}m(2\beta + \gamma)$	yes	[99]
Linear pipeline	$(P-2)\alpha + 2\sqrt{(P-2)\alpha}\sqrt{m(\beta + \gamma)} + \beta m + \gamma m$	yes	[148]
Two-tree	$4(n-1)\alpha + 4\sqrt{(n-1)\alpha}\sqrt{\beta m/2} + m\beta + 2m\gamma$	yes	[164]
Clairvoyant	$(n-1)\alpha + 2\sqrt{(n-1)\alpha}\sqrt{m(\beta + \gamma)} + m\beta + m\gamma$	no	This dissertation

Communication cost is calculated as the time required for the last process to complete execution in the worst case, for balanced PATs. Problem size is denoted by  $m$ , the number of process by  $P$  and  $n = \lceil \log_2 P \rceil$ . It is assumed that  $P$  is a power-of-two.

operations have been long proposed for shared memory architectures [35], perhaps the first to propose imbalance resilient algorithms in the domain of distributed memory machines were Mamidala et al. in [113]. The authors implement imbalance resilient barrier and allreduce algorithms that use hardware multicasts, provided by Infiniband, to dynamically re-arrange the tree topologies inherent to the algorithms. In their algorithms, the root is identified to be the process that currently holds the root token. This token can however be passed to an adjacent process in the tree, moving the root closer to the belated process. The root, which ideally would be the last arriving process, then performs a hardware multicast to conclude the collective operation.

A more comprehensive study of imbalanced PATs on application performance was reported by Faraj, Yuan and Patarasuk in [6]. The authors have examined a set of NAS parallel benchmarks and identified large imbalances in PATs at collective operation call sites. A startling result of their study is that even with explicit load balancing it is difficult to fully eliminate imbalanced PATs in cluster environments. A common observation in both of the papers is that algorithms that perform better in absence of imbalance tend to perform worse in presence of imbalance. The study by Faraj et al. shows that the performance of collective operations is sensitive to process arrival time. Another interesting result of this study is that in most of the examined applications the patterns of PATs imbalance at collective call sites remain highly correlated for sustained durations. The same authors in [140] present two algorithms for the collective broadcast operation with focus on large messages.

They split the communicator into groups of early arrived process and perform the collective operation within the groups.

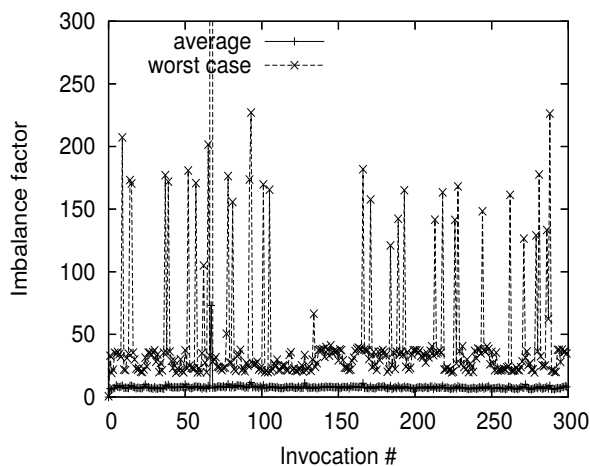
A more efficient Remote Direct Memory Access (RDMA) based solution for Infiniband alltoall and allgather algorithms is proposed in [151] that can handle both small and large messages without overhead. In their implementations, the algorithms use direct Point-to-point (p2p) communication, so that all phases of communication are mutually independent. To notify one process of another's arrival, they use RDMA control registers, which are monitored by each process. Thus no control messages are required to spread the information. The communication is then scheduled according to the order of process arrivals. While exhibiting good performance, their implementation is not portable, and limited to Infiniband interconnects.

Parsons et Pai [139] develop imbalance tolerant reduction, broadcast and all-to-all collective operation hierarchical algorithms, optimized for distributed systems consisting of large multicore shared memory nodes. Their reduction and broadcast algorithms utilize a dynamic leader selection method to opportunistically select for each node the process responsible to participate in the inter-node communication. Intra node, their algorithm use a single or multiple shared memory buffers, depending on problem size, into which early arrived processes eagerly reduce their data, using locks to guarantee mutual exclusion. In that way, processes that arrive early can reduce their data and exit the collective operation without waiting for the other processes to arrive. In this sense, the reduction schedule of their algorithm is dynamic. They compare their algorithm against the native MPICH 3.0.4. hierarchical binomial tree implementation and report speedups exceeding 10x on 32 thousand cores of the Cray XE6 supercomputer. However, as was discussed in Section 3.3.2, reporting the mean instead of total runtime for collective operation runtime, tends to overestimate speedup.

Another dynamic leader selection based algorithm, for imbalance robust pipelined broadcast was proposed by Liu et al [109] where intra-node communications can be overlapped with inter-node communication. Shared memory was used to perform intra-node communication, while the inter node communication was implemented via MPI p2p primitives.

### **3.7 An argument for clairvoyancy: PAT cognizant reduction algorithms**

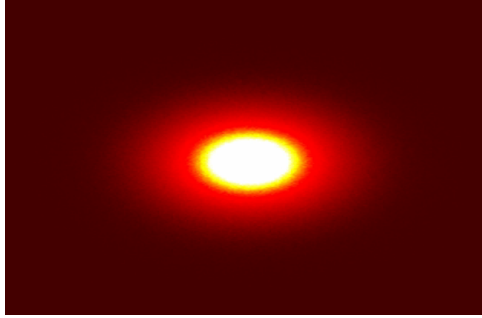
In this section we present an argument in defence of the assumption that it is feasible to provide information on PATs to collective operations at runtime. Much of the argument will center on the assumption that it is feasible to provide information on PATs to collective operations at runtime. We discuss the difficulties present therein, and the costs involved.



**Figure 3.4:** Imbalance factors for *MPI\_Allgather* in NBODY on Lemieux cluster ( $P=128$ ). Reprinted from "A study of Process Arrival Patterns for MPI Collective Operations", by Faraj A. Et al., 2008., International Journal of Parallel Programming, Volume 36, Issue 6, pp 543-570. Reprinted according to fair use.

Faraj et al. [6] conducted a study of PATs at collective operation call sites for various MPI routines in a set of MPI benchmarks consisting of HPC application kernels. They found that PATs for different invocations of the same collective operation exhibit a phased behaviour: PATs are strongly autocorrelated for a period of time before they change (Fig. 3.4). For some collective operation call sites, the PATs are autocorrelated for the entire program duration. These findings indicate that it might be feasible to construct a model in the form of a stochastic difference equation (such as ARMA) to predict PAT patterns from one invocation to the other.

Motivated by their findings, we performed a trace of the per process image rendering time across 100 iterations of the in-situ visualized Helsim particle-in-cell space weather simulation on  $P = 128$  processes with 8 processes per node, on the Lynx cluster machine (Fig. 3.5). Helsim is an Electromagnetic Explicit 3D In-Situ-Visualized Resilient Particle-in-cell simulator, developed in the Leuven Intel ExaScale Lab, Belgium. It is a combined multidisciplinary effort integrating astrophysics, linear solvers, runtime environment, in-situ visualization and architectural optimization focused simulations. It was developed to be a proto-app, showing a realistic example of trade offs between computation and communication on a small, manageable code-base with modern implementation techniques. It was implemented in C++11 utilizing the inlab Shark PGAS library for all distributed data structures and the Cobra library for load balancing and resiliency. As was explained in Chapter 2, in sort-last distributed rendering, each process produces one full sized image of



**Figure 3.5:** A render of one time step in the evolution of the Helsim simulation for spherically distributed data in a 64x64x64 cell structure, 2 particle species and 6x6x6 particle arrangement per species, per cell.

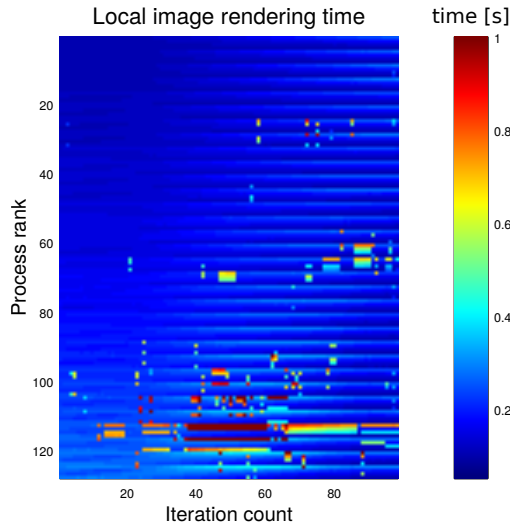
the data that is subsequently composited into the final image with a global reduction operation. The variance in local image rendering time then manifests as an imbalance in PATs at the collective image compositing operation. A depiction of the variance evolution across simulation iterations is presented in Fig. 3.6.

The PATs of the first 24 processes, exhibited a recurring pattern (Fig. 3.7) with a period of 4 process ranks. The non-randomness and strong trends in the per-process PATs indicate that it might be feasible to construct a stochastic difference equation based model to predict PAT patterns in this setting. A simple moving average model (SMA) with a window size of 5 was shown to fit the data very well (Fig. 3.7). This conjecture is further reinforced by the autocorrelation plots of the data (Appendix A.1). In Chapter 5 we will investigate the fitness of the SMA model for PAT aware, i.e. clairvoyant reduction algorithms.

The clairvoyant schedule generation algorithms introduced in Chapter 5 and Chapter 6 require that the entire PAT pattern be known at the time of schedule construction. In an iterative setting, this might be accomplished by communicating the PAT pattern every  $k$  iterations to all the processes in the communicator with an all\_gather operation, and relying on a PAT prediction model to construct the PAT patterns in-between the communications. Our findings (Fig. 3.7 and that of Faraj [6] indicate that this indeed might be feasible.

However, for this to be an efficient approach, the number of iterations  $k$  has to be sufficiently large so that the speedup brought about by the clairvoyant algorithm amortizes the PAT pattern dissemination cost. Because each process only communicates a single floating point value, the dissemination cost will become negligible for moderate to large problem sizes  $m > 128$  KiB, that are considered in this dissertation. Moreover, the clairvoyant schedule generation algorithm should be robust to the small inaccuracies produced by the model predictions.

From a design standpoint, it would be best to incorporate the schedule construc-



**Figure 3.6:** Distribution of image render time across the 100 iterations of the Helsim simulation, Lynx cluster  $P = 128$  with 8 processes per node. The values range from 0s to 1s.

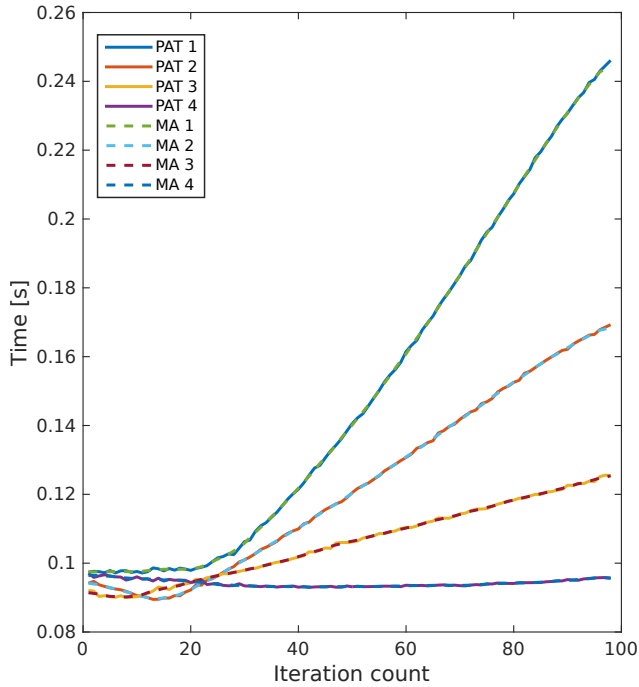
tion within the reduction operation. In this way, the whole process would be entirely transparent to the user and delegated to the library runtime system. An environment variable could be used to toggle the usage of the PAT imbalance features in the library implementation.

## 3.8 Summary

In this chapter we discussed what effects system noise and load imbalance have on the performance of collective operations and formally defined the idea of Process Arrival Time (PAT). We briefly examined how two different measures of collective operation runtime (average per process runtime and time-to-solution runtime) and how they lead to wildly different relative algorithm performance figures.

Following that, a brief survey of algorithm complexity models was introduced and the linear model adopted in this dissertation was elaborated. We argued the absence of computation-communication overlap in our abstract performance model with the experimentally confirmed lack of independent progress on our experimental testbed: the PRACE CURIE supercomputer.

The chapter is concluded with a discussion on the lower bounds attainable by reduction algorithms and with a presentation of time complexity equations for each of the algorithms examined in this work.



**Figure 3.7:** Examples of the 4 principal clusters of image render times across the 100 iterations of the Helsim simulation, valid for process ranks  $p < 24$ . The displayed PAT sequence plots are those for ranks 0-4. Superimposed on each pattern is the moving-average fit computed with a window of size 5. The data was originally gathered on the Lynx cluster with  $P = 128$  and 8 processes per node (ppn=8). The x-axis denotes iteration count, while the y-axis the image render time in seconds

## Chapter 4

# Benchmarking Parallel Computing Systems

---

Benchmarking of parallel computer systems and parallel algorithms constitutes one of the driving forces of High Performance Computing (HPC) scientific advancements. Typically, the objective of benchmarks is to assess the relative performance of a system or an algorithm by running a set of standard tests and trials. Some benchmarks attempt to discover the key architecture features of parallel systems, while others attempt to offer insights into how typical HPC applications might be expected to perform on a given system. For example, the High Performance Linpack (HPL) benchmark [144] is used to rank the world's most powerful supercomputer systems into a bi-annually updated list, known as Top500. HPL is a benchmark whose kernel is a dense linear system solver in double precision arithmetic. This is a common task in engineering and science and the result of this benchmark, expressed in aggregate floating point computing power can be used to gauge the performance of large system when running typical HPC systems.

Another representative benchmark, or set of benchmarks, is the NAS Parallel Benchmarks (NPB) [10]. NPB is composed of a set of programs designed to help evaluate the performance of supercomputers by running code representative of typical scientific HPC applications. Originally the programs were derived from Computational Fluid Dynamics (CFD) applications, but have been extended to include new benchmarks for unstructured adaptive mesh, parallel I/O, multi-zone applications and computational grids.

In the domain of parallel and distributed computing, there is little consensus on what constitutes a set of standard techniques for *measuring and reporting* the performance of parallel computational systems. In this chapter we present a review of state-of-the-art in Message Passing Interface (MPI) performance benchmarking and present a benchmarking suite developed for assessing the performance of MPI collective operations under various patterns of imbalanced Process Arrival

Times (PATs). We also discuss in some depth the statistical analyses that are necessary in establishing the significance of experimental findings, something that is conspicuously absent in most works in this domain [77].

## 4.1 Hardware Platforms

We begin by briefly delineating the three cluster systems that were used in generating the experimental data analyzed in this dissertation. The first machine, Lynx, was primarily used for code development, debugging and initial testing and validation. All the data presented and analyzed in Chapter 5 and Chapter 6 were generated and collected on the VSC muk cluster and the Partnership for Advanced Computing in Europe (PRACE) CURIE supercomputer.

### 4.1.1 Lynx

Lynx is a 32 node cluster installed at the Intel ExaScale Lab in Leuven, housed at IMEC. Each node is a DL170e G6 blade: dual socket, six core Intel Xeon X5660@2.8 GHz, 96 GB of memory and 500 GB of disk space. Each node comes with a Mellanox Technologies MT26428 Infiniband card (ConnectX VPI PCIe 2.0 5GT/s - IB QDR/10GigE). The nodes are interconnected using a single Voltaire 36P QDR switch. This is a crossbar switch, so this network should achieve full bisection bandwidth for all communication patterns. All nodes ran Ubuntu 12.04.3 LTS Precise.

### 4.1.2 VSC muk

The second machine was the VSC muk cluster in Ghent, Belgium. This cluster consists of 528 computing nodes, each with two 8-core Intel Xeon processors from the Sandy Bridge generation (E5-2670, 2.6 GHz). Each node features 64 GiB of RAM, for a total memory capacity of more than 33 TiB. The computing nodes are connected by an FDR InfiniBand interconnect with a fat tree topology. The nodes ran Scientific Linux release 6.7 (Carbon). The resource manager was Torque 4.2.9 and the job scheduler MOAB 7.1.0. All experimental data was produced on allocations of 32 compute nodes, with one process per node.

### 4.1.3 PRACE CURIE

CURIE is a supercomputer located in France at the Très Grand Centre de Calcul (TGCC), operated by CEA near Paris. It was installed in two phases, starting at the end of 2010 and is now fully operational. During this period, CURIE has

gradually been made accessible for research purposes through the PRACE Regular and Preparatory Access Calls.

CURIE is a BULL x86 system based on a modular and balanced architecture of thin (5040 blades each equipped with 2 Intel Xeon E5-2680 8 core processors running at 2.7 GHz and 64 GB of RAM, interconnected using a fat-tree Infiniband QDR interconnect), large (90 servers, each with 128 cores and 512 GB of memory) and hybrid (144 blades, each with 288 Nvidia M2090 Graphics Processing Units (GPUs)) nodes with more than 360 TB of distributed memory and 15 PB of shared disk. Altogether, CURIE can deliver a peak performance of 2 PFLOP/s (2 million billion operations a second), as measured in HPL [150].

In the experiments conducted for the purposes of this dissertation, only the thin nodes were allocated. All the nodes ran the Bullx Supercomputer Suite RC1 operating system, based on Red Hat Enterprise Linux 6 beta.

## 4.2 Measurements of elapsed time

A major obstacle in accurate and reproducible time measurements on parallel systems is the absence of time synchronization. This is particularly obvious in cluster systems, where every node is a complete and independent system with its own local time source.

### 4.2.1 Hardware clocks

Most computers come equipped with a quartz crystal clock mechanism that oscillates with a fixed frequency that continuously updates a register that can be used to count the number of oscillations since a certain point in time (for example, the system startup event). However, not all devices offer the same interface to access this information. Some devices are configured to issue interrupts when the register reaches a certain value, while others allow the programmer to read the register explicitly [83].

The basic computer clock, present in most modern PCs is the Real Time Clock (RTC). These clocks are typically supplied with a separate power source enabling them to keep time while the primary source is off or unavailable. Due to low resolution of RTC, beginning with 2005, a High Precision Event Timer (HPET) hardware timer, developed jointly by Intel and Microsoft, was incorporated into PC chipsets. An HPET consists of a 64-bit up-counter, counting at a frequency of at least 10 MHz and a set of comparators. Each comparator can generate an interrupt when the least significant bits are equal to the corresponding bits of the 64-bit main counter value.

However, for very fine grained timing, neither of the afore mentioned timers are sufficient. To that end, modern Central Processing Unit (CPU)s based on the x86 Instruction Set Architecture (ISA) provide Time Stamp Counter (TSC) hardware counters. The TSC is a 64-bit register that is monotonically incremented by the processor and set to zero whenever the processor is reset [90]. The hardware instruction RDTSC returns the TSC in EDX:EAX register. Other manufacturer's ISAs offer similar facilities: for example, the SPARCv9 ISA provides the TICK register.

Modern CPUs employ power saving features such as dynamic core clock changes, based on observed demand, that can result in invalid time measurements if clock registers are used. It is recommended that cluster systems disable such features. Another problem that may occur on multi-core systems is the migration of processes from one core to the other, dictated by the operating system, as the counters on different cores might not be identical.

### 4.2.2 Synchronization

In distributed systems, each machine (compute node) has its own physical clock. Due to their manufacturing nature, quartz crystal clocks in each of the machines (even if they are produced as identical models) run at slightly different frequencies, causing the clock values to slowly drift from each other, once they have been synchronized. This divergence is formally called *clock skew*. It was shown that this drift is sufficient to distinguish/identify single computers and sometimes even the timezone in which they reside [?]. If the clock difference is roughly linear, then it can be corrected by a linear error function [83]. Clock synchronization has to be repeatedly performed to robustly correct this skew in distributed systems [175]. Physical clock synchronization algorithms attempt to coordinate distributed clocks to reach a common value. They are based on estimates of transmission delay which may not always be simple to estimate [34].

Clock synchronization of a group of processes in a communicator is based on synchronization of pairs of processes. A clock synchronization between two processes can be established using a ping-pong scheme, similar to the one previously described: two processes (server and client) calculate their clock difference so that the client process knows his clock offset relative to the server. This offset can then be subtracted from the clients local time when clock synchronization is required [34, 83]. Because the observed Round Trip Time (RTT)s can significantly vary in value, many synchronization algorithms adopt an approach where RTT measurements are conducted as many times as necessary to ensure that the observed RTT does not become smaller for  $N$  consecutive measurements. While this scheme is guaranteed to converge, it is not possible to know a priori how many measurements  $N$  will have to be conducted. In the design of Netgauge, the authors selected  $N = 100$ , which

had a measured error less than 10% on the networks tested and converged after approximately 180 measurements [83].

The pair synchronization algorithm can easily be extrapolated to a group of  $P$  processes by having one process (root) perform Point-to-point (p2p) synchronization with the remaining  $P - 1$  processes in sequence.

### 4.2.3 MPI timer

The Message Passing Interface (MPI) defines a timer through the function `MPI_Wtime`. This function returns a floating point number of seconds, representing elapsed wall-clock time since some event in the past [125]. The fact that the function returns elapsed time in seconds makes it portable.

Ordinarily, the time returned by calls to `MPI_Wtime` is local to the node where it was invoked. However, if the flag `MPI_WTIME_IS_GLOBAL` is set, then the system supports a global synchronized clock.

The resolution of `MPI_Wtime` can be queried by a call to `MPI_Wtick`, which returns a double precision value denoting the number of seconds between successive timer ticks.

## 4.3 Benchmarking of Message Passing primitives

Benchmarking of message passing primitives can broadly be divided into benchmarks that assess the performance of p2p operations and benchmarks that assess the performance of collective operations. The former are typically conducted to ascertain key features of a parallel system, such as communication latency, bandwidth and overhead associated with message passing communication. As most collective operations are implemented in terms of p2p primitives, a thorough understanding of p2p primitives performance on a given architecture is paramount to efficient implementation of collective operations.

### 4.3.1 Point-to-Point operations

The ping-pong latency benchmark [69] is the most commonly used technique for assessing the latency and bandwidth of p2p communication. In this experiment, one node sends a single message to another. After receiving the message, the second node sends a reply. This process is typically repeated several times to obtain an average round-trip latency. The time measurements are performed only on the sender side. This is typically done because most systems lack an accurately synchronized global clock with a resolution sufficient to measure data transmissions across the network (in the order of microseconds). Benchmarks such as NetPIPE [172], Netperf [93], Intel MPI [91], MPIBench [60] are all based upon this method. The method was

subsequently patented by Faraj in 2009 [39] and is presented in pseudo code form as Listing 4.1.

**Listing 4.1:** Method for determining communications latency for transmission between nodes in a data communication network (patented by Faraj [39])

```
/****** SENDER *****/
MPI_Recv(&ack,0,MPLCHAR,receiver,tag,comm,&status)
start=MPI.Wtime();
MPI_Send(sbuff,size,MPLCHAR,receiver,tag,comm);
MPI_Recv(rbuff,size,MPLCHAR,receiver,tag,comm,&status);
rtt=MPI.Wtime()-start;
owt=rtt/2;
/***** RECEIVER *****/
MPI_IRecv(rbuff,size,MPLCHAR,sender,tag,comm,&request);
MPI_Send(&ack,0,MPLCHAR,sender,tag,comm);
MPI_Wait(&request,&status);
MPI_Send(sbuff,size,MPLCHAR,sender,tag,comm);
/*****
```

However, the results of this simple measurement are widely held to be poorly representative for the communication patterns of real applications [18]. This is due to the latency evaluations being conducted with only one item in the posted receive queue and the bandwidth extrapolated from communication with pre-posted receives.

In real world applications, typical communication patterns will see multiple items in the posted MPI receive queue. The size of this queue was shown to have direct impact on latency [18, 182], and exhibits near linear increase in latency as function of queue size. For example, Hoeffler et al [79] modeled the matching queue overhead as  $206 \text{ ns}R$ , on the BlueGene/P system, where  $R$  denotes the number of outstanding messages in the queue. They illustrate how a simple 27-point stencil would cause a traversal of an average of 13 requests, adding  $2.678 \mu\text{s}$  to the latency before the right message is matched.

Unexpected messages, that typically occur during runs of real applications, can also have tangible impacts on communication latency. Ping-pong tests are often conducted with a high number of iterations for two principal reasons. On many systems the clock resolution is not fine grained enough, so larger time intervals are necessary for such timing to return meaningful results. This approach, however, averages out the variations that might be present in individual latency measurements with the trade off of being more repeatable due to a large sample used to compute the average. The authors in [161] present a latency micro-benchmark designed to capture the variance in p2p transmission time.

### 4.3.2 All pairs communication microbenchmark

In contrast to the ping-pong experiment (Listing 4.1), we decided to construct an experiment that more closely reflects the communication patterns of tree based reduction algorithms, like Binomial Tree reduction defined in Chapter 3 and the algorithms that are introduced and discussed in Chapter 5.

In binomial tree reduction, at any stage, half the outstanding processes send the data, while half the processes receive and combine the data. Only processes that have received the data proceed to the next round of reduction. Thus, to accurately assess the duration of one round of reduction it suffices to measure the time it takes for a message of size  $m$  to be *received and combined*, in a communication pattern where  $P/2$  processes are sending a message of size  $m$  and  $P/2$  processes are receiving a message of size  $m$ . Multiplying the cost of one round with the total number of rounds  $R$  (for binomial tree reduction,  $R = \lceil \log_2 P \rceil$ ) gives us a good estimate of binomial tree reduction run time.

However, the number of participating processes and the particular communication pattern (or process grouping) can have tangible performance impact on Infini-band fat-tree networks [202, 82], as the networks feature multi-stage switches with static routing leading to potential link over subscription for some communication patterns. In order to achieve a higher level of confidence on binomial tree reduction runtime, one should measure the time to receive and combine a message of size  $m$  for all groups of processes  $P, P/2, \dots, 2$  that occur in the stages of the algorithm execution.

The microbenchmark developed to sample such data (Algorithm 6), measures the time process  $i$  takes to receive a message of size  $m$  from process  $j$  in a communication pattern where all process pairs in a communicator are exchanging messages concurrently. The measurements are conducted for  $\forall i, j : 0 \leq i, j < P, i \neq j$  so that in each round, process  $i$  sends a message to process  $j$  and process  $j$  receives and combines a message from process  $i$ , and vice-versa.

In each round, a total of  $\frac{P}{2}$  communicating pairs are established and synchronized so that  $\frac{P}{2}$  messages are transmitted simultaneously. Let such a partition be called a  $\frac{P}{2}$  *pairs communication pattern*. When  $P$  is a power-of-2, this leads to a total of  $P - 1$  rounds for an all pairs measurement.

This measurement is, however, by no means exhaustive. Splitting the  $P$  processes into two equally sized partitions can be performed in  $\binom{P}{\frac{P}{2}}$  different ways. Once a partitioning has been chosen, each process from one partition has to be mapped to a process in the other partition: but this is a bijection from a set to itself, i.e. a permutation. It follows that the total number of  $\frac{P}{2}$  pairs communication patterns is:

$$\binom{P}{\frac{P}{2}} \prod_{i=1}^{\frac{P}{2}} i = \frac{P!}{\frac{P}{2}!}$$

The communication patterns for which we measure the transmission times are a tiny fraction of this number. In fact,  $P - 1$  is the minimum number of rounds to sample all-pairs message transmission times from  $P$  processes, by using the  $\frac{P}{2}$  pairs communication pattern.

Zahavi et al [202] showed that exhaustively testing the entire set of possible communication patterns might not be necessary. They used a collective permutation sequence classification to classify the permutation of source-destination pairs that communicate at each stage of the collective operation. They show that most collective operations (as enumerated in [146, 178]) are realized with only 8 different permutation sequences: binomial tree, butterfly recursive doubling, recursive halving, dissemination, reverse dissemination, tournament, ring and shift.

As discussed earlier, the number  $\frac{P}{2}$  was chosen as the potential worst case scenario out of the communication patterns expected to arise from tree reduction algorithms defined in Chapter 5.

---

**Algorithm 6:** All pairs communication microbenchmark

---

**Require:** Input data is non-atomic  $\wedge P = 2k, k \in \mathbf{N}$

**Ensure:**

S0 Let  $P$  be the communicator size

S1 For  $i = 0, \dots, P - 1$

S1.1 Generate a new group  $g$  of  $P/2$  unique pairs

S1.2 Split the main communicator into  $P/2$  communicators defined by group  $g$

S1.3 For each new communicator

\* Perform a double barrier

\* Let process rank 0 send a msg of size  $m$  to process rank 1

\* Measure the time to receive a msg at rank 0

\* Measure the time to combine the received msg at rank 0

\* Repeat the measurements, reversing the roles (rank 1 send to rank 0)

---

The  $P - 1$  partitions of  $P/2$  pairs of processes were generated with a deterministic algorithm, built upon a bi-directional greedy strategy of selecting the  $\frac{P}{2}$  pairs communication patterns. We present the all-pairs generation algorithm in its native ISO C++11 form as Listing 4.2.

**Listing 4.2:** Deterministic algorithm for generating  $P - 1$  partitions of  $P/2$  pairs used in the microbenchmark defined by Algorithm 6

```
//1st step: generate all pairs among P processes
list<pair<int ,int>> pairs;
for (int i=0;i<P-1;i++)
    for (int j=i+1;j<P;j++)
```

```

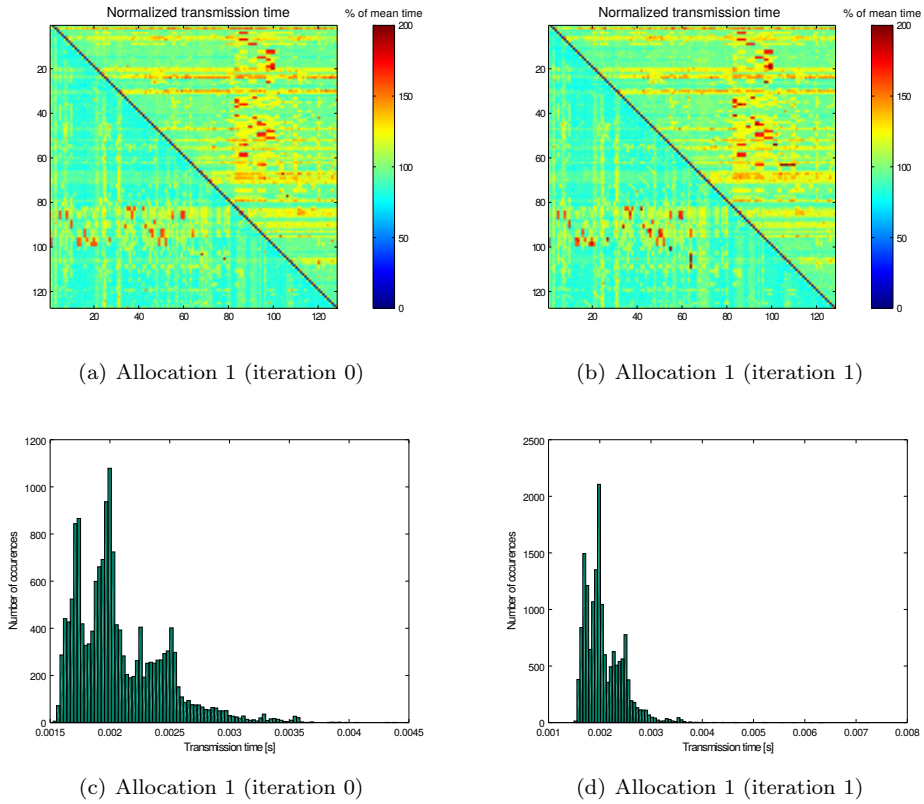
    pairs.push_back({i, j});

for(int i=0; i<P-1; i++) {
    vector<pair<int, int>> selected;
    set<int> found;
    while (selected.size() < P/2) {
        //First look for a suitable pair from the end
        auto rit = find_if(pairs.rbegin(), pairs.rend(),
            [&found](pair<int, int> &y) {
                return found.find(y.first) == found.end() &&
                    found.find(y.second) == found.end();}
            );
        selected.push_back(*rit);
        found.insert(rit->first);
        found.insert(rit->second);
        // Once found, the pair will not be considered again
        pairs.erase(--rit.base());
        //Now find one from the beginning
        auto it = find_if(begin(pairs), end(pairs),
            [&found](pair<int, int> &y) {
                return found.find(y.first) == found.end() &&
                    found.find(y.second) == found.end();}
            );
        selected.push_back(*it);
        found.insert(it->first); found.insert(it->second);
        pairs.erase(it);
    }
}

```

We now present and discuss the results of the p2p data transfer microbenchmark (Algorithm 6). For each message size  $m \in \{40 \text{ kB}, 400 \text{ kB}, 4 \text{ MB}, 40 \text{ MB}\}$ , 5 different allocations of  $P = 128$  nodes were acquired using the native slurm job scheduler (version slurm 2.6.9-Bull.1.1) on the PRACE CURIE supercomputer. For each of the five allocations, ten consecutive iterations of the microbenchmark were performed. For brevity, we will only present in detail the results obtained for  $m = 4 \text{ MB}$ . The five node allocations obtained were as follows:

1. Nodes={1500-1501,1503,1505-1507,1514-1515,1755,1758,1760-1764,1766-1769,1789-1790,1793-1796,1801,1803,2997-2998,3001-3002,3004,3007-3009,3011,3391-3392,3395,3401-3402,3404-3407,3823-3824,3826,3828,3832-3833,3836,3838-3839,3930,3934,3937-3938,3940-3942,3946-3947,4290,4292-4294,4303-4307,4470-4473,4482-4505,4560-4563,4572-4577,5716-5718,5722,5724-5727,5729,6239,6244-6251}
2. Nodes={1933-1934,1937-1939,1944,1947,1949,2008-2014,2058,2063,2065-2067,2069,2074,2194,2196-2201,2245,2247-2252,2310-2317,2328-2329,2331,2333,2336-2337,2345,2724,2727,2729,2734,2738,2740,3095-3101,3265-3271,3280,3336,3338,3342,3347-3348,3350,3352-3353,3471-3476,3478-3479,3912,3914,3916-3917,3920,3924,3926,3929,5280-5283,5292-5295,5316-5317,5319-5320,5325-5328,6026-6029,6032-6035,6092-6094,6098,6103,6105-6107}
3. Nodes={1853,1855,1858-1859,2148,2151-2152,2156,2159,2228,2230-2233,2350-2353,2421-2423,2432,2906-2907,2910-2911,2988-2993,3127,3129,3133-3135,3140-3141,3146,3150-3151,3154,3230,3235,3240,3243,3354-3357,4434-4437,4460-4463,4468-4469,4838,4842-4844,4958,4968,4970,4972-4973,4977,4979,4981-4982,5183-5189,5903-5909,5954-5957,6059-6062,6111,6113,6115,6120-6121,6124-6126,6128,6132,6134-6135,6137,6142,6272,6278-6282,6342-6343,6345,6349,6352,6355,6434,6436,6439-6441,6444}



**Figure 4.1:** p2p message transmission time (PRACE CURIE; 128 nodes,  $m = 4$  MB)

4. Nodes={1500-1501,1503,1505-1507,1514,1755,1758,1760-1764,1766-1769,2997-2998,3001-3002,3004,3007-3009,3011,3391-3392,3395,3401-3402,3404-3407,3823-3824,3826,3828,3832-3833,3836,3838-3839,3930,3934,3937-3938,3940-3942,3946-3947,4110-4127,4290,4292-4294,4303-4307,4470-4473,4482-4505,4560-4563,4572-4577,5716-5718,5722,5724-5727,5729}
5. Nodes={1592,1605,1607,1789-1790,1793-1796,1801,1803,1933-1934,1937-1939,1944,1947,1949,2008-2014,2058,2063,2065-2067,2069,2074,2194,2196-2201,2245,2247-2252,2310-2317,3265-3271,3280,3336,3338,3342,3347-3348,3350,3352-3353,3471-3476,3478-3479,3912,3914,3916-3917,3920,3924,3926,3929,5280-5283,5292-5295,5316-5317,5319-5320,5325-5328,6026-6029,6032-6035,6092-6094,6098,6103,6105-6107,6239,6244-6251}

For each allocation and each iteration, the same  $P - 1$  pairings of  $\frac{P}{2}$  processes were evaluated. We selected 5 different allocations of  $P$  nodes in order to determine, what if any effect the scheduling policy has on message transmission time.

The results of the microbenchmark (Fig. 4.1, Appendix A.2 indicate a high degree of variation in transfer time for messages of size 4 MB and  $P = 128$ . Interestingly, the distribution of message transmission times appears to be multimodal (Fig. A.3). We conjecture that the multimodal distribution of message transfer time is a consequence of a static routing protocol employed in Infiniband multistage switches of the tested machine. Static routing might oversubscribe some of the

**Table 4.1:** Message transmission time as function of concurrently communicating pairs. Mean times are taken over all node allocations and normalized to  $t_1$ .

$m$	$\mu(t_{64})$	$\mu(t_{32})$	$\mu(t_{16})$	$\mu(t_8)$	$\mu(t_4)$	$\mu(t_2)$	$\mu(t_1)$
4 MB	1.0777	1.0346	1.0190	1.0081	1.0012	1.0067	1.0000
2 MB	1.1035	1.0492	1.0244	1.0127	1.0076	1.0045	1.0000
400 kB	1.1534	1.0723	1.0317	1.0162	1.0084	1.0075	1.0000
200 kB	1.1648	1.0982	1.0287	1.0251	1.0142	1.0135	1.0000
40 kB	1.1017	1.0521	1.0027	0.9996	1.0058	1.0137	1.0000

physical links in the network for certain communication patterns, reducing the available bandwidth. This conjecture is made stronger by the observation that a nearly identical distribution recurs for all the 10 performed iterations for the same allocation of nodes (only two of which are shown in Fig. 4.1, Fig. A.3).

Hoefler et al. [82] showed that static routing in networks with full bisection bandwidth (like Infiniband) results in effective bisection bandwidth of only 55-60% of best case performance. They too report a multimodal distribution of message transmission times in the 512 node bisection experiment on the CHiC Infiniband system. Similar patterns were observed in the other 4 examined allocations.

Finally, examining the combined cost of the message receive and combination, we can observe that most of the variation is accountable to the variation in network data transmission times (Appendix A.4). This is illustrated quantitatively in Table 4.2.

Most rounds in tree reduction algorithms involve communication of less than  $\frac{P}{2}$  pairs. To determine whether the number of simultaneous communicating pairs has impact on message transmission time, we conducted an experiment wherein the number of communicating pairs was progressively halved from  $\frac{P}{2}$  to 1. Each communication pattern was generated by uniform random shuffles, for a total of 100 generated and measured patterns and 6400 produced samples. The results of this experiment are shown in Table 4.1. We can observe that the time  $t_p$  of transmitting one message does depend on the number of concurrently communicating pairs of nodes  $p$ , but only by a small margin. The ratio  $\frac{t_{64}}{t_1}$  ranges from 7.8% to 16.5%, with tendency of increasing with decreasing message size. We conjecture that this ratio would be bigger still with a larger communicator.

In light of the data presented, it would appear that the assumption of a homogeneous network proposed in Chapter 2 is not met in practice. A more realistic model of message transmission cost would take into account not only message size, but the type of communication pattern, the topology of allocated nodes within a network and the network topology itself. This is in line with the observations reported in [82, 202]. On the other side, Zahavi showed in [202] that with topology aware MPI node ordering much of the hot spots in fat-tree networks like Infiniband can be eliminated and the network made to appear homogeneous for the typical

communication patterns exhibited by collective operations.

**Table 4.2:** Message transmission and combination time across different node allocations. Each sample represents a different allocation of P=128 nodes. Problem size  $m = 4$  MB

Sample	$\mu(\alpha + m\beta)$	$\sigma(\alpha + m\beta)$	$c_v(\alpha + m\beta)$	$\mu(m\gamma)$	$\sigma(m\gamma)$	$c_v(m\gamma)$
1	0.00208 s	$3.69 \times 10^{-4}$ s	0.177	$6.88 \times 10^{-4}$ s	$8.43 \times 10^{-6}$ s	0.0122
2	0.00202 s	$3.15 \times 10^{-4}$ s	0.156	$6.88 \times 10^{-4}$ s	$8.85 \times 10^{-6}$ s	0.0129
3	0.00205 s	$3.22 \times 10^{-4}$ s	0.157	$6.88 \times 10^{-4}$ s	$7.41 \times 10^{-6}$ s	0.0108
4	0.00205 s	$3.67 \times 10^{-4}$ s	0.179	$6.88 \times 10^{-4}$ s	$7.03 \times 10^{-6}$ s	0.0102
5	0.00207 s	$3.40 \times 10^{-4}$ s	0.164	$6.88 \times 10^{-4}$ s	$7.09 \times 10^{-6}$ s	0.0103

### 4.3.3 Collective Operations

At the moment, there seems to be no consensus on how to microbenchmark and report the performance of collective operations. Upon close examination, three principal approaches can be identified. The first approach uses explicit synchronization of participating processes by calling library synchronization routines (`MPI_Barrier`), such as the Intel MPI benchmarks [91]. The second approach, relies on global clock synchronization to ensure that all processes commence the collective operation simultaneously, such as the one introduced in the Special Karlsruher MPI Benchmark (SKaMPI) microbenchmark [192, 193, 2], Netgauge [81, 83] or MPIBench [60]. Finally, a quality based approach that reports the performance of collective operations relative to the cost of a point-to-point operation has been proposed by Shroff et al, for relative comparison of different library implementations [169].

Before proceeding to elaborate the microbenchmark scheme adopted for the purposes of this dissertation, we will first present an overview of the various microbenchmarks for collective operations that are publicly available. On closer inspection, it is clear that there is a lot of disparity in what particular microbenchmarks measure:

1. time between two events on a single designated process
2. for every process in the communicator, the time span between two events
3. the time between two events on different processes

The first two measurements are a natural choice in environments where hardware implemented globally synchronized clocks are absent. However, it is possible to measure the time between two events on different processes by using one of the several global clock synchronization algorithms [192, 83] and window-based mechanisms to issue calls to collective operations. This mechanism is contingent

on the assumption that the difference between local clocks resident on each process does not change, or change in a predictable and hence correctable manner.

**MPPTTEST** is a microbenchmark developed by the MPI over Chameleon (MPICH) group designed from the outset with reproducibility as its primary goal [59] and was originally used to measure and characterize the performance of pre-MPI message passing implementations. Before measuring the performance of a collective operation, a warm-up run is performed to establish initial communication links and warmup the CPU caches. Following that, all the processes in the communicator are synchronized with a call to `MPI_Barrier` library function before the target collective operation is run in sequence  $N$  times. The reported elapsed time is that recorded on rank 0, assumed as the root process. To ensure reproducibility, this microbenchmark reports the minimum observed time.

**MPBench** is a microbenchmark developed by Mucci et al [134] to evaluate the performance of MPI and Parallel Virtual Machine (PVM) library implementations. A peculiar feature of this microbenchmark is that it eschews any synchronization prior to measuring collective operation run time. By default, collective operations are evaluated for messages ranging from 4 bytes to 16 Megabytes, in powers of two for 500 iterations. All the time measurements are performed locally on process rank 0, designated as the root process.

**Intel MPI Benchmarks** use a double `MPI_Barrier` call to synchronize all the processes in the communicator prior to measuring the collective operation run time. The microbenchmark reports the minimum, maximum and mean recorded time on all processes. The timings are averaged over multiple samples. For reduction operations, the basic data type is `MPI_FLOAT`.

**SKaMPI** uses a time-window based approach [158] to ensure that all processes in the communicator start the collective operation simultaneously. The number of samples per operations is determined automatically by controlling both the systematic and statistical error. The latter is controlled by stopping the repetitions once the cumulative coefficient of variation is smaller than a specified threshold value, i.e  $\sigma_{\bar{x}}/\bar{x} < \epsilon$ . An adaptive refinement method is employed to sample the problem space of message sizes  $m$  and process counts  $P$  to derive a linear model describing the behaviour of the operation.

**MPILib** is a microbenchmark [105] designed to address a wide range of user requirements by offering a high degree of customizability. It allows users to select either global, maximum or root timings, a fixed or variable number of repetitions, where in the latter samples are collected until a certain confidence level (user supplied) has been attained. It comes with as set of gnuplot scripts for visualization of gathered data.

**MPIBench** is a microbenchmark designed to accurately measure the variations in run times of individual collective operation calls, which cannot be recorded by the previous three listed microbenchmarks that report averages of multiple

samples [60]. The microbenchmark implements a high resolution (for example, by using the RDTSC assembly instruction on x86 ISA CPUs), portable globally synchronized clock to ensure that all processes commence the collective operation simultaneously. The timing data of all individual calls is recorded and various statistical analyses are provided during post-processing of the data. To reduce the data storage requirements, MPIBench performs on-the-fly outlier removal, that removes all samples exceeding 2 times the value of the 99<sup>th</sup> percentile.

**NBCBench** is a microbenchmark [81, 74] designed to measure the performance of both blocking and non-blocking collective operations, using a time-window based scheme to ensure process synchronization, similar to the SKaMPI benchmark. However, unlike SKaMPI the global clock synchronization algorithm that scales linearly with the number of processes  $P$ , NBCBench uses a faster algorithm that scales logarithmically with  $P$ , and was shown to be 16 times faster on a communicator of size  $P = 128$  [83].

## 4.4 Experimental Method

To evaluate the runtime performance of collective operations under load imbalance, we implemented a benchmarking methodology that is in design most similar to that of the Intel MPI benchmarks. To ensure the reproducibility of our results, we followed the guidelines laid out in [59] and benchmarked the collective operations for a range of message sizes and a large number of iterations. We measured the runtime of each process  $i$ , i.e.  $t_{\mathcal{A}}(i) = e_i - \min_{0 \leq i < P} a_i$ . For elapsed time estimation we used the MPI timing routine `MPI_Wtime`.

### 4.4.1 Microbenchmark rationale

On the PRACE CURIE supercomputer this clock (`MPI_Wtime`) was not global and the reported precision was 1  $\mu\text{s}$ . Each time, before timing the performance of a collective operation, we synchronized all processes by explicit calls to two consecutive `MPI_Barrier` routines, as is done in Intel MPI benchmark version 4.0[28]. This ensures that runtimes of collectives are measured in isolation and that the arrival times are sufficiently balanced: assuming a binomial tree broadcast algorithm (as part of barrier implementation), for the evaluated number of processes  $P = 128$ , this approach leads to approximately 18  $\mu\text{s}$  of spread in arrival times. We compute this spread from the measured latency of a control message  $L=2.66 \mu\text{s}$ , as reported by Netgauge [83]. The minimal injected absolute imbalance was 50  $\mu\text{s}$ , and the minimal imbalance increments also 50  $\mu\text{s}$ . For the smallest evaluated message size  $m = 128 \text{ KiB}$ , where the minimum collective runtime was  $\approx 200 \mu\text{s}$ , we considered this approach to be sufficiently accurate. If the broadcast operation is implemented

with hardware multicast, a feature common to Infiniband systems, then even larger communicators or smaller message sizes could be evaluated. The entire runtime estimation procedure is described by Algorithm 7.

---

**Algorithm 7:** Runtime estimation procedure for a problem size  $m$ , constant arrival pattern  $\psi = (a_0, \dots, a_{P-1})$ , a list of algorithms  $\mathcal{A}^v = \{\mathcal{A}_0, \dots, \mathcal{A}_{n-1}\}$  and a list of operators  $\star^v = \{\star_0, \dots, \star_{f-1}\}$ .

---

**Require:** Input data of size  $m$  at each process in the communicator

**Ensure:** One output file for each combining operator, with  $numIter - 1$  recorded observations per algorithm

```

L0 For each algorithm  $\mathcal{A}_i \in \mathcal{A}^v$  do
L1 For each combining operator  $\star_j \in \star^v$  do
S1 Perform a double barrier
S2 Sleep for time  $a_i$ 
S3 Start the timer (time  $t_s$ )
  S3.1 Execute the collective operation
  S3.2 Stop the timer (time  $t_f$ )
S4 Report the elapsed time  $t_i = t_f - t_s$ 
S5 Collect the reported times at the root
  S5.1 (Root) Store the time  $\max(a_i + t_i) - \min(a)$ 
S6 If the number of iterations  $k < numIter$  goto L0

```

---

Instead of using double barriers to synchronize processes in the communicator, we could have adopted a window based approach, such as the one in [177], as was discussed in Section 4.2.2.

In fact, we did so, following the approach set forth in Netgauge [83]. However, due to very long tails in collective operation runtime distributions, when subjected to PAT imbalance, we found it difficult to fix the window size, such that we can guarantee that all processes will have completed the collective operation call before the new window onset.

The solution was to either make time windows very long or to re-synchronize the processes after each iteration and re-establish the window. Both solutions were found to be relatively slow and due to limited computational resources at our disposal, we were forced to adopt the faster, if less precise barrier based benchmarking approach. Yet, for completeness, we allow our microbenchmark to be configured to either synchronize the processes with double barrier calls, or with the global clock window based scheme, by setting a compile time flag in the configuration file.

In the execution of the microbenchmark, the PAT patterns are computed at the

root and broadcast to all processes in the communicator, prior to collective operation runtime measurement. To produce imbalanced PATs we used high resolution *sleep* calls, provided by the C++11 standard library, that are preceded by two consecutive invocations of `MPI_Barrier()`. A similar approach was adopted by Parsons et al in [139].

The developed microbenchmark can be used to evaluate the performance of any MPI collective operation. The list of collective operations to benchmark is assembled at compile time, so adding a new operation requires recompilation. Ditto for combining operators. The input data type can be specified as any of the predefined MPI datatypes. All implementations of the reduction operation have to conform to the standard interface, as defined by MPI and are to be written as function templates with two type parameters: input data type and combining operator type. However, for those algorithms that require side information, using the C++11 `std::bind` function template to perform partial function application allows the user to expand the argument list of reduction operations, while still conforming to the standard interface.

One of the microbenchmarks's key features is the wide range of supported imbalance patterns. These range from singular imbalance (only one process in the communicator delayed), wherein any one process can be selected for suspension, an alternating distribution where even processes are delayed for time  $d_e$  and odd processes for time  $d_o$ , to various types of stochastically generated imbalance patterns (uniform distribution, normal distribution, gamma distribution and Bernoulli distribution). Finally, the microbenchmark can read a trace file consisting of  $N$  lines of  $P$  values, where each line represents one PAT pattern, and subject all the selected algorithms to this PAT sequence. Some algorithms, like Radix-k, can be further customized through usage of custom input files specified as command line arguments.

To use the microbenchmark, the user at a minimum has to specify the problem size, imbalance pattern, delay magnitude and the number of repetitions as command line arguments. The program is then passed to `mpirun`.

Because many communication systems establish their working state on demand, we exclude the first measurement out of the experiment data. Given the large sample size used, this has a small effect on the statistics. At each iteration, prior to performing the collective reduction operation, all processes in the communicator generate a new input buffer state using a pseudo-random number generator. Depending on input data size, this allows for the possibility that the input data resides within the cache memory of corresponding CPUs.

### 4.4.2 Experimental setup

For each combination of input parameters (message size  $m$ , and PAT pattern  $\psi$ ), the microbenchmark is invoked with separate calls to `mpirun`. Hunold et al [88] have shown in an experiment where 30 distinct calls to `mpirun` were performed, within which an MPI function was executed 1000 times, the difference in the mean of the function runtimes was found to be significant, if relatively small. They conclude that “The call to `mpirun` leads to a specific system state (software or hardware), which affects the experimental outcome. As a result, the call to `mpirun` is a significant factor that influences the mean runtime of an MPI function and therefore needs to be considered in the experimental design.”

In all experiments, for each problem size  $m$  that was evaluated, a different allocation of  $P = 128$  nodes was chosen, as produced by the batch system on the PRACE CURIE supercomputer (slurm 2.6.9-Bull.1.1).

The microbenchmark and the evaluated algorithms were written in ISO C++11 and compiled with the GNU’s Not Unix (GNU) gcc-4.6.3 compiler using `-O3`. On the PRACE CURIE supercomputer this code was linked to the BullxMPI library based on OpenMPI, version 1.2.8.2. On the Lynx cluster, this code was linked to the MVAPICH2 library, version 1.9.

## 4.5 Statistical analysis

All the experimental data in this work was obtained through measurement of stochastic (random data generation) processes, and is quantitative in nature. If the data was obtained in an experimentally sound manner, then it should be reproducible and amenable to statistical analysis and interpretation. Such data can then be said to be statistically random. We emphasize this, because statistical randomness does not imply true randomness, i.e. objective unpredictability. A numeric sequence is said to be statistically random when it contains no recognizable patterns or regularities. The sequence of numbers produced by an ideal dice or the digits of  $\pi$  meet the criteria of statistical randomness [181]. Yet, the digits of  $\pi$  can be computed by a deterministic computational process. However, what constitutes a pattern or regularity is subject to debate. According to principles of Ramsey theory, sufficiently large objects must necessarily contain a given substructure (“complete disorder is impossible”).

### 4.5.1 Well behaved data

Statistical analyses of measurement data typically place certain “requirements” on the nature of produced data, above merely that it be statistically random. In this dissertation, we follow the guidelines laid out in the National Institute of Standards

and Technology (NIST) Handbook of Statistical Methods [138], that require that the experimental data be:

1. random
2. from a fixed distribution
3. the distribution has a fixed location
4. the distribution has fixed variation

To quote [138]: “The location is the expected value of the output being measured. For a stable process, this is the value around which the process has stabilized”.

Measures of location determine the typical or central value that best describes a given data. For some stable random data generation process, this will be the value around which the process will have stabilized. A numerical measure of location is typically the sample mean, the sample median or the sample mode. Typical numerical measures of variation are the sample range or the sample variance. If the four assumptions are met, then statistical analyses can produce models that describe the behaviour of the random data generation process of the form:

$$\text{response} = \text{deterministic component} + \text{random component} \quad (4.1)$$

The simplest data generation process is univariate, and then Eq. 4.1 becomes:

$$\text{response} = \text{constant} + \text{error} \quad (4.2)$$

In the case of univariate data, the “fixed location” requirement becomes the unknown constant. Such a process is said to be operating under constant conditions.

Ideal experimental univariate data is such where the observations are uncorrelated with one another, the error component of the model has a fixed distribution, the deterministic component consists only of a constant and the error component has fixed variation.

The univariate model can easily be extended to the general case where the deterministic component is a function of several variables and the engineering objective is to devise a model of the function. If the chosen model is good, then the differences (residuals) between the raw response data and the data predicted by the model should behave like a univariate data generation process. The residuals will then satisfy the four principal requirements of good data: randomness, fixed distribution, fixed location and fixed variation.

“Moreover, if the four assumptions[requirements] are valid, then the process is amenable to the generation of valid scientific and engineering conclusions. If the four assumptions are not valid, then the process is drifting (with respect to location, variation, or distribution), unpredictable, and out of control. A simple

characterization of such processes by a location estimate, a variation estimate, or a distribution "estimate" inevitably leads to engineering conclusions that are not valid, are not supportable (scientifically or legally), and which are not repeatable in the laboratory." [138]

There are many techniques to verify that given data meets these four requirements. Some are graphical, while others are quantitative. In the discipline of Exploratory Data Analysis (EDA), 4-plots are a popular tool for quick inspection that can give key insight into the characteristics of the data. A 4-plot consists of a run sequence or time series plot, a lag plot, a histogram and a normal probability plot. Each of the tools can tell a part of the story. Let  $y = (y_0, \dots, y_k)$  be a sample consisting of  $k$  observations. Then the 4-plot can be constructed as follows:

**Run Sequence Plot** Ordinate:  $y_i$ . Abscissa:  $i$ . If the run sequence (time series) plot is roughly a horizontal line, then the fixed location requirement holds. If the spread of the data in the plot is roughly the same across the entire time range, the fixed variation requirement can be said to be met.

**Lag Plot** Ordinate  $y_i$ . Abscissa:  $y_{i-1}$ . If the lag plot has no particular structure, then the randomness requirement might hold

**Histogram** Ordinate: count of occurrences. Abscissa: buckets of  $y_i$ . If the histogram is bell-shaped, the underlying distribution of the data generation process is symmetric and potentially Gaussian

**Normal Probability Plot** Ordinate: ordered  $y_i$ . Abscissa: computed values for ordered  $y_i$  from a Gaussian distribution  $N(0, 1)$ . If the normal probability plot is roughly linear, we can claim that the underlying distribution is approximately normal (Gaussian).

A simple quantitative method to test for fixed location in the data, is to produce a linear fit of the data and examine the slope of the line. To determine whether the underlying distribution has fixed variation, Barlett's [13, 171] or Levene's test [107] is typically employed. To quantify the randomness of the data, i.e. that the observations are not mutually correlated, a Wald-Wolfowitz runs test [186] may be conducted. This test can be further reinforced with autocorrelograms.

If experimental data is not stochastically random, then all the usual statistical tests are no longer applicable. For example, the calculated uncertainties or confidence intervals for common statistics such as the mean, become meaningless. However, this does not necessarily mean that non-random data cannot be a representative sample of the population.

If the location parameter is not-fixed then the single location estimate may be meaningless (if the process is drifting), or the location estimate may be biased. The same applies for non-fixed variation.

The underlying distribution of the data has a direct impact on the quality and hence choice of estimators. The mean or average is a commonly used location estimator, but for some distributions it represents a poor choice. In fact, for any given distribution there exists an optimal location estimator, with minimum variation and noisiness. This can be the median, the trimmed mean, the midrange, etc. This suggests that one should estimate the distribution first and then select the optimal location estimator.

The Chebyshev's inequality guarantees that in any probability distribution, nearly all values are close to the mean: more formally, no more than  $1/k^2$  of the distribution's values can be more than  $k$  standard deviations away from the mean. This implies that at minimum 75% of values must lie within two standard deviations of the mean and 89% within three standard deviations. However, this theorem applies only to the true population mean and standard deviation.

If the underlying distribution is changing, the stochastic process may be unpredictable or un-modelable. Many statistical procedures, such as t-tests, linear regression analysis and Analysis of Variance (ANOVA) require as a precondition that the experimental data has an underlying normal distribution.

To that end, several methods stand available to researchers to assess whether a random sample of independent observations comes from a population with a normal distribution. Some are graphical (histograms, normality plots, Q-Q plots [189]), while others produce quantitative answers. Among the latter, the Shapiro-Wilk [167] test stands out as the most powerful as shown by Monte Carlo simulations in [156]. This test, however, can be misleading for large samples, as the percentage of p-values signalling deviation from perfect normality increases with sample size.

It is possible sometimes to normalize the data, for example by averaging intervals of length  $k$ . The Central Limit Theorem (CLT) states that with a large number of sufficiently long intervals such transformation will result with a normal distribution.

### 4.5.2 Comparison of statistical analyses among microbenchmarks

As in [88] we summarize in Table 4.3 the common MPI benchmarks found in literature, together with the principal statistics they report. A weak point of the majority of these benchmark suites is the lack of rigorous statistical analysis of collected data. One important result of such analysis would be the production of confidence limits for employed statistics. A common example are interval estimates for the mean. Hoeffler et al [77] report in the study that encompassed 95 papers from leading conferences (ACM HPDC, ACM/IEEE Supercomputing and ACM PPoPP) that only 2 papers report confidence intervals around the mean.

Confidence intervals around the mean can be derived by applying a hypothesis test that the population mean has a specific value  $\mu$ , against the alternative that it

**Table 4.3:** Overview of statistical methods applied in MPI benchmarks

Benchmark	mean	min	max	dispersion metric
mpptest [59]	min of means			<b>X</b>
SKaMPI [2]	✓			std. error
OSU	✓	✓	✓	<b>X</b>
Intel MPI [28]	✓	✓	✓	<b>X</b>
MPIBlib [105]	✓			CI of the mean (95%)
MPIBench [61]	✓	✓		sub-sampled data
mpicoscope [179]	✓	✓	✓	<b>X</b>
Phloem MPI	✓	✓	✓	<b>X</b>

does not have the value  $\mu$ . This is done by applying a one-sample Student's t-test or a z-test depending on sample size and assumptions about the population distributions. However, when the mean is not a desirable location estimator, confidence intervals tend to be mathematically difficult to derive. Then, bootstrap methods can be used to obtain confidence intervals [138].

### 4.5.3 Bootstrapping and bootstrap plots

It is very attractive to select one number that would best describe the experimental data. The most common choice being the typical or central value of the data, derived from some measure of central tendency. These are most commonly the sample mean, the sample median and the sample mode. Not all measures are equally applicable to all underlying distributions. For example, while the sample mean is the most widely used measure of location it is not a robust statistic, i.e. it is sensitive to outliers in data. Robust statistics are typically preferred in presence of data whose underlying distribution is non-normal.

Classical estimation methods are built on assumptions that are not always met in practice. In particular, it is often assumed that the error (or disturbance) is normally distributed or that the Central Limit Theorem (CLT) can be relied on to produce normally distributed sampling distributions. The Cauchy distribution is an interesting example of when the latter does not apply. It is a symmetric distribution with heavy tails and a single peak at the center. It is peculiar that when data is distributed according to a Cauchy distribution, collecting more data does not provide a more accurate estimate of the mean. More precisely, the sampling distribution of the mean of  $n$  samples from a Cauchy distribution has the same distribution as the original Cauchy distribution, regardless of  $n$ . This means that for the Cauchy distribution the mean is useless as a measure of the typical value.

Tukey and Mosteller [133] defined two types of robustness, where robustness is defined as a lack of susceptibility to the effects of non-normality.

- Robustness of validity (tolerance to non-normal tails) states that the  $p$  confidence intervals for the population location have a  $p$  chance of covering the population location whatever the sampled population might have been.
- Robustness of efficiency refers to high effectiveness in the face of non-normal tails. It is exemplified by confidence intervals for the population location which tend to be almost as narrow as the best that could be done if we knew the true shape of the distribution.

The sample mean is an example of an estimator that is the best we can do if the underlying distribution is normal. However, it lacks robustness of validity. That is, confidence intervals based on the mean tend not to be precise if the underlying distribution is non-normal. The median is an example of an estimator that tends to have robustness of validity but not robustness of efficiency.

Bootstrap plots are a useful tool in deciding which measure of location (or some other statistic) is the better choice for some set of data. Bootstrap uncertainty estimates of statistics for some set of data, are generated by computing the same statistic for a subsample of equal (or smaller) size of the original set. The subsampling is performed with replacement, i.e. the same data point may be sampled multiple times. Repeating this computation a large number of times will result with an estimate of the sampling distribution of the statistic.

For example, to estimate the uncertainty of the sample mean from a dataset of 100 elements, we generate a subsample of 100 elements and compute the mean. The process is repeated 1000 times, so that we have 1000 values for the mean. To calculate a 95% confidence interval for the mean, the sample means are sorted into ascending order. Then the value of the 25th mean is chosen as the lower confidence limit and the value of 975th element as the upper confidence limit.

Bootstrap plots can be used to ascertain which statistic has a sampling distribution with the smallest variance for a given data set, i.e. which statistic produces the narrowest confidence interval.

#### 4.5.4 Permutation tests

Significance tests are typically used to determine whether an observed effect, such as the difference between two sample means, could be reasonably ascribed to randomness introduced in selecting the sample. If the effect cannot be ascribed to randomness, then we can establish a strong argument that the effect observed in the sample is representative of the population itself. A typical significance test can be conducted as follows [66]:

- Select a statistic of interest

- Compute/construct the sampling distribution of the statistic, if the effect were not present in the population
- Locate the observed value of the statistic in the computed distribution. If the value is located in tail, then this is evidence that something other than random chance is responsible for the observation.

More formally, we establish the *null hypothesis* so that it states that the effect is not present in the population. Then, the computed probability that the null hypothesis is true, is the probability that we would observe a statistic value as extreme or more extreme than the one we did observe. This probability is the P-value.

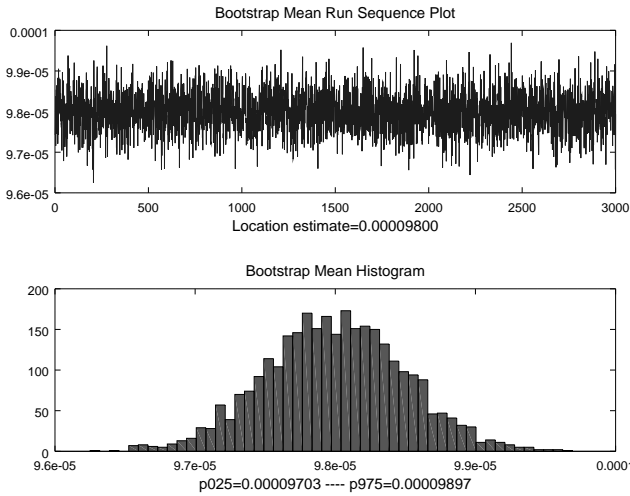
The resampling in permutation tests is performed without replacement, hence the name permutation. If, for example, we have 100 samples from population  $A$  and 100 samples from population  $B$  and we would like to use a permutation test to establish that the difference in means  $\mu(A) - \mu(B)$  is statistically significant, we would proceed as follows.

- Select at random out of the total 200 samples, 100 samples that form group  $A'$ . The remaining 100 samples will form group  $B'$ .
- Compute the difference of means  $\mu(A') - \mu(B')$ .
- Repeat the resampling procedure 1000 times. Make sure that each time a new and unique permutation has been generated. The computed distribution of the statistic  $\mu(A') - \mu(B')$  constitutes the sampling distribution under the condition that the null hypothesis is true.

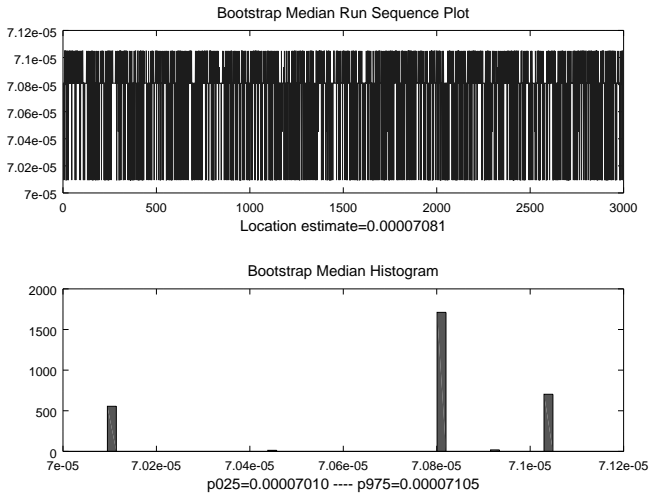
The one sided P-value of the test is calculated as the proportion of sampled permutations where the difference  $\mu(A') - \mu(B')$  was greater than the observed difference  $\mu(A) - \mu(B)$ .

The principal difference between bootstrapping and permutation tests is that bootstrapping requires sampling with replacement, while permutation tests sample without replacement. It is important to observe that in either case the time order of observations in data is lost.

To conclude, permutation tests are best used to test a null hypothesis that only random sampling/randomization explains the observed difference in statistics. In contrast, bootstraps are best to derive confidence intervals for some statistics, or to compare statistics for bias and variance.



(a) bootstrap of mean,  $m = 40\text{kB}$



(b) bootstrap of median,  $m = 40\text{kB}$

**Figure 4.2:** Bootstrap plots for the mean and median of observed message transfer times for  $m \in \{40\text{kB}\}$ . A sample of  $128 \times 128$  p2p message transfer observations was resampled for  $B = 3000$  times to produce the plots. All measurements were performed on the PRACE CURIE supercomputer;  $P=128$ , one process per node

## 4.6 Selecting the most appropriate measure of location

The multimodal nature of the underlying distribution of p2p message transmission time investigated in Section 4.3.2 invites the question of what the most appropriate measure of location would be. To answer that question, we performed bootstrap Monte Carlo simulations for two statistics: mean and median. For each message size  $m \in \{40\text{kB}, 200\text{kB}, 400\text{kB}, 2\text{MB}, 4\text{MB}\}$ , we took a single sample of  $128 * 128 - 128 = 16256$  of p2p message transfer observations gather on a single allocation of  $P = 128$  nodes of the PRACE CURIE supercomputer and performed  $B = 3000$  resampling of each sample. The results of this simulation are depicted in Figure 4.2 and Appendix ???. The details of how the bootstrap simulations were conducted is explained in more detail in Section 4.5.3.

Examining the computed sampling distributions of the mean and the median, we can observe that the location of the median is consistently smaller than that of the mean (by roughly 10%). This is a consequence of a positive skew in the distribution of the underlying data. Examining the width of confidence intervals, we can state that in all cases, except when  $m = 4\text{MB}$ , the median has narrower confidence intervals and thus higher robustness of efficiency (Section 4.5.3). For  $m = 4\text{MB}$  both statistics have confidence intervals of roughly the same width.

## 4.7 Summary

This chapter has provided a detailed insight into the benchmarking philosophy underpinning modern microbenchmarks of MPI p2p and collective operations. Its primary purpose was to introduce the experimental and statistical methodology utilized in the performance evaluation of reduction algorithms in Chapter 5 and Chapter 6.

The contributions of this chapter are of technical nature, and chiefly revolve around the novel microbenchmarking tool developed for performance evaluation of collective operations subjected to various patterns of imbalanced PATs. The imbalance patterns are specified as command line arguments to the tool, while the list and implementation of the reduction algorithms has to be provided in the form of C++ source code to be compiled before conducting benchmarking sessions. The tool records all the measured observations of algorithm runtimes for a posteriori statistical analyses.

In addition to this, an algorithm for measuring p2p message transmission and combination times, constructed in the pattern of a binomial tree was defined. This microbenchmark was run on a set of  $P = 128$  nodes, allocated on the PRACE CURIE supercomputer to collect a detailed communication profile. The

subsequent analysis of the experimental data has revealed an underlying multimodal distribution, an artifact of the static routing used by the Infiniband fat-tree network. This non-homogeneity violates one of the assumptions in Definition 2.5.1. In the subsequent chapters we will investigate in more detail how this affects the algorithm runtime predictions based on Definition 2.5.1.

The chapter is concluded with an overview of the statistical methods utilized in a posteriori data analysis. The primary purpose thereof was to ensure the soundness of experimental data, decide on the optimal measures of location used to report algorithm performance and to establish statistical confidence levels on reported results. This concludes the foundation work on the experimental methodology employed in Chapter 5 and Chapter 6.

## Chapter 5

# Optimal reduction algorithm for atomic messages

---

In this chapter, we address the problem of global reduction for atomic input data. We present two new algorithms. The first algorithm is *clairvoyant* and *static*. Clairvoyancy implies that the algorithm is endowed with side-information: i.e., that it has full knowledge of the Process Arrival Times (PATs) at the reduction operation invocation point and of the time required to complete Point-to-point (p2p) message transfer and message combination. The staticity implies that the algorithm follows a static schedule, one that is not adaptable during algorithm runtime. With the side-information as its input, the clairvoyant algorithm pre-constructs a reduction schedule, i.e. a directed graph defining the order in which processes send/receive and combine their subresults. In this chapter we prove that this schedule is of optimal length for all PAT patterns. Furthermore, we quantify the maximum theoretically attainable speedup compared to that of Binomial Tree algorithm as a function of communicator size.

The second algorithm is non-clairvoyant (agnostic) and dynamic. The dynamicity of the algorithm's reduction schedule makes it more robust to circumstances where the PATs are unpredictable or where the communication network exhibits a high degree of heterogeneity.

Both of the algorithms were implemented in terms of Message Passing Interface (MPI) p2p communication primitives. Their performance was experimentally validated on a 128 node subset of the PRACE CURIE supercomputer and compared with that of Binomial Tree reduction. The experimental results showed that both of our implementations dominate the Binomial Tree reduction algorithm in presence of imbalanced PATs. The clairvoyant reduction algorithm was shown to produce reduction schedules whose runtime is near identical to that of the Binomial Tree algorithm, whenever the PATs were balanced. This result was expected, as the Binomial Tree algorithm is known to be the optimal algorithm for balanced PATs

(and atomic input data).

## 5.1 Optimal reduction algorithm for atomic messages

Let *atomic P-way reduction* be defined as in Definition 2.5.2 (Chapter 2) and let the problem size be  $m$ . Then let  $\mathcal{A}$  be an algorithm for *atomic P-way reduction*. Let  $t_{\mathcal{A}}(i)$  and  $t_{\mathcal{A}}$  be the normalized run time of process  $i$ , and the running time of algorithm  $\mathcal{A}$ , respectively as defined in Chapter 2.

As laid out by Definition 2.5.1, we assume that the network has no capability to overlap communication with computation (i.e. no offload). Furthermore, for simplicity, we assume that the system does not buffer small messages, and that a single communication protocol is used in communication, analogous to synchronous send (where a process cannot begin sending a message before the destination posts a matching receive). This implies that the time to send a message will be equal to the time to receive a message. In implementations of reduction algorithms, it is the receiving process that will combine the message with its local data. This will constitute a complete round in the execution of a reduction algorithms. In algorithm runtime complexity analysis, the runtime is typically expressed in the number of rounds required to solve the reduction problem. Using the linear model, the cost of one round where a message of size  $m$  is transmitted and combined is expressed as:

$$d(m) = \alpha + m\beta + m\gamma \tag{5.1}$$

All the algorithm analyzed in this chapter will communicate messages of equal size: thus we will write  $d$  to denote the time to transmit and combine one message, or in other words, the time to complete one round of reduction. This is then essentially the *postal model*, introduced in [11]. Similar linear models, all of which assumed network homogeneity, were used in [178, 153, 24, 22].

**Definition 5.1.1.** If  $d$  is the number of time steps to send and combine one message of size  $m$ , the complexity of atomic P-way reduction for problem size  $m$  and a PAT  $\psi$  is defined as:

$$R(P; d, \psi) = \min_{\mathcal{A}} t_{\mathcal{A}}(\psi)$$

$\mathcal{A}$  is optimal for atomic P-way reduction if  $\forall \psi, t_{\mathcal{A}}(\psi) = R(P; d, \psi)$ .

We will now develop an algorithm for *atomic P-way reduction* that takes  $R(P; d, \psi)$  time steps. The proposed algorithm is intuitive: all ready processes combine their messages as soon as they can until all messages have been combined. The algorithm sends no redundant messages and no process waits to combine a message, unless forced to. We make this idea more precise below.

**Definition 5.1.2.** A reduction algorithm is greedy if

- When two processes combine their messages, the process that sends its message exits upon completion of the transfer
- If at least two processes are ready at time step  $t$  then they immediately combine their messages, i.e. no process waits (idles) if it has choice not to.

The first property of the greedy algorithm is straightforward: once a process sends its message there is no reason for it to stay active other than dispatch messages from one process to the other. However, in a fully connected homogeneous network this is strictly slower than having the two processes communicate directly. The second property merits more attention.

**Definition 5.1.3.** Let  $\mathcal{G}$  be a greedy reduction algorithm for *atomic  $P$ -way reduction*. Let  $G = \{i_1, \dots, i_k\}, k = P - 1$  be the processes whose messages have been combined in algorithm  $\mathcal{G}$ , ordered by time when their messages were combined. Then  $t_{\mathcal{G}} = t_{\mathcal{G}}(i_k)$ , i.e. the time at which the root process has received and combined the last message. Let  $O = \{j_1, \dots, j_k\}$  be an optimal solution generated by algorithm  $\mathcal{O}$ , i.e.  $t_{\mathcal{O}}(j_k) = R(P; d, \psi)$ .

**Lemma 5.1.4.** *If  $\psi$  is a balanced PAT  $(0, \dots, 0)$  then  $R(P; d, \psi) = \lceil \log_2 P \rceil d$ . Moreover,  $t_{\mathcal{G}} = R(P; d, \psi)$ .*

*Proof.* In a single port network, the number of outstanding messages to combine can at most be halved per round. It follows that the number of required rounds for the last message to reach the root is  $n = \lceil \log_2 P \rceil$ . From this it immediately follows that  $R(P, d, \psi) = nd$ . By definition, greedy algorithm combines messages as soon as the respective processes are ready, implying that it will take  $n$  rounds to execute.  $\square$

The problem of finding an algorithm that reduces  $P$  messages in minimum time, can without loss of generality be rephrased into one of finding an algorithm that reduces most messages at any given time step.

**Definition 5.1.5.** ( *$t$ -step reduction*)

Given  $t \geq 0$ , the number of time steps available, let

$$P^*(t; d, \psi) = \max\{i : R(i; d, \psi) \leq t\}$$

be the maximum number of messages that can be combined in  $t$  time steps. We can also think of  $P^*(t; d, \psi)$  as the maximum number of processes whose messages can be combined in  $t$  time steps. If  $\mathcal{G}$  is a greedy reduction algorithm, then let  $P(t; d, \psi)$  be the size of set  $G$  at time  $t$ , i.e. the number of messages that  $\mathcal{G}$  has combined by time step  $t$ .

We will now show that the greedy reduction algorithm never falls behind some optimal algorithm. For simplicity, we will write  $P^*(t)$  and  $P(t)$  instead of  $P^*(t; d, \psi)$  and  $P(t; d, \psi)$  respectively.

The proof will follow from the observation that at any point in the execution of a reduction algorithm, waiting for some time  $x$  and then combining is never better than immediately combining whenever two processes are ready.

We will first demonstrate that waiting for time  $t' < d$  to collect more ready processes, will not result in more processes combined at time  $t' + d$ , compared to the greedy strategy:

**Lemma 5.1.6.** *Let  $r(t) = 2k_1 + r_1$  be the number of ready processes at time  $t$ . Let  $\delta = 2k_2 + r_2$  be the number of additional ready processes at time  $t' = t + x$ ,  $x < d$ . Then waiting till time  $t'$  to combine all the ready processes will not result with more processes combined compared to greedily combining them at time  $t$  and  $t'$ .*

*Proof.* Waiting till time  $t'$ , and the combining, will result with

$$\frac{2k_1 + r_1 + 2k_2 + r_2}{2} = k_1 + k_2 + \lfloor \frac{r_1 + r_2}{2} \rfloor$$

processes combined, compared with

$$k_1 + \frac{2k_2 + r_1 + r_2}{2} = k_1 + k_2 + \lfloor \frac{r_1 + r_2}{2} \rfloor$$

if the processes were greedily combined. □

We now demonstrate that that algorithm  $\mathcal{G}$  is optimal for  $t$ -step reduction.

**Lemma 5.1.7.**

$$P^*(t) \leq P(t), 0 \leq t \leq R(P)$$

*Proof.* We prove by induction on  $t$ , with a step of size  $d$ .

For  $0 \leq t < d$  it is clear that  $P(t) = P^*(t) = 0$  because the first message can be combined only at time  $t \geq d$ .

Assume that  $P^*(t) \leq P(t)$ , for some time step  $t \geq 0$ . We will show that the statement holds for  $t + d$ .

If  $P^*(t) = P(t)$  then either the number of ready processes  $r^*(t) = r(t)$  or  $r^*(t) > r(t)$ , from the definition of  $\mathcal{G}$ . If the former, then combining greedily (compared to waiting) is strictly better:  $P^*(t + d) \leq P(t + d)$ . If the latter, then it must be the case that algorithm  $\mathcal{O}$  forewent combining at least one pair of processes in the time period  $[t - d, t)$ . However, from Lemma 5.1.6 it follows that if both  $\mathcal{O}$  and  $\mathcal{G}$  combine all ready processes at time  $t$ ,  $P^*(t + d) \leq P(t + d)$ .

If  $P^*(t) < P(t)$  then it must be the case that algorithm  $\mathcal{O}$  decided not to combine at least one pair of ready processes at some time point  $t' \leq t - d$ . Since the total

number of online processes at time  $t$  is the same for both algorithms, the following equality holds:

$$P(t) + o(t) = P^*(t) + o^*(t) \quad (5.2)$$

where  $o(t)$  and  $o(t)^*$  are the number of outstanding processes at time  $t$  (processes that are either ready or exchanging data). At time step  $t + d$  algorithm  $\mathcal{G}$  will have combined a total of  $P(t) + \lfloor \frac{o(t)}{2} \rfloor$  processes compared to  $P^*(t) + \lfloor \frac{o^*(t)}{2} \rfloor$  for algorithm  $\mathcal{O}$ . Then, from Eq. 5.2 it follows that  $P^*(t + d) \leq P(t + d)$ .  $\square$

Thus, at time step  $t = R(P)$ ,  $P(t) = P - 1 \Rightarrow t_{\mathcal{G}} = R(P)$

We have proven the following theorem:

**Theorem 5.1.8.** *Let  $\mathcal{G}$  be a greedy reduction algorithm.  $\mathcal{G}$  is optimal for atomic  $P$ -way reduction.*

**Conjecture 5.1.9.** *Since reduction is essentially an inverse operation to broadcast, the greedy reduction algorithm can be trivially modified to derive an optimal broadcast algorithm by changing  $d$  to be the time to send one message.*

Some of the results derived in this section share parallels with the work of Canon and Antoniu [111] wherein the authors presented a clairvoyant reduction algorithm for atomic data, based on the assumption of homogeneity of network transmission and processor computing power. While their work does not take imbalanced PATs into account, they do consider overlap of communication with computation in the derivation of optimal reduction schedules.

### 5.1.1 Absorption potential

If absorption time (Definition 3.3.7) of some reduction algorithm  $\mathcal{A}$  is equal to the absolute imbalance, then  $\mathcal{A}$  will not exhibit any slowdown due to imbalance in process arrival times. However, there exists an upper bound on the absorption time.

**Proposition 5.1.10.**

$$A(\psi, \mathcal{A}) \leq t_{\mathcal{A}}(\pi) - d,$$

where  $d$  is the time to transmit and combine one message of size  $m$ , and  $\pi$  is a balanced PAT.

*Proof.* The largest absorption time will be achieved if by the time the last process (let that be process  $i$ ) has become ready only the message  $m_i$  remains to be combined to derive the final result. The optimal time to reduce two messages with two processes is  $d$ .  $\square$

## 5.2 Atomic message reduction algorithms

In this section we describe three algorithms: binomial tree reduction, the implementation of the greedy reduction algorithm and a dynamic reduction algorithm. Of the three algorithms, two are new and discussed in depth. In the discussion that follows, it is assumed that the root lies at process rank 0.

### 5.2.1 Binomial Tree

This is the optimal reduction algorithm for balanced PATs, under the assumption of atomic messages and a fully connected full-duplex no overlap network. The algorithm is found in implementations of many MPI libraries, serving in various forms as implementation of `MPI_Reduce`, `MPI_Gather` and `MPI_Broadcast`. This algorithm was defined in Chapter 2 as Algorithm 1.

**Proposition 5.2.1.** *Let  $\mathcal{B}$  be the Binomial Tree algorithm and  $P = 2^x, x \in \mathbb{N}$ . Then  $\forall \psi, A(\psi, \mathcal{B}) = 0$ .*

*Proof.* By Definition 3.3.7,  $A(\psi, \mathcal{B}) = t_{\mathcal{B}}(\pi) - t_{\psi} + I(\psi)$ . Under the assumption that  $P$  is a power-of-two, the idle time of each process, i.e. the time it waits to either send or receive a message in the algorithm is zero. Should some process  $i$  be delayed for time  $I(\psi)$ , then in the critical path of processes dependent on  $i$  every process will be delayed by time  $I(\psi)$ . As this path extends to the root, the root process too will be delayed by time  $I(\psi)$ , implying that  $t_{\mathcal{B}}(\psi) = t_{\mathcal{B}}(\pi) + I(\psi)$ .  $\square$

For non-power-of-two communicator sizes, the Binomial Tree reduction schedule is not fully balanced and there is sufficient slack to absorb some of the imbalance. This is because the number of rounds of execution is a function of  $\lceil \log_2 P \rceil$ , where  $P$  is the communicator size. For PATs  $\psi$  where any one process is delayed by time  $\delta$ , i.e.  $I(\psi) = \delta$ , the Binomial Tree algorithm could absorb anywhere from 0 to  $d(\lceil \log_2 P \rceil - 1)$  of absolute imbalance  $\delta$ , depending on the communicator size and the position of imbalance. We will illustrate the case where we might expect the upper absorption limit of  $d(\lceil \log_2 P \rceil - 1)$ .

The upper absorption limit could be achieved for communicator sizes of  $P = 2^x + 1, x \in \mathbb{N}$ , if the delay occurs at process rank  $P - 1$ . Then the algorithm will proceed for  $R = \lceil \log_2(P - 1) \rceil$  rounds, regardless of whether process rank  $P - 1$  is ready or delayed. The minimum number of rounds for algorithm Binomial Tree to complete  $P$ -way reduction on a communicator of size  $P$  is  $R + 1$ . Should the delay befall some other process rank, then the static nature of the reduction schedule will result in zero imbalance absorption.

As was previously stated, the algorithms in this work will be evaluated only for power-of-two communicator sizes, where we expect the Binomial Tree algorithm to

have zero absorption capacity. This is a reasonable choice, as most High Performance Computing (HPC) applications are executed with such communicator sizes. One reason for this is that some non-atomic reduction algorithms (like Butterfly [152]), perform well only for power-of-two size communicators.

### 5.2.2 Clairvoyant Reduction

This algorithm is an implementation of algorithm  $\mathcal{G}$  defined in Section 5.1. The function interface is expanded so that  $d$ , the time to receive and combine a message of size  $m$ , and  $\psi$  the current PAT at the invocation time, are accepted as parameters.

The algorithm works as follows. Processes  $0..P-1$  are inserted into a priority queue, with greatest priority assigned to processes with the smallest arrival times. In the first step, the first two processes in this ordering  $a_i, a_j$  are popped from the queue and combined so that process  $i$  receives from process  $j$ . Process  $j$  is eliminated and process  $i$ 's ready time is updated to the new time  $a_j+d$ , after which it is inserted back into the queue. The algorithm repeats, always selecting the top two processes in the queue and combining them, until only the root process remains. Care is taken that during the combining the root process is the one returned to the queue. The result is a schedule tree, which the algorithm then executes till completion.

We define the algorithm in pseudo code as Algorithm 8.

---

#### Algorithm 8: Clairvoyant

---

**Require:**  $root=0$

**Ensure:**  $R = m_0 + m_1 + \dots + m_{P-1}$

- S1 Insert processes  $0..P-1$  into a priority queue  $Q$ , where the greatest priority is assigned to processes with the smallest arrival times.
  - S2 Pop the top two processes  $i$  and  $j$  from  $Q$ , with  $a_i \leq a_j$ .
    - S2.1. If  $j$  is root swap( $i, j$ ).
  - S3 Set process  $i$  to be the parent of  $j$ .
  - S4 Add process  $j$  to the queue of process  $i$ 's children:  $C_i+ = j$ .
  - S5 Update process  $i$ 's arrival time to be  $a_i+ = d$ .
    - S5.1. Insert process  $i$  back into  $Q$ .
  - S6 If  $size(Q) > 1$  goto 2
  - E1 Process rank  $r$ . Execute the constructed schedule tree
    - E1.1. Receive and combine in order data messages from  $C_r$ .
    - E1.2. Send data message to established parent.
-

**Theorem 5.2.2.** *Let  $\mathcal{B}$  and  $\mathcal{C}$  be Binomial Tree and Clairvoyant reduction algorithms, respectively. Assume that  $t_{\mathcal{B}}(\pi) = t_{\mathcal{C}}(\pi)$  for balanced PAT  $\pi = (0, \dots, 0)$  and  $P = 2^x, x \in \mathbb{N}$ . Let the speedup  $s$  for some PAT  $\psi$  be  $s(\psi) = \frac{t_{\mathcal{B}}(\psi)}{t_{\mathcal{C}}(\psi)}$ . Then, the maximum speedup  $S = \max_{\psi} \left( \frac{t_{\mathcal{B}}(\psi)}{t_{\mathcal{C}}(\psi)} \right) = \frac{2}{1 + \frac{1}{\lceil \log_2 P \rceil}}$ , when  $I(\psi) = t_{\mathcal{B}}(\pi)$ .*

*Proof.* Let  $\psi'$  be a PAT where only one process is delayed. Then out of all PATs  $\psi, I(\psi) = I(\psi') = \delta, \bar{a}(\psi')$  is minimal. Let  $d = \frac{t_{\mathcal{B}}(\pi)}{\lceil \log_2 P \rceil}$  be the time to perform one round of reduction, i.e. the time to combine two messages. If  $I(\psi') < d$  then  $t_{\mathcal{C}}(\psi') = t_{\mathcal{C}}(\pi) + I(\psi')$ , i.e.  $A(\psi', \mathcal{C}) = 0$ . This is because according to the Clairvoyant algorithm all but  $P - 2$  processes will combine immediately with the other two trailing by time  $I(\psi')$ . As  $I(\psi') < d$ , no other ordering is possible and the reduction will proceed identical to Binomial Tree, implying  $s(\psi) = 1$ . If  $I(\psi') \geq d$  then  $t_{\mathcal{C}}(\psi') = t_{\mathcal{C}}(\pi) + d + \max\{I(\psi') - t_{\mathcal{C}}(\pi), 0\}$ . In this case, we can imagine the algorithm proceeding as if there are  $P - 1$  ready processes and combining the  $P$ -th process once all the  $P - 1$  have been combined. Under the assumption that  $P$  is a power-of-two,  $t_{\mathcal{C}}(P - 1, \pi) = t_{\mathcal{C}}(P, \pi) = t_{\mathcal{B}}(\pi)$ . Thus,  $s(\psi') = \frac{t_{\mathcal{B}} + I(\psi')}{t_{\mathcal{B}} + d + \max\{I(\psi') - t_{\mathcal{B}}(\pi), 0\}}$ . This fraction will be maximal when  $I(\psi') = t_{\mathcal{B}}(\pi) = \lceil \log_2 P \rceil d$ , i.e.

$$S = \frac{2 \lceil \log_2 P \rceil d}{\lceil \log_2 P \rceil d + d} = \frac{2}{1 + \frac{1}{\lceil \log_2 P \rceil}}$$

For any other PAT  $\psi$ , where more than one process is delayed  $t_{\mathcal{C}}(\psi) \geq t_{\mathcal{C}}(\psi')$ , as there is less slack to absorb the imbalance.  $\square$

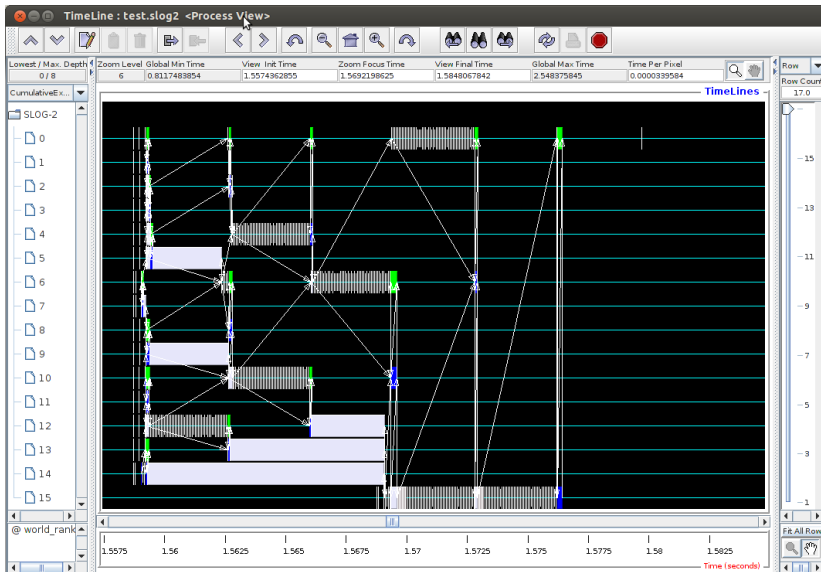
Thus, the maximum attainable speedup converges to 2 as  $P$  goes to infinity. In our experiment, the communicator size will be set to  $P = 128$ , so we will expect to observe a maximum speedup of 1.75, when  $I(\psi)$  is roughly equal to  $t_{\mathcal{B}}(\pi)$ . In Section, 6.6 we use the result of this theorem as a yardstick in the analysis of observed runtimes of algorithm Clairvoyant for the PAT where a single process is delayed.

**Corollary 5.2.3.** *Assuming  $t_{\mathcal{C}} = t_{\mathcal{B}}$ , and the PAT  $\psi$  defined in Theorem 5.2.2, the maximum absorption time of algorithm Clairvoyant  $A(\psi, \mathcal{C}) = t_{\mathcal{C}}(\pi) - d = t_{\mathcal{B}}(\pi) - d$ .*

### 5.2.3 Local Redirect

In this section we elaborate an algorithm that dynamically assigns communication peers and was first described in [114] under the name of Local Reduce. We define *Local Redirect* as Algorithm 9.

In this algorithm, each process acquires only minimal information about the PAT: namely just who of its immediate two neighbours is the first ready. In contrast to the previous algorithm, Local Redirect does not enjoy the luxury of clairvoyance.



**Figure 5.1:** Jumpshot view of a MPE trace of algorithm Local Redirect with one suspended process (Lynx, P=16). In the visualization, black color represents time spent outside MPI library functions, while white color represents time spent in MPI.Wait. Green and blue color represent time spent in MPI.Recv and MPI.Send, respectively.

---

**Algorithm 9:** Local Redirect

---

**Require:**

**Ensure:**  $R = m_0 + m_1 + \dots + m_{P-1}$

- S1 Initialize left and right neighbours according to a predefined linear ordering of processes in the communicator. The processes at the head or tail do not have a left or right neighbour respectively, i.e.  $left=-1$  and  $right=-1$ , respectively. Data is received from a left neighbour and sent to a right neighbour. Processes are redirected to the right. Rightmost neighbour is the root.
- S2 Send `MSG_DONE` message to right neighbour.
- S3 Wait for incoming messages.
- S3.1 On receipt of `MSG_DONE` message from left neighbour, send to source `MSG_OK`. Goto S3.
- S3.2 On receipt of `MSG_OK` message from right neighbor, send to source `MSG_ACK`. Initiate *handshake*, i.e. Goto H.
- S3.3 On receipt of `MSG_REDIRECT` message from right neighbour, change *right* according to message. Send `MSG_DONE` to source. Goto S3.
- S3.4 On receipt of `MSG_ACK` message from left neighbour, complete *handshake* by sending `MSG_SEND` to source. Goto R.
- H Wait for incoming messages and complete handshake
- H1 On receipt of `MSG_REDIRECT` from right neighbour, change *right* according to message. Send `MSG_DONE` to source. Goto S3.
- H2 On receipt of `MSG_SEND` from right neighbour, send data to destination. Send rank of your left neighbour to destination. Goto W.
- R Receive data
- R1 Receive data from source of `MSG_ACK` in S3.4.
- R2 Receive from the source your new left neighbour's rank (`MSG_LEFT`):  $left = new\_left$ . If  $rank = root \wedge left = -1$  exit, else goto S3.
- W If  $left \neq -1$  wait for any message from left neighbour. Once received send `MSG_REDIRECT` with  $value=right$ . Exit.
-

Instead, to acquire information about the PAT, each process in the algorithm has to pay a price: the time to send or receive one control message. Acquiring the information on global PATs would require  $P$  messages per process, resulting in a significant overhead. Local Redirect approaches the problem from the minimal cost side of the equation: with the minimal local information attempt to perform the best (re)scheduling of the global reduction.

The algorithm then proceeds to dynamically construct a schedule tree according to the idea of a greedy reduction algorithm: each process sends a message only once and to the first neighbour ready, so as to minimize its idle time. For example, in Fig. 5.2 processes  $P2$  and  $P3$  are the first to arrive. Process  $P2$  sends a completion message (Step S2, Algorithm 9) to  $P3$ , while  $P3$  sends a completion message to  $P4$ . However, as  $P4$  has not still arrived, the two processes start and complete a handshake (Step S3.1, Algorithm 9), upon which  $P2$  sends its message to  $P3$ .

The handshake is essential to synchronize processes before data transfer, due to possible time races in the arrival of completion messages. In the implementation of the algorithm, the handshake cost is equal to three control messages. Once a process has received a `MSG_DONE` control message from its left neighbour, it commences the handshake by first sending a `MSG_OK` control message back to the source of `MSG_DONE`. Then, it continues waiting for either a `MSG_DONE` from a different source or a `MSG_ACK` control message. If the received message is `MSG_ACK`, the process stops waiting and sends a `MSG_SEND` control message to the source of `MSG_ACK` to complete the handshake and begin receiving the data.

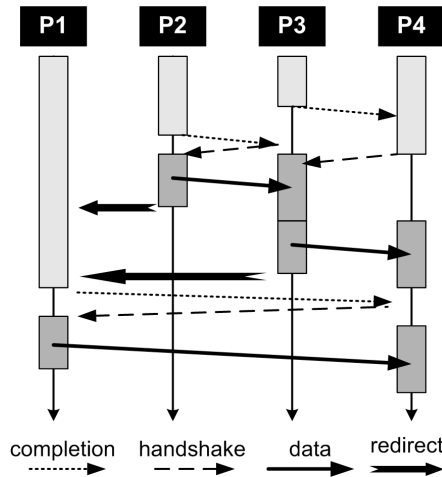
Similarly, the process to send data, after sending a `MSG_DONE` control message waits for either a `MSG_OK` control message or a `MSG_REDIRECT` control message. If the message received is `MSG_OK`, it starts the handshake by sending a `MSG_ACK` back to the source of `MSG_OK`. Upon that, it waits for either a `MSG_REDIRECT` or `MSG_SEND` control message. If the message received is the latter, it completes the handshake and starts sending the data. A Jumpshot visualization of the recorded execution of the algorithm for a communicator of size  $P = 16$  is displayed in Fig. 5.1.

## 5.2.4 Correctness of Local Redirect

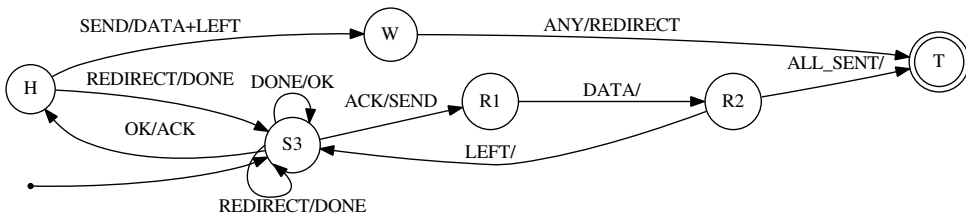
We will now demonstrate a proof of correctness for algorithm Local Redirect. To assist in following the proof we provide an abstract finite state automata (Fig. 5.3 that illustrates the states and transitions of Algorithm 9.

Assuming that all processes have contributed their input data, then from the commutativity and associativity of the combining operator  $+$  it follows that:

$$m = \sum_{i=1}^{P-1} m_{p(i)} \quad (5.3)$$



**Figure 5.2:** Illustration of algorithm Local Redirect execution for  $P=4$  and root at rightmost process. Light gray denotes time spent before the onset of the reduction, while dark gray denotes time spent in communication and computation of messages.



**Figure 5.3:** Finite state automata for Algorithm 9. Algorithm flow starts from left and terminates at state T.

for any permutation of process ranks  $p(i) : \{0 \dots P-1\} \Rightarrow \{0 \dots P-1\}$ . Thus, regardless of the PAT pattern, as long as all processes contribute their input data, algorithm Local Redirect will compute a correct result.

It remains to demonstrate that every process in the communicator will contribute its data and exit the operation, and that in the process the algorithm will not deadlock. We will show this for process rank  $x : 0 \leq x \leq P-1$ .

When  $x$  starts the algorithm, it sends `MSG_DONE` to its right neighbor (we will call it  $R$ ) and reaches state  $S3$ . A process in state  $S3$  will either attempt a handshake (Case 1) or get redirected (Case 2). We will first examine the former case. Process  $x$  can start the handshake by either receiving `MSG_OK` (Case 1.1) from its right neighbor or `MSG_DONE` (Case 1.2) from its left neighbor.

1.1 Upon receiving `MSG_OK` process  $x$  will reply with `MSG_ACK` and transition to state  $H$ . In this state, process  $x$  can only proceed by receiving a message from  $R$ .

Now, process  $R$  may or may not receive `MSG_ACK` from  $x$  before handshaking with its right neighbor  $R'$ . If the latter, then process  $R$  will also reach state  $H$  where it can only proceed by receiving a message from  $R'$ . This process can conceivably recur resulting with a sequence of processes  $x, R, R', \dots, R^*$  where each process is waiting for its right neighbour to respond before proceeding. However, this sequence can at furthest extend to the root (the rightmost process) which by definition (S1) has no right neighbor, and will respond with `MSG_SEND` to its left neighbor ( $R^*$ ), ensuring its progress.  $R^*$  will then send its data to the root (plus the rank of its left neighbor) and reach state  $W$ . In this state, it will wait for a message from its left neighbor. Upon receiving it, process  $R^*$  will send `MSG_REDIRECT` to its left neighbor  $L^*$  and exit. This will in turn allow process  $L^*$  to progress from state  $H$  to state  $S$ . In this way, all the processes back to  $x$  will eventually progress, ensuring that a deadlock does not occur. Process  $x$  will either send its data to  $R$ , or be redirected to one of the processes  $R', \dots, L^*$ . Alternatively, process  $R$  receives `MSG_ACK` from  $x$ , replies with `MSG_SEND` and transitions to state  $R1$ . This is, however, a straightforward scenario: process  $x$  will send its data and reach state  $W$ , eventually redirect its left neighbor to  $R$  and exit. The continuous left-to-right order of processes is preserved.

1.2 Upon receiving `MSG_DONE` from its left neighbor, process  $x$  will reply with `MSG_OK` and remain in state  $S3$ . From here it can either goto state  $H$ , like in the previous scenario, or to state  $R1$ , after receiving `MSG_ACK` from its left neighbor  $L$  and replying with `MSG_SEND`. While in state  $R1$ , process  $x$  cannot proceed until receiving data from  $L$ . It will ignore all other messages. Progress is ensured by the observation that process  $L$  has to be in state  $H$ , where it only listens to messages from  $x$ . Process  $L$  will then receive `MSG_DONE` from  $x$  and the two will exchange data.  $L$  will redirect its left neighbor to  $x$  and  $x$  will transition to state  $S3$ . Process  $x$  is again in state  $S3$  and the continuous left-to-right order of processes is preserved.

2. Finally,  $x$  can instead receive `MSG_REDIRECT`. In that case, it will reply with `MSG_DONE` to its new right neighbor. Process  $x$  is again in state  $S3$  and the left-to-right order of processes is preserved. Process  $x$  could conceivably be repeatedly redirected, i.e. prevented from sending its input data. However, the finite size of the communicator and the left-to-right ordering ensures that it will eventually be served by the root process.

We have thus demonstrated that a process will either complete a handshake or be redirected. By repeated application of this principle all processes will have contributed their input data and the reduction will complete.

As an observation, we should point out that due to time races in the arrival of completion messages the algorithm is non-deterministic in the order of processes combined, even for balanced PATs. This might lead to undesirable results if the combining operation is not strictly associative, as is the case with floating point data, which could lead to rounding errors or over/underflows.

### 5.2.5 Time complexity of Local Redirect

We will now examine in more detail the runtime of algorithm Local Redirect in presence of balanced PATs. The best case performance will occur when the algorithm follows a Binomial Tree schedule, with the additional overhead of redirects and handshakes in each of the  $\lceil \log_2 P \rceil$  rounds. Assume that  $P = 2^x$ ,  $x \in \mathbb{N}$ . If the cost of redirection is denoted by  $r$  and the cost of handshakes by  $h$ , then the best case performance  $t_{\mathcal{L}}^b = \log_2 P(d + h) + (\frac{P}{2} - 1)r$ .

Quantifying worst case performance of the algorithm is trickier. There are two factors involved: additional rounds of reduction due to suboptimal process pairing and redirections that take place due to time races and imbalanced PATs. Let us first examine what impact the imbalanced PATs may have on process pairing.

As processes in the algorithm only have the information on the arrival order of their two immediate neighbours, there will be many PATs for which the algorithm will not produce a good schedule tree. We will highlight a particular type of PATs  $\psi$ , for which the algorithm ( $\mathcal{L}$ ) will have absorption time  $A(\psi, \mathcal{L}) = 0$ .

**Proposition 5.2.4.** *Worst case imbalance. Without loss of generality, let  $P$  be even. Let  $\psi'$  be a PAT where every odd process has the arrival time  $a_M$  and every even process the arrival time  $a_m$ . Let  $a_M > a_m$ . Then the absorption time of Local Redirect  $A(\psi', \mathcal{L}) = 0$ . Equivalently,  $t_{\mathcal{L}}(\psi') = t_{\mathcal{L}}(\pi) + I(\psi')$ , where  $I(\psi') = a_M - a_m$ .*

*Proof.* In  $\psi'$  the ready processes that arrive at time  $a_m$  are a distance of 2 from each other. As each ready process in the algorithm examines only its immediate neighbours (distance 1) it will discover that they are both busy and will idle. After time  $a_M - a_m$  the busy processes will have become ready and the reduction can proceed as it would for a balanced PAT for a total time  $t_{\mathcal{L}}(\pi) + I(\psi')$ .  $\square$

The worst case imbalance  $\psi'$  as identified here is not unique, but is minimal in the sense that out of all PATs  $\psi$ ,  $I(\psi) > 0$  that will result with  $A(\psi, \mathcal{L}) = 0$ , the average arrival time  $\bar{a}(\psi')$  is minimal.

Algorithm Local Redirect will require the most rounds when at each round of reduction it combines the least possible number of process pairs. As a process can either combine with its left or right neighbour, we can arrange the pairings so that for every three processes, one is left unpaired. Let  $g(n)$  be the number of processes remaining to be combined in round  $n$  using the worst case strategy and let  $g(0) = P$ . It is not hard to show that this strategy leads to the recurrence relation:

$$g(n+1) = \lfloor \frac{g(n)}{3} \rfloor + \lfloor \frac{1}{2}(g(n) - \lfloor \frac{g(n)}{3} \rfloor) \rfloor + (g(n) - \lfloor \frac{g(n)}{3} \rfloor) \% 2,$$

where  $g(0) = P$  and  $g(n)$  is the number of processes remaining to be combined in round  $n$  and  $\%$  is remainder of division operator. The recurrence can be further

simplified into

$$g(n+1) = \lceil \frac{1}{2}(\lfloor \frac{g(n)}{3} \rfloor - g(n)) \rceil + g(n)$$

We can eliminate the floors and ceilings from the recurrence by substituting  $g(n) = 6k + i, i \in \{0, 1, 2, 3, 4, 5\}$  and solving for  $\forall i$ . This will give us the following solutions for  $g(n+1)$ :  $4k, 4k+1, 4k+1, 4k+2, 4k+3, 4k+3$ , for  $i = 0, 1, \dots, 5$  in that respective order. It follows that  $g(n+1) = \lfloor \frac{2}{3}g(n) \rfloor$ , when  $g(n) = 6k + j, j \in \{0, 2, 3, 5\}$ , otherwise  $g(n+1) = \lceil \frac{2}{3}g(n) \rceil$  for  $g(n) = 6k + j, j \in 1, 4$ . We can either solve both recurrences separately or approximate them by accepting a maximum absolute error of 1 as  $g(n+1) \approx \frac{2}{3}g(n)$ . We choose the latter, i.e. we will approximate the two recurrences as

$$g(n+1) \approx \frac{2}{3}g(n). \quad (5.4)$$

Telescoping Equation 5.4 and inputting the initial condition leads to the following solution:

$$g(n) \approx \left(\frac{2}{3}\right)^n P$$

Finally, solving for  $n$  gives us the worst case number of rounds:

$$n \approx \log_{\frac{2}{3}} P \Rightarrow n \approx 1.71 \log_2 P$$

It is reasonable to expect that the number of redirections here would be roughly that shown for the best case performance, implying a ratio of 1.71 between worst and best case performance for balanced PATs. However, if we consider pathological redirections, the worst case performance may be significantly worse. This would be the case where every process is redirected to the root for a total of  $\frac{(P-1)(P-2)}{2}$  redirections and  $P-1$  reductions.

## 5.3 Experiment Setup

To evaluate algorithm robustness to imbalanced PATs, we constructed five experiments, each differentiated by the nature of the injected PAT pattern:

1. PAT pattern where only a single process in the communicator is delayed by time  $I(\psi)$ .
2. PAT pattern where every odd process is delayed by time  $I(\psi)$ .
3. PAT pattern where a progressively larger number of processes are delayed by time  $I(\psi)$ .
4. PAT pattern where process arrivals were distributed according to either a truncated or unbounded gamma distribution.

5. PAT pattern determined by a trace of 100 iterations of image render times from the Helsim simulation.

As demonstrated by Theorem 5.2.2), the first PAT pattern is one where we expect to observe the largest speedups. The second PAT pattern was selected as one where neither Local Redirect nor Clairvoyant are expected to have any resiliency to imbalance (Proposition 5.2.4). The third experiment was designed to demonstrate the importance of available slack in the PATs for the achievable speedup of imbalance robust algorithms. In this experiment the absolute imbalance  $I(\psi)$  was kept constant, but the number of delayed processes was progressively increased from 1 to half the communicator size. The fourth experiment was designed to evaluate PAT patterns falling between the two extremes presented in the first and second experiment, as might be expected in real-world applications. Gamma distribution has long been advocated as for modelling job runtimes [45]. It is a flexible distribution offering the possibility of adjusting its expected value and standard deviation, is positive, has a tail extending to infinity and may be adjusted to have a mode instead of being monotonically decreasing. Finally, the last experiment was constructed as a case study of PATs in a real-world application.

The experiment was setup as follows: for each message size  $m \in \{ 40 \text{ KiB}, 200 \text{ KiB}, 400 \text{ KiB}, 1 \text{ MiB}, 4 \text{ MiB} \}$  a different allocation of 128 nodes was performed. Message elements were of type `MPI_INT` and the utilized combining operator was `MPI_SUM`.

For each combination of input parameters (message size  $m$ , and PAT pattern  $\psi$ ), the microbenchmark is invoked with separate calls to *mpirun*. In addition, for each problem size  $m$  that was evaluated, a different allocation of  $P = 128$  nodes was chosen, as produced by the batch system on the PRACE CURIE supercomputer (`slurm 2.6.9-Bull.1.1`).

To estimate the time  $d$  required to complete one round of the reduction (the time to receive and combine a message of size  $m$ ), we used the message transmission time microbenchmark (Algorithm 6) introduced and explained in Chapter 4. Due to the multimodal nature of the underlying distribution of collected samples, we decided to use the median as the measure of location, the decision that was supported by bootstrap analysis conducted in Chapter 4.

Each sample in the reported results is composed out of 256 runtime observations. Because many communication systems establish their working state on demand, we exclude the first measurement out of each sample. Given the large sample size, this has a small effect on the statistics. At each iteration, prior to performing the collective reduction operation, all processes in the communicator generate a new input buffer state using a pseudo-random number generator. Depending on input data size, this allows for the possibility that the input data resides within the cache memory of corresponding CPUs. As the measure of algorithm runtime, we report the

minimum observed runtime, as suggested in [59] as the most reproducible measure.

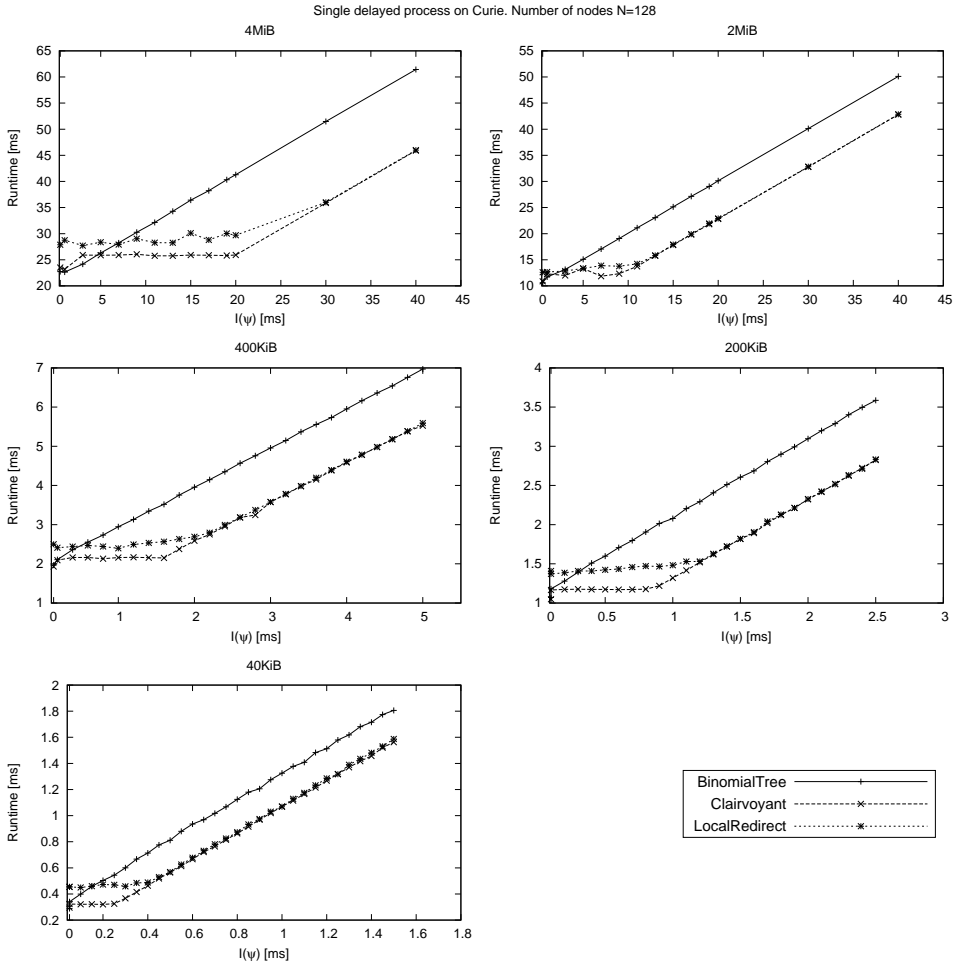
## 5.4 Results and discussion

The experimental findings for the first PAT pattern (single delayed process) are shown in absolute values in Fig. 5.5. For the smallest problem sizes the observed runtime was in the sub millisecond range, while for the largest ( $m = 4$  MiB) they spanned between 20 ms and 65 ms. It is immediately apparent that algorithm Binomial displayed no resiliency to the injected imbalance in PATs, as its observed runtime closely corresponds to the sum of base runtime for the balanced PAT  $\pi$  and the absolute imbalance  $I(\psi)$ . On the other hand, both algorithm Local Redirect and Clairvoyant exhibited strong resiliency to the injected imbalance. Despite the injected imbalance, the observed runtime of both algorithms roughly matches the base runtime, until their maximum absorption limit is reached. Algorithms Local Redirect and Clairvoyant algorithms converge in performance once the absolute imbalance is large enough.

The observed runtime of algorithm Local Redirect is consistent with the estimate proposed in subsection 5.2.3:  $t_{\mathcal{L}}^b = \log_2 P(d+h) + (\frac{P}{2} - 1)r$ . For the balanced PAT  $\pi$  and problem size  $m = 40$  KiB the observed runtime of algorithm Local Redirect was 454  $\mu$ s vs 449  $\mu$ s predicted. The redirect ( $r$ ) and handshake ( $h$ ) overhead was modeled as two and three control messages, respectively ( $\alpha = 2.66$   $\mu$ s). The points where algorithm Local Redirect overtakes Binomial Tree is consistent with  $A(\psi, \mathcal{L}) = I(\psi)$ . Fig. 5.8 and Appendix A.7 depict the measured absorption times of each algorithm for all the evaluated problem sizes. We can see that the absorption time of algorithm Local Redirect closely matches the absolute imbalance, until it approaches its absorption limit, explaining the overtake points. It can be observed that the absorption time of algorithm Binomial Tree, as stated in Proposition 5.2.1, is close to zero.

Finally, to better appreciate how close the observed runtime of algorithm Clairvoyant matches the prediction of Theorem 5.2.2 we produced a relative runtime plot (Fig. 5.5). Superimposed onto the plot is the relative runtime predicted by Theorem 5.2.2. The observed speedup of algorithm Clairvoyant roughly corresponds to the theoretical model. While algorithm Clairvoyant achieves a maximum observed speedup of 1.69, this is still short of 1.75, as predicted by Theorem 5.2.2. For balanced PATs  $\pi$ , algorithm Clairvoyant, as expected, performs nearly identical to Binomial Tree. For balanced PATs, the relative performance of algorithm Local Redirect shows an improving trend with increasing message size: from  $0.64t_{\mathcal{B}}$  to  $0.84t_{\mathcal{B}}$ . This trend can be explained by the smaller relative impact of synchronizations and redirects to the algorithm's runtime the larger the messages become.

Fig. 5.8 and Appendix A.7 confirm that the maximum absorption time of algo-



**Figure 5.4:** Observed algorithm runtime for the PAT pattern  $\psi$  wherein a single process (rank  $P - 1$ ) was delayed for time  $I(\psi)$ . Each plotted data point represents the minimum of 256 observations.

gorithm Clairvoyant is less than predicted. We conjecture that this is due to overheads outside the tree reduction itself and is a consequence of method invocation latency and memory buffer allocations. If we model the algorithm runtime as consisting of a fixed overhead and the variable tree reduction time, i.e.  $t_C = \sigma + r_c$  then according to Theorem 5.2.2 the maximum speedup over Binomial Tree, assuming  $t_C = t_B$  can be expressed as:

$$S' = \frac{\sigma + 2r_c}{\sigma + r_c + t_s}$$

Clearly,  $S'$  will be maximal when  $\sigma = 0$ . Profiling the implementation of algorithm Clairvoyant for  $m = 40$  KiB has revealed a nearly constant overhead  $\sigma = 0.44$  ms. With the experimentally determined  $r_c = 0.244$  ms and  $t_s = \frac{r_c}{7}$ , using the above equation we can derive the maximum practical speedup  $S' = 1.65$  which is close to the maximum observed speedup of 1.67 for this message size. The remaining discrepancy of 0.02 is likely the result of network noise or variation in the overhead  $\sigma$ . The combined impact of the two is observable in the oscillations of measured absorption times for algorithm Binomial Tree in Fig. 5.8.

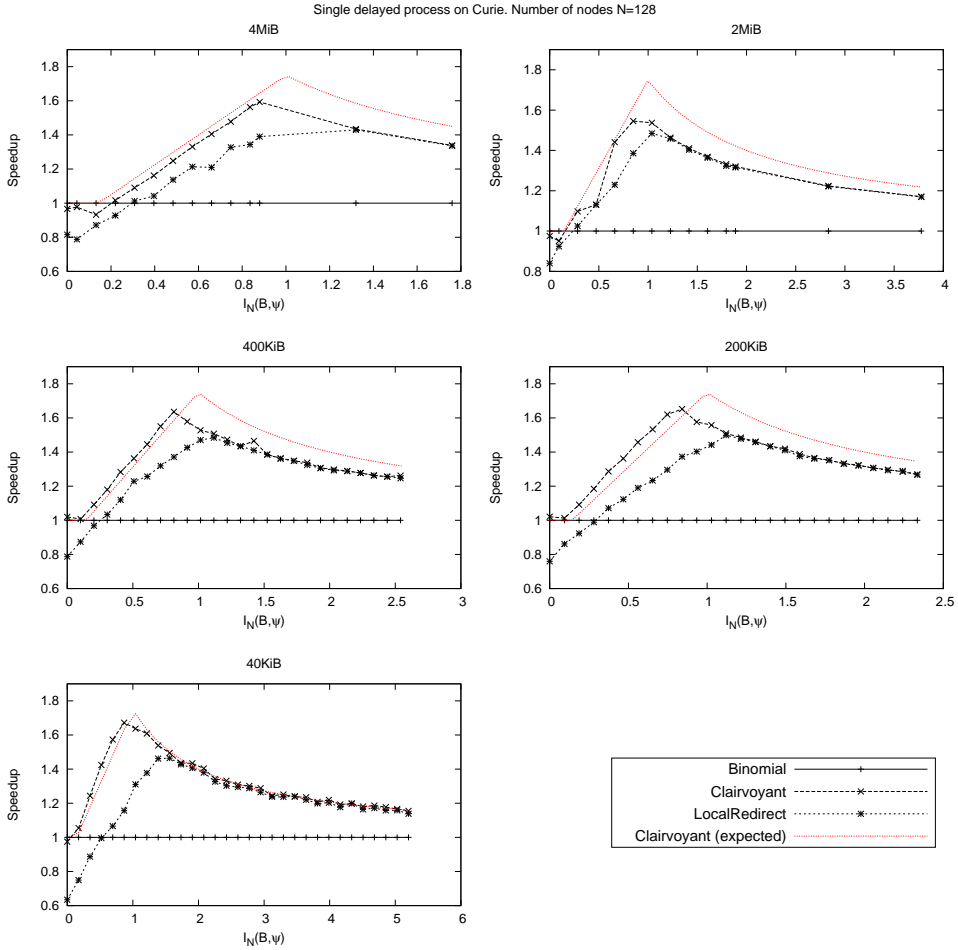
The fact that this was achieved despite the observed heterogeneity of the network leads us to conclude that this particular PAT is forgiving towards message transmission time deviations.

The runtime of algorithm Clairvoyant spans only the time required to execute the computed schedule. The computation of the schedule was performed a priori, where the computation time averaged  $36.16 \mu\text{s}$  ( $\sigma = 4.13 \mu\text{s}$ ). Although algorithm Clairvoyant almost universally dominates in performance, it fails to attain the expected performance predicted by Theorem 5.2.2. In fact, in none of the experiments does it approach the maximum theoretical speedup of 1.75.

It is interesting to see how the choice of the message transmission and combination cost  $d$  impacts the performance of algorithm Clairvoyant for this pattern of PATs. Our empirical investigation (Table 5.2) supports our selection of the median ( $p_{50}$ ) as the estimate of message transmission time that leads to the best runtime performance of algorithm Clairvoyant.

While the previous pattern of PATs was identified as one where we might expect optimal speedups of algorithms Clairvoyant and Local Redirect compared to algorithm Binomial, the second investigated pattern where every odd process is delayed should result in runtime performance of the two algorithms no better than the Binomial Tree, as both of the algorithms are expected to have zero absorption time. Our experimental findings roughly confirm this hypothesis (Fig. 5.9). For this PAT pattern, algorithm Local Redirect performs strictly worse than the Binomial Tree algorithm. As expected, algorithm Clairvoyant shows runtime performance similar to that of Binomial Tree.

Similar results were also observed on the VSC muk cluster (Appendix A.8). The performance evaluation for the third PAT pattern was conducted only on the



**Figure 5.5:** Relative algorithm performance where the PAT pattern  $\psi$  is composed of a single delayed process (rank  $P-1$ ). Red line denotes the expected (modelled) runtime of algorithm Clairvoyant relative to measured algorithm Binomial Tree runtime. Computation of expected runtime is based on the speedup predicted by Theorem 5.2.2 and the assumption that  $t_B(\pi) = t_C(\pi) \wedge t_B(\psi) = t_B + I(\psi)$ .

VSC muk cluster for problem size  $m = 2$  MiB, and consisted of 17 PAT patterns (Appendix A.8). Each pattern had  $k, k \in \{0..16\}$  randomly selected processes delayed for a constant time  $\tau \in \{4 \text{ ms}, 8 \text{ ms}, 15 \text{ ms}\}$ . The experimental findings indicate a strong dependence of speedup on the available slack, as expected.

To simulate PAT patterns that would more closely resemble those expected to be observed in real-world systems, we generated two different types of gamma distributed PAT patterns.

The first type, the unbounded gamma distributed PAT patterns were constructed by selecting the average PAT time  $\bar{a} = 2$  ms and the coefficient of variation  $c_v = 12.0$ . For this experiment, the problem size was set to  $m = 400$  KiB. The results of this experiment (Table 5.3) show that both the Local Redirect and Clairvoyant algorithms outperform Binomial Tree, if by a small margin: by 6% and 7%, respectively, for median runtime. A similar experiment was conducted on the VSC muk machine (Appendix A.8).

For the second type, the truncated gamma distributed PATs, we generated two PATs files with  $(\bar{a} = 2 \text{ ms}, \max a = 4 \text{ ms})$  and  $(\bar{a} = 4 \text{ ms}, \max a = 4 \text{ ms})$ . Each file is a matrix of dimensions  $(P, n)$ , where  $P = 32$ , and  $n = 1024$ . In this way, we have produced 1024 different PATs.

This experiment was conducted only on the VSC muk machine. Each algorithm was evaluated 90 times for each file, rendering a sample of 90 observations per PAT pattern. Appendix A.9 shows a scatter plot wherein each point is the median of 90 observations per PAT pattern, for a total of 1024 points. For this PAT pattern, algorithm Clairvoyant universally dominates the Binomial Tree algorithm with the majority of the observed speedups falling in the range  $[1.05, 1.15]$ . For the first series of gamma distributed PAT patterns  $(\bar{a} = 0.42t_B)$ , algorithm Clairvoyant achieves an average speedup of 1.11, compared with 1.07 for algorithm Local Redirect. For the second series, however, it is outperformed by algorithm Local Redirect that achieves an average speedup of 1.16 compared with 1.08 for algorithm Clairvoyant.

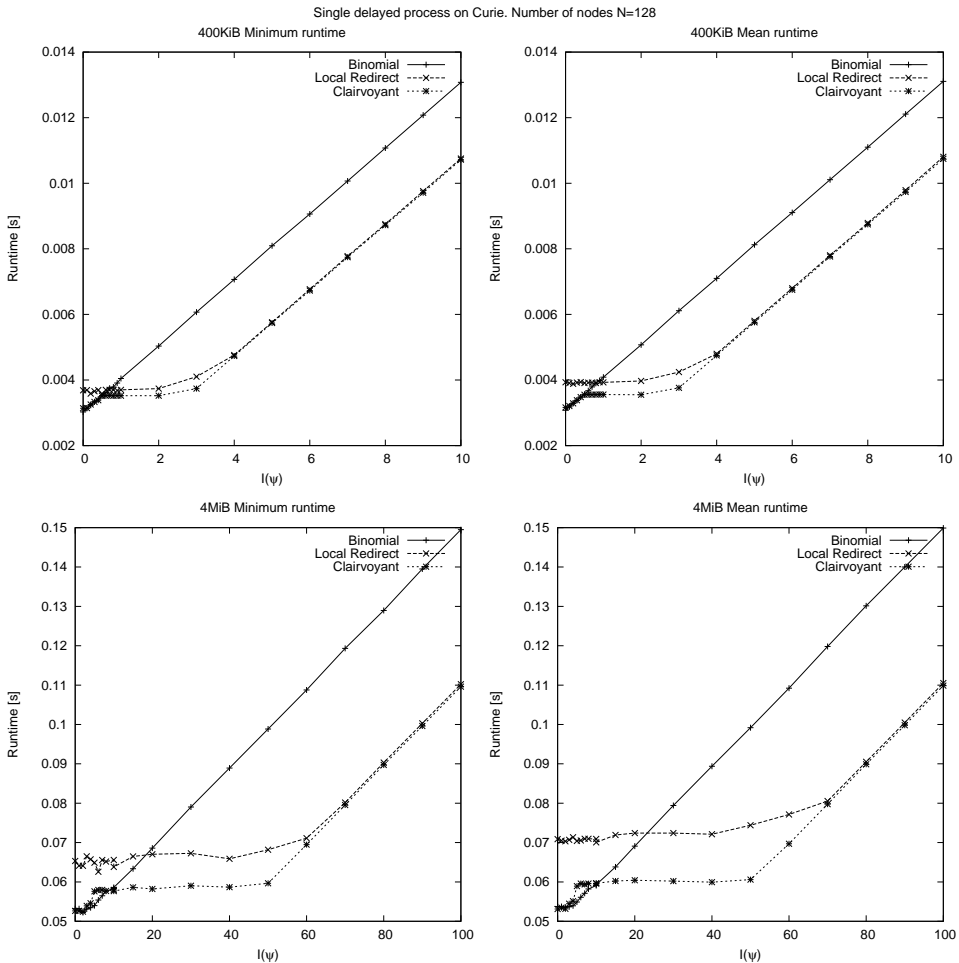
Finally, we used the microbenchmark to simulate the PATs observed during the image compositing phase of the Helsim simulation (depicted in Fig. 3.6). In this experiment, two different PAT patterns were supplied to algorithm Clairvoyant. The first represented a perfect prediction of the PAT patterns at the current collective operation invocation point, while the second one consisted of PAT patterns computed using the SMA model, described in Chapter 3 (Fig. 3.6). The PAT information was dissemination after each invocation ( $k = 1$ ).

#### 5.4.1 Simulated performance using the Helsim trace file

The PAT vectors passed as input to algorithm Clairvoyant constituted a perfect prediction, unlikely to occur in practice. As was discussed in Chapter 3, Section 3.7 PATs passed to clairvoyant algorithms would normally be computed using some

**Table 5.1:** SMA model predicted PATs vs. true PATs clairvoyancy. Data generated on the VSC muk cluster;  $P = 64$ , one process per node. Problem size  $m = 16$  MiB. Reported values are the mean accumulated runtime for 100 consecutive reduction operations, in a sample of 5 observations.

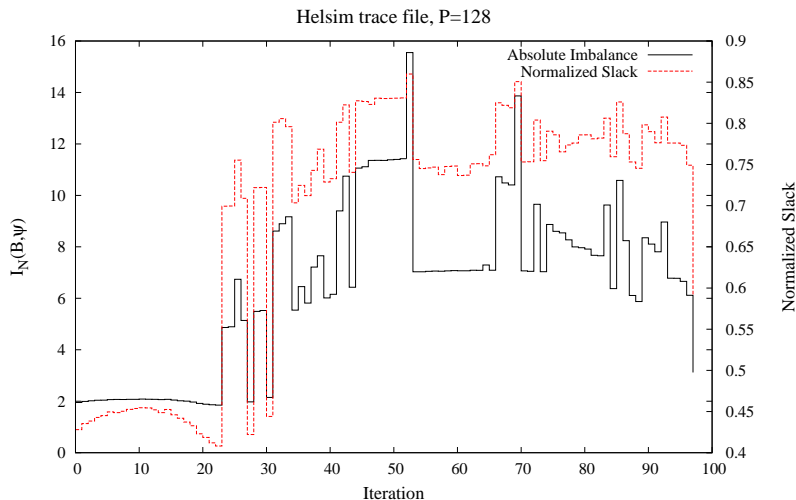
Binomial Tree	Local Redirect	Clairvoyant (SMA)	Clairvoyant (true)
13.88 s	13.04 s	12.94 s	12.81 s



**Figure 5.6:** Observed algorithm runtime with single delayed process (rank  $P-1$ ). The data was gathered on the PRACE CURIE supercomputer with the microbenchmark linked to BullxMPI version 1.2.7.2.

model based on historical data. As a preliminary study, we have used the simple moving average (SMA) model described in Section 3.7 to generate PATs based on the trace file from the 128 process run of the Helsim simulation (Fig. 3.5).

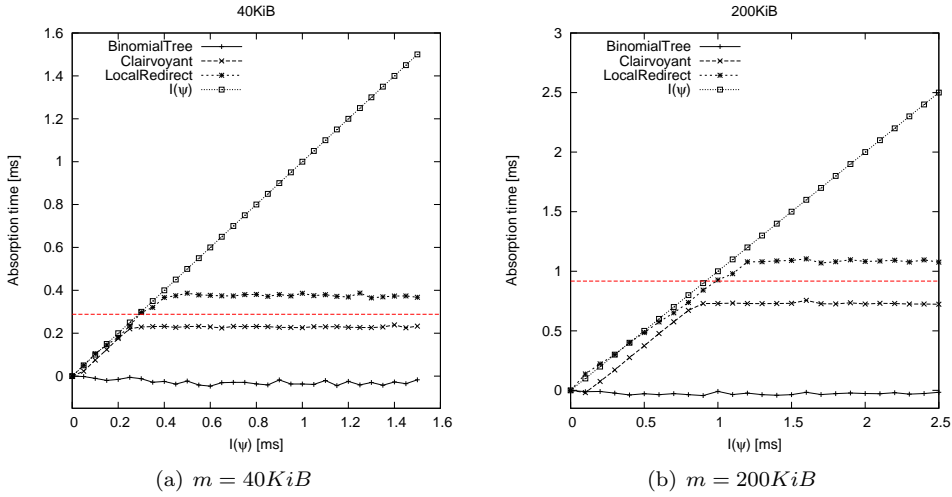
The experiment was constructed as follows. We extracted the PAT data for the first 64 process from the Helsim trace file and used the SMA model (Fig. 3.7) with window size 5 to compute the predicted PATs. For each recorded iteration of the Helsim simulation, the microbenchmark injected the imbalances according to the trace file, while the PAT prediction passed to Clairvoyant was computed by the SMA model. We ran the microbenchmark on the VSC muk cluster, for a communicator of size  $P = 64$ , with one process per node. The results of the experiment are shown in Table 5.1. In this particular example, the absolute imbalance dominates the runtime (Fig. 5.7) of the reduction operations, and thus the relative runtimes only marginally differ.



**Figure 5.7:** Absolute imbalance and slack in the Helsim trace file (100 recorded image render times on a communicator of size  $P = 128$ ).

### 5.4.2 The problem of different communication library versions

While our decision to report the minimum of all measured runtimes was based on the guidelines of [59] as the best way to achieve reproducible results, examining the distribution of runtimes can reveal potential problems if there are significant deviations from the mean time. This was observed for algorithm Local Redirect



**Figure 5.8:** Absorption time (CURIE,  $N=128$ ). Red arrow marks the maximum expected absorption time for algorithm  $\mathcal{C}$  (Clairvoyant), assuming  $t_{\mathcal{C}}(\pi) = t_{\mathcal{B}}(\pi)$  and  $\mathcal{B}=\text{BinomialTree}$ .

runtimes. When moving from BullxMPI version 1.2.7.2. to 1.2.8.2., the mean runtime of the algorithm increased by two orders-of-magnitude, as shown in Appendix A.6. We can see that the distribution of Local Redirect’s runtime became multimodal, a recurring pattern even with increasing values of absolute imbalance. We conjecture that this behaviour might be related to the manner the unexpected message queue is handled in the newer version of the library. This claim is supported by microbenchmarks of a modification of the algorithm, where the redirection scheme was modified so as to avoid multiple redirect messages swamping any single process (Local Redirect (optimized) in Appendix A.6).

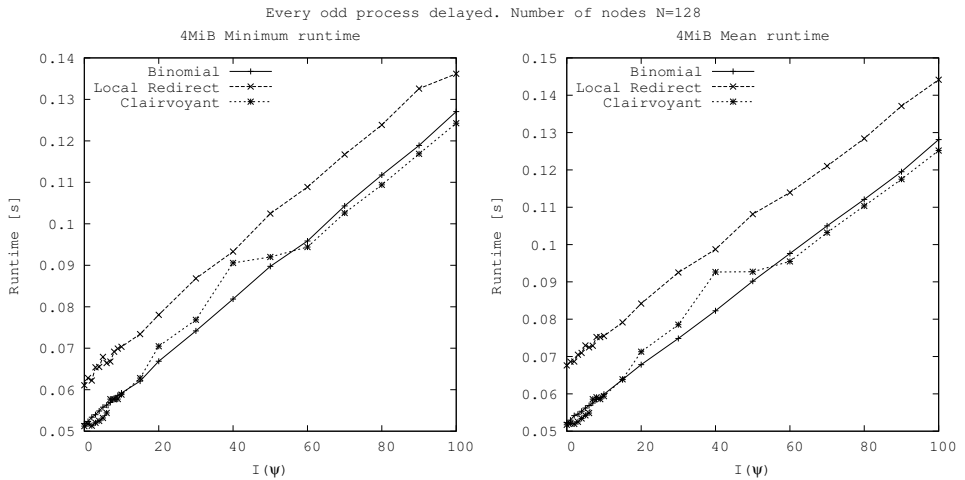
This behaviour was not observed when using BullxMPI version 1.2.7.2., where the minimum and mean runtimes of algorithm Local Redirect were closely matched. We provide the results of the single delayed process experiment for BullxMPI version 1.2.7.2. in 5.6. We report the relative speedups for both the minimum and mean observed runtimes. As can be seen, the relative speedups for minimum observed runtimes are roughly equivalent across the different library versions.

## 5.5 Summary

We presented two novel reduction algorithms: one static and one dynamic. The static, clairvoyant algorithm was proven optimal (Theorem 5.1.8) for reduction of atomic messages in homogeneous, fully connected, simultaneous send/receive no-

**Table 5.2:** Runtime of algorithm Clairvoyant as function of different estimates of message transmission time  $d$  for  $m = 40$  KiB. Quartiles of 819200 collected samples are used as estimates.

Algorithm	Metric	Estimate of message transmission time			
		p25	p50	p75	p95
Clairvoyant	min. [s]	$3.3440 \times 10^{-4}$	$3.2541 \times 10^{-4}$	$3.3395 \times 10^{-4}$	$3.3853 \times 10^{-4}$
	median [s]	$3.7458 \times 10^{-4}$	$3.5171 \times 10^{-4}$	$3.6675 \times 10^{-4}$	$3.9836 \times 10^{-4}$



**Figure 5.9:** Observed algorithm runtime with an alternating PAT pattern  $\psi$ . Every odd process was delayed by time  $I(\psi)$ . The data was gathered on the PRACE CURIE supercomputer with the microbenchmark linked to BullxMPI version 1.2.7.2.

**Table 5.3:** Observed runtime for  $m = 400$  KiB and gamma distributed PATs. The reported statistics are computed from a data set of 100 observations. For each observation, a different instance of PATs was generated from a random generation process following a gamma distribution ( $\bar{a} = 2$  ms  $\wedge$   $c_v = 12.0$ ). The number of nodes  $P = 128$ . All results were generated on the PRACE CURIE supercomputer with the microbenchmark linked to BullxMPI version 1.2.7.2.

	min	mean	median
Binomial Tree	0.0194 s	0.0378 s	0.037 s
Local Redirect	0.018 s	0.0359 s	0.0349 s
Clairvoyant	0.0174 s	0.0355 s	0.0345 s

overlap networks (Definition 2.5.1). The implementation of this algorithm was evaluated against the Binomial Tree algorithm for a range of imbalanced PAT patterns on the PRACE CURIE supercomputer and VSC muk cluster. The algorithm was shown to dominate in runtime for the large majority of test points (Figure 5.5).

For the PAT pattern where a single process was delayed, the algorithm reached up to 95% of the predicted maximum speedup (1.67 vs 1.75), as predicted by Theorem 5.2.2. The observed performance gap was shown to be a consequence of memory allocation overheads, in absence of which the algorithm would have met the theoretical prediction. This was achieved despite the heterogeneous nature of the utilized network. The algorithm is contingent on full knowledge of process arrival times at the invocation point of the reduction operation making it applicable to cases where process arrival times follow a recurring pattern or where the pattern is predictable. We conjecture that the same algorithm could be minimally modified to perform optimal broadcast.

In addition, we presented a dynamic imbalance resilient reduction algorithm, agnostic of PATs and therefore applicable to situations when these are unpredictable. This algorithm, while not quite matching Binomial Tree in performance for balanced PATs, consistently outperforms it whenever the absolute imbalance is greater than  $0.3t_B$  (single delayed process PAT pattern), where  $t_B$  is the runtime of algorithm Binomial Tree for balanced PATs. In this case, the maximum observed speedup was close to 1.5. When the PATs were distributed according to those traced from a real-world plasma physics simulation, the algorithm was shown to bring a 7.6% performance advantage.

## Chapter 6

# Clairvoyant reduction algorithm for non-atomic messages

---

This chapter is devoted to the performance evaluation of reduction algorithms operating on non-atomic input data. Such algorithms can offer significantly higher performance than the atomic data reduction algorithms analyzed in Chapter 5. As was demonstrated in Chapter 2 there exists a rich body of research on this topic and many near optimal algorithms for balanced Process Arrival Times (PATs) have been long known and can be found in many a Message Passing Interface (MPI) library implementation.

The main challenge tackled in this chapter is the problem of designing and implementing an optimal collective reduction algorithm for non-atomic input data. We discuss what optimality in this context might mean, and present a new imbalance resilient reduction algorithm that preconstructs reduction schedules that are of minimal possible length given a particular PAT pattern. In this way the algorithm mitigates the impact of PAT imbalance by performing as much of the reduction while a part of the processes are still unavailable. We compare its performance with a selection of four most frequently used reduction algorithms. We ensure that all algorithm implementations adhere to the function interface and semantics defined by the MPI standard [125] for `MPI_Reduce`. We perform the experiments on 128 nodes of the PRACE Curie supercomputer.

In addition to this, we also show how relevant current collective operation microbenchmarks are in predicting the performance of reduction algorithms for imbalanced PATs. This is important because all microbenchmarks, without exception, evaluate collective operation performance under balanced PATs only. We demonstrate this through the use of the microbenchmark explicated in Chapter 4. The relative rankings of the algorithms are then compared with and without presence of balanced PATs.

This chapter is structured as follows. The next section analyzes the absorption

potential of competitive algorithms and defines the maximally achievable levels. Section 6.2 introduces and defines the novel segmenting clairvoyant schedule generation algorithm. Its time complexity is analyzed both theoretically and empirically. The subsequent section examines in brief the time complexity of evaluated algorithms and attempts to produce orderings based on the linear complexity model introduced in Chapter 3. Section 6.4 presents an analysis of experimental data and the rationale behind the selection of statistical measures used the results report. The subsequent section presents and discusses the experimental findings, and the implications pertaining to implementations of reduction algorithms. Finally, Section 6.7 concludes the chapter.

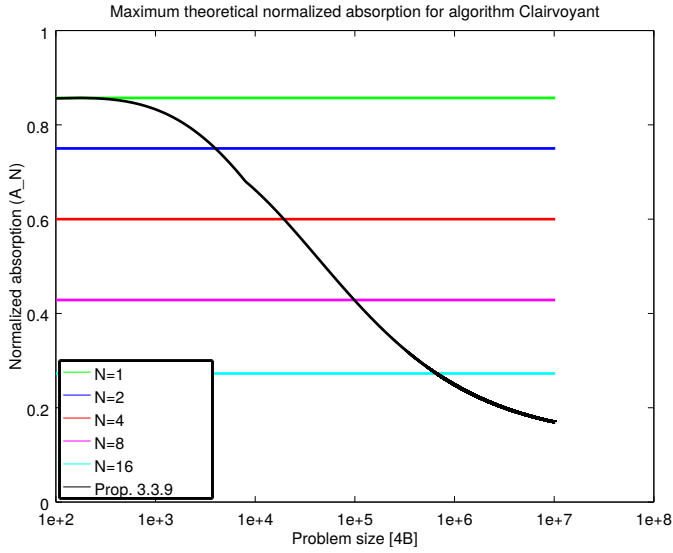
## 6.1 Absorption potential of a clairvoyant reduction algorithm

As previously discussed in Chapter 3, no reduction algorithm can meet all three lower bounds: latency, bandwidth and computation. Thus the algorithm  $\mathcal{O}$  that solves the two way reduction problem in optimal time (Proposition 3.3.10) will employ one of the following two strategies: either the workload is evenly divided between the two processes by first performing a reduce-scatter operation followed by a gather operation for the total time complexity  $t_1 = 2\alpha + m\beta + 1/2m\gamma$  or the latency is minimized by having one process send all its data to the root in a single message, with the root performing all the computation, for time  $t_2 = \alpha + m\beta + m\gamma$ . The first strategy will be better whenever  $\alpha < 1/2m\gamma$ . Computing the maximum absorption according to Proposition 3.3.10 we can observe a switch from the second to first strategy near the point  $m = 10 \text{ KiB}$  (Fig. 6.1). The exact point will depend on linear model parameters of each particular machine. This result, combined with the fact that we evaluated algorithm performance for  $m \geq 128 \text{ KiB}$  has motivated our decision to opt for the first strategy of distributing the computational workload in the design of algorithm Clairvoyant.

In fact, there is another strategy for implementing algorithm  $\mathcal{O}$ : utilize the remaining  $P - 2$  processes to decrease the per-process workload. To do that, a scatter from each of the 2 processes to other  $P - 1$  processes would be required, followed by local computation on data of size  $\frac{m}{P}$  concluded by a collective gather of computed  $P - 1$  blocks of size  $\frac{m}{P}$  to the root. The total time of this operation is:

$$2\alpha \log_2 P + 2\frac{P-1}{P}m\beta + \frac{1}{P}m\gamma$$

This strategy is hampered by extra latency of roughly  $\log_2 P$  and extra data transmission time of  $m\beta$ . For very large messages and moderately large systems,



**Figure 6.1:** Theoretical prediction of maximum normalized absorption of algorithm Clairvoyant as function of problem size (multiples of 4 bytes) and number of segments ( $N$ ). Normalized absorption is computed according to the equation:  $A_N = \frac{t_C(\pi, 128) - t_C(\pi, 2)}{t_C(\pi, 128)}$ , where  $t_C(\pi, P)$  is the runtime of algorithm Clairvoyant for balanced PATs as defined in Table 6.2. The black curve was computed according to Proposition 3.3.10. Parameters  $\alpha = 2.66 \mu\text{s}$ ,  $\beta = 4.8179 \times 10^{-10} \text{ sB}^{-1}$ ,  $\gamma = 1.6654 \times 10^{-10} \text{ sB}^{-1}$ ; experimentally determined on the PRACE CURIE supercomputer

we can ignore the latency term leading to the conclusion that network bandwidth would have to be double that of computation speed. This is, however, contrary to current trends in high performance systems where computation speed is upwards of three times that of effective bandwidth.

## 6.2 Clairvoyant (non-atomic) schedule generation

We define the clairvoyant non-atomic schedule generation algorithm as Algorithm 10. Henceforth, for reasons of brevity, we refer to this algorithm as *Clairvoyant*. This should not create confusion with Algorithm 8 defined in Chapter 5, as the two algorithms will not be compared in the same context.

The algorithm generates a reduction schedule which is personalized for each of the  $P$  processes. This algorithm is executed by all the  $P$  processes in the communicator, to avoid communication costs. However, it might also be viable to have the first process to arrive compute the schedule and scatter it to the remaining  $P - 1$  processes.

The algorithm operates in rounds, within which a process can at most send one segment, receive one segment and combine one segment. The data is split into  $N$  equal sized segments, which in turn determines the minimum required number of rounds. At the beginning of each round, the algorithm establishes which processes are ready to participate in the reduction. This is done by first selecting the process  $Q_0$  with the minimum arrival time, i.e. the top element of the priority queue  $Q$ , where priority is assigned to process ranks  $i$  with smaller (earlier) arrival time  $a_i$  (Line 3).

Initially, the set  $Q$  is comprised of all  $P$  processes. Then a group  $G$  of ready process is constructed from the queue  $Q$ , so that for  $\forall i \in G, a_i \leq a_{Q_0} + d$  (Line 5). Thus all members of  $G$  are formed by those processes whose arrival time is less than or equal to the arrival times of process  $Q_0$  plus the time to complete one round of reduction ( $d$ ). A state matrix  $M$  of size  $P \cdot N$  is used to keep track of states for each segment on every process (Line 1). The possible states are  $A$ , for available,  $E$  for empty (a segment that has been sent) or  $P$  for partially combined segment (a segment that contains information from at least one other segment). The algorithm then proceeds by adhering to a greedy principle: each process attempts to receive and combine a segment that still resides in its local buffer (state  $A$  or  $P$ ) by giving priority to segments with lower indices (Line 21 and 25). Care is taken to ensure that a process sends a segment only once within each round. This is done by keeping record in the vector  $S$  (Line 2). In each round, a sink process ( $r^*$ ) is established: if process  $r$ , the root selected in the collective operation call, is not part of  $G$ , then process  $Q_0$  with minimum arrival time is selected as the sink. Otherwise, process  $r$  is selected (Lines 14-18). Implicit to the algorithm is the principle, that once a process

sends a segment of data, it will no longer receive segments of that index - unless that process is the sink process. This ensures that all segments eventually trickle down to the sink process. The sink process follows a slightly different greedy principle: it attempts to receive a segment regardless of whether a segment of matching index resides in its local buffer, with priority assigned to segments with lower indices (Line 23 and 25). This ensures that even if the root process  $r$  is the last to arrive, the reduction can proceed uninterrupted as long as there are segments to be received and reduced.

At the end of a round, if a process has sent all of its segments then its schedule is complete and it is not put back into the set  $Q$  (Line 36). The algorithm repeats until all processes except the root process  $r$  have been removed from set  $Q$ .

The schedule constructed by the algorithm for  $P = 4 \wedge N = 4$ , when the PAT is balanced, is shown in Table 6.1. The execution of this schedule is illustrated in Fig. 6.2. We will briefly trace the schedule generation algorithm for the first two processes in Round 1. Because the PAT is balanced,  $G = P$  and  $Q_0 = r$ . Beginning with process rank  $i = r = 0$  (the for loop on Line 12), the algorithm sets the index of interest  $j = 0$  (Line 13). The linear search on Line 21 determines that the index  $j = 0$  is indeed eligible ( $(M(i, j) \in \{A, P, E\})$ ) - at algorithm start, all elements of the matrix are set to the value A (Line 1). Then, the algorithm searches for the first process  $z$  (Line 19) among the processes in group  $G$  whose segment of that index has not yet been sent (Line 25). If such a segment has been found on process  $z$ , the algorithm checks that process  $z$  has not already sent a segment in the current round (Line 25). In this case,  $z = 1$  and the algorithm proceeds to Line 30. The inbound queue of process 0 ( $I[0]$ ) is enqueued with the pair  $(z, j) = (1, 0)$ . At line 31 the outgoing queue of process 1 ( $O[1]$ ) is enqueued with the pair  $(i, j) = (0, 0)$  (Compare with Table 6.1). Finally, the state matrix  $M$  is updated accordingly (Lines 32-33).

The algorithm loops back, and the next process  $i = 1$  in the group  $G$  is selected. The search on Line 21 determines that the first eligible segment is of index  $j = 1$ . The linear search on Line 25 determines that the first process that can send the segment of index  $j = 1$  is process rank  $z = 0$ . The algorithm proceeds to line 30, and enqueues the pair  $(0, 1)$  to the inbound queue of process rank 1 ( $I[1]$ ) (Line 30) and the pair  $(1, 1)$  to the outbound queue of process rank 0 ( $O[0]$ ) (Table 6.1). This concludes the first round trace for the first 2 processes.

The execution of the generated schedule for the imbalanced PAT  $\psi = (0, 0, 0, \delta)$ ,  $P = 4 \wedge N = 4$  is illustrated in Fig 6.3. Here we can observe that the algorithm has generated such a schedule that allows for the entire *3-way reduction problem* to be solved among processes  $\{0, 1, 2\}$  by the time process rank 3 arrives at the collective call site.

---

**Algorithm 10:** Clairvoyant non-atomic schedule generation algorithm

---

**Input:**

- 1:  $P$ : integer, the communicator size
- 2:  $N$ : integer, the number of segments
- 3:  $a$ : vector of double, the PAT vector of size  $P$
- 4:  $d$ : double, the time to complete one round of reduction
- 5:  $r$ : integer, the root rank
- 6:

**Output:**

- 7:  $I$ : queue of pair (rank, index) //Inbound schedule queue
  - 8:  $O$ : queue of pair (rank, index) //Outbound schedule queue
  - 9:
  - 10: Let  $M$  be a state matrix of size  $P \cdot N$ . Set  $M(\cdot) = A$ , i.e. mark all segments as available.
  - 11: Let  $S$ : bool be an array of size  $P$
  - 12: For  $\forall i \in \{0 \dots P - 1\}$  insert process rank  $i$  into a priority queue  $Q$ , where priority is given to process ranks  $i$  with smaller arrival time  $a_i$
  - 13: **while** size( $Q$ ) > 1 **do**
  - 14: Pop processes  $Q_0 \dots Q_k$  from  $Q$  to form a sorted vector  $G$ , so that  $\forall i \in G, a_i \leq a_{Q_0} + d$ .
  - 15: **for**  $\forall i \in G$  **do**
  - 16: let  $S(i) = \perp$  // No ready process in  $G$  has yet sent a segment
  - 17: **end for**
  - 18: **if**  $r \in G$  **then**
  - 19: insert  $r$  at first position in  $G$
  - 20: **end if**
  - 21: **for**  $\forall i \in G$  **do**
  - 22: let  $j = 0$ , let  $z = \emptyset$
  - 23: **if**  $r \in G$  **then**
  - 24: let  $r^* = r$  // sink is the root
  - 25: **else**
  - 26: let  $r^* = Q_0$  // sink is the earliest to arrive process
  - 27: **end if**
  - 28: **while**  $z \in \emptyset$  **do**
  - 29: **if**  $i \neq r^*$  **then**
  - 30: starting from  $j$ , find first segment  $x$ , such that  $M(i, x) \in \{A, P\}$ . Let  $j = x$ .
  - 31: **else**
  - 32: starting from  $j$ , find first segment  $x$ , such that  $M(i, x) \in \{A, P, E\}$ . Let  $j = x$ .
  - 33: **end if**
  - 34: perform linear search through group  $G$  to find the first process  $z \neq i$  such that  $M(z, j) \in \{A, P\} \wedge S(z) = \perp$
  - 35: **if**  $z \in \emptyset$  **then**
  - 36: let  $j = j + 1$
  - 37: **end if**
  - 38: **end while**
-

**Algorithm 10:** Clairvoyant non-atomic schedule generation algorithm

---

```

39:   enqueue to  $I(i)$  the pair  $(z, j)$ 
40:   enqueue to  $O(z)$  the pair  $(i, j)$  and let  $S(z) = \top$ 
41:    $M(z, j) = E$ 
42:    $M(j, z) = P$ 
43:   end for
44:   for  $\forall i \in G$  do
45:     if  $\exists j \in \{0 \dots N - 1\} : M(i, j) \neq E$  then
46:        $a_i = a_i + d$ 
47:       push process  $i$  into  $Q$ 
48:     end if
49:   end for
50: end while

```

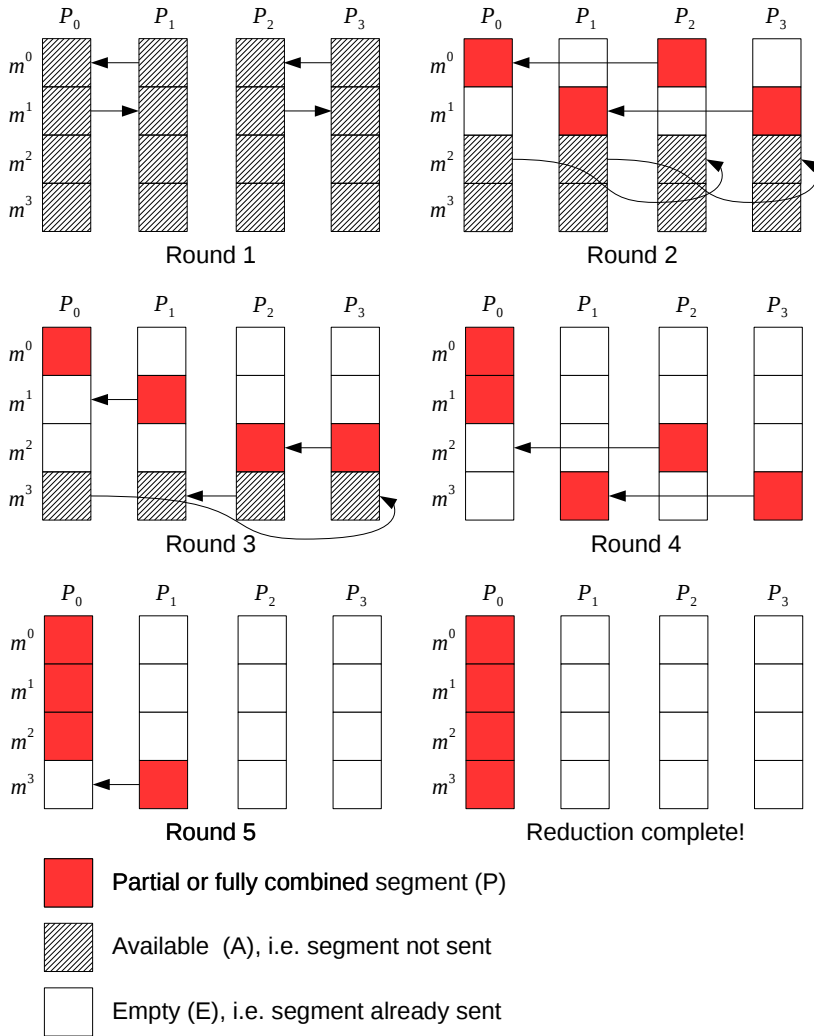
---

For  $N = 1$  the schedule generated by this algorithm degenerates into a binomial tree. We experimentally evaluated the schedule generation algorithm for balanced PATs and all permutations of  $P = \{4, 8, 16, 32, 64, 128, 256, 512\}$  and  $N = \{4, 8, 16, 32, 64, 128, 256, 512\}$ . All the generated schedule lengths were of length  $R = n + N - 1$ , thus matching the optimal equi-segmentation schedule length. At this time we do not have a proof that the schedule generation algorithm produces a schedule of length  $R$  for all input parameters.

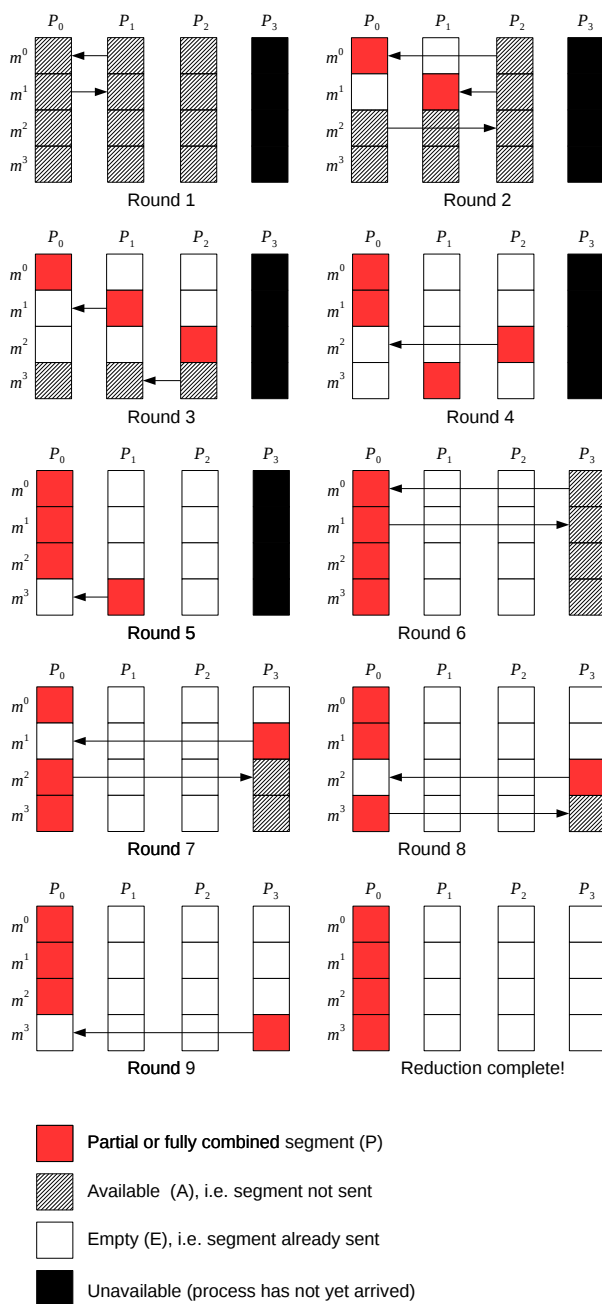
Due to out-of-order combination of data blocks, this algorithm can only be used with commutative operations. This is, however, unavoidable for any algorithm that endeavours to take best advantage of available slack caused by imbalanced PATs.

### 6.2.1 Implementation details

We implemented the algorithm in C++ 11. In the implementation of the algorithm, the linear search in lines 21 and 23 is optimized by placing indices to non-empty segments in a flat set, built over `std::vector`. This was made in favour of asymptotically better ordered list where the cost of item removal is  $O(1)$  vs  $O(N)$  for flat set. Since the number of segments  $N$  is comparatively small, and each element is an integer, a contiguous data structure such as flat set is more cache friendly. Furthermore, the linear search on line 25 is optimized by keeping a priority queue for each segment. The queue is implemented with a pairing heap data structure [53]. The two operations, `heap.push()` and `heap.erase()`, performed by the algorithm both have amortized complexity of  $O(2^{2\sqrt{\log \log N}})$  [147]. Profiling the algorithm execution indicates that 41% of the runtime is spent in `heap.erase()`. We suspect that much of this cost is due to expensive memory deallocation performed by the implementation of `boost::heap::pairing_heap`. Modifying the algorithm to use `heap.decrease()` followed by `heap.increase()`, instead of `heap.erase()`



**Figure 6.2:** Execution of the schedule generated by Algorithm 10 for the balanced PAT  $a = (0,0,0,0)$ , communicator size  $P = 4$  and the number of segments  $N = 4$ . The complete schedule is presented in Table 6.1



**Figure 6.3:** Execution of the schedule generated by Algorithm 10 for an imbalanced PAT. In this example,  $P = N = 4$  and the PAT  $a = (0, 0, 0, \delta)$ , where the delay  $\delta = t_c(\pi, 3)$ , i.e. the time required for algorithm Clairvoyant to solve the 3-way reduction problem. The generated schedule length is  $R = t_c^*(\pi, 3) + t_c^*(\pi, 2) = 5 + 4 = 9$  rounds, where  $t_c^*(\pi, P)$  is the schedule length algorithm Clairvoyant generates for the  $P$ -way reduction problem with balanced PATs 141

**Table 6.1:** Schedule generated by algorithm Clairvoyant (Algorithm 10) for the balanced PAT  $a = (0, 0, 0, 0)$ , communicator size  $P = 4$  and number of segments  $N = 4$ . The schedule consists of two queues I and O (inbound&outbound), where the queue elements are integer pairs (**rank**, **index**), where **rank** denotes the communication peer rank and the **index** denotes the ordinal number of the segment to be communicated. For these input parameters, the generated schedule length is  $R = n + N - 1 = 5$  rounds.

Rank	Queue	$R_1$	$R_2$	$R_3$	$R_4$	$R_5$
0	I	(1,0)	(2,0)	(1,1)	(2,2)	(1,3)
	O	(1,1)	(2,2)	(3,3)	$\perp$	$\perp$
1	I	(0,1)	(3,1)	(2,3)	(3,3)	$\perp$
	O	(0,0)	(2,2)	(0,1)	$\perp$	(0,3)
2	I	(3,0)	(0,2)	(3,2)	$\perp$	$\perp$
	O	(3,1)	(0,0)	(1,3)	(0,2)	$\perp$
3	I	(2,1)	(1,2)	(0,3)	$\perp$	$\perp$
	O	(2,0)	(1,1)	(2,2)	(1,3)	$\perp$

might lead to further performance improvement.

## 6.2.2 Time and space complexity

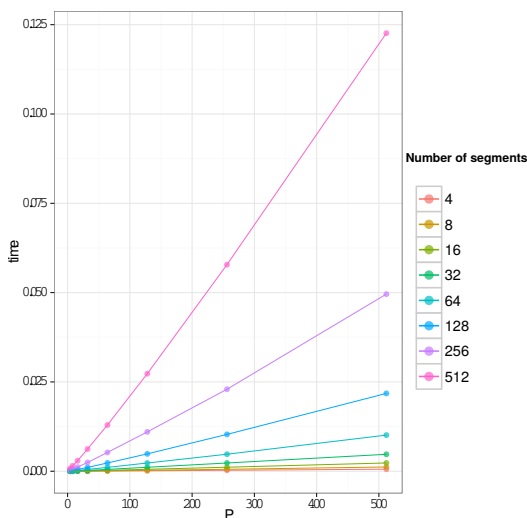
To give a rough estimate of schedule generation time complexity, we will examine the case of balanced PATs. Then the algorithm will take  $R$  rounds where  $R = n + N - 1$ . In each round a loop of at most  $P$  iterations (Line 12) is executed. In each iteration we can assume that on average a single lookup of the first element of both the flat set and pairing heap will be required ( $O(1)$  time), followed by 2 pop (or one pop and one erase) and push operations on the heap, each of which is roughly  $O(\log P)$  in complexity, plus a potentially  $O(N)$  operation to remove an element from the flat set. This leads to a total of:

$$O(P(n + N - 1)(N + 3n)) \approx O(PN^2 + P\log^2 P)$$

However, for small  $N$  such that the flat set resides within cache memory, we can expect the element removal operation for flat set to be of near constant cost, as all the data can be rotated with a single contiguous memory operation. In that case, we can expect the algorithm to scale with the complexity:

$$O(PN + P\log^2 P)$$

Empirical data suggests (Fig. 6.4) that the runtime of the schedule generation algorithms grows roughly linear in both  $P$  and  $N$ . As we will see later, this has strong implications on the usage scenarios of this algorithm.



**Figure 6.4:** Schedule generation runtime as function of number of processes ( $P$ ) and number of segments ( $N$ ). Reported runtimes are the mean of 1000 observations per each pair of input parameters ( $P, N$ ), denoted in seconds.

In the space domain, the algorithm requires a matrix of  $P \cdot N$  integers denoting the status {A (available), P (partially combined), E (empty)} of individual segments. In addition to this, for each segments a priority queue of maximum size  $P$  is maintained, wherein each element consists of a single integer denoting process ranks. Greater priority is handed to smaller ranks. Moreover, a matrix of handles to priority queue elements of size  $P \cdot N$  is maintained to perform `heap.erase()` if a match has been found in the search on Line 25. As its return value, the algorithm generates two queues per process: queue I and queue O, the incoming and outgoing queue respectively. The maximum length of the queues is determined by the total number of rounds  $R$  to complete the reduction, where  $R = n + N - 1$ . Each element of the queue is a pair of two integers: the rank of the communication peer and the ordinal number of the segment to communicate.

### Clairvoyant schedule execution

Algorithm 10 generates a per-process personalized schedule consisting of two queues: the inbound and outbound communication queue. Each element  $i$  of the queue defines the inbound and outbound communication peer in round  $i$ , as a pair of two integers (rank,index). The former represents the rank of the peer process and the latter the index of the segment that is to be communicated. The schedule execution algorithm then linearly iterates through the schedule, issuing up to one `MPI_IRecv`

and `MPI_Isend` call per-process in each round and combining at most one segment of size  $m/N$  elements. Before a process proceeds to the next round, all its outstanding MPI calls are completed by a call to `MPI_Wait`. This means that there is no explicit synchronization within the subgroup  $G$  of ready process in round  $i$ . This decision was influenced by the dynamic nature of the subgroup and the non-trivial creation and maintenance cost.

For simplicity, we will hitherto refer to this algorithm as Clairvoyant.

### 6.3 Time complexity of selected algorithms

Table 6.2 (first introduced in Chapter 3) presents the computed time complexity of some well-known reduction algorithms, including those implemented in this study, assuming balanced PATs. The equations in Table 6.2 were derived using the linear cost model, introduced and explained in Chapter 3. It is illustrative to point out that among the listed algorithms, Butterfly will outperform Clairvoyant for some problem sizes  $m$ . In fact, for the data set presented in Fig. 6.5, Butterfly achieves a minimum of 0.96 relative runtime compared to Clairvoyant. However, the range of problem sizes  $m$  for which Butterfly is better or equal to Clairvoyant (Fig. 6.5) is  $[1.7460 \times 10^4 \text{ 4B}, 8.8820 \times 10^4 \text{ 4B}]$ . This represents only 0.697% of the set of problem sizes in Fig. 6.5. In general, the size of the range  $R_B$  where Butterfly outperforms Clairvoyant will depend on the ratio  $r = \frac{\beta}{\gamma}$  and process counts  $P$ . The size of the range  $R_B$  is inversely proportional to both  $r$  and  $P$ .

### 6.4 Analysis of experimental data

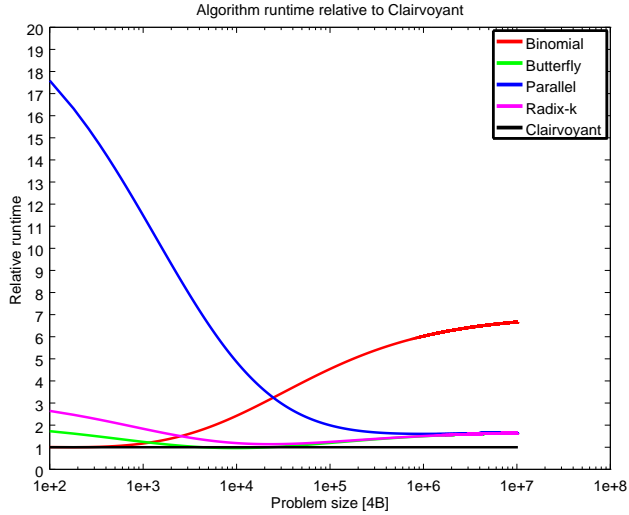
Any analysis of univariate experimental data should begin with an inspection of run sequence plots [21]. These can be instrumental in identifying shifts in location and scale, and to identify outliers. As producing run sequence plots for all the experimental parameters would take up too much space, we will depict only the sample gathered for  $m = 128 \text{ kB}$  and balanced PATs (Fig. 6.6). For ease of analysis, we indicate a measure of central tendency by plotting the sample median. We discuss the choice of the median later in this section.

A quantitative summary of results of the linear regression is displayed in Table 6.3. It can be observed that, despite the slope parameters being almost zero, most samples do not display a statistically significant stationarity of location ( $\alpha = 0.05$ ), except for the sample that was gathered from benchmarked runtimes of algorithm Butterfly. This is because the range of the data is also very small. However, in all cases the location drift is relatively small in value. It should be noted that the least squares linear regression can be skewed by presence of outliers, as was certainly the

**Table 6.2:** Time complexity of some reduction algorithms for homogeneous, fully connected full-duplex networks.

Algorithm	Communication cost (upper bound)	non-comm ops	Source
Binomial Tree	$n[\alpha + \beta m + \gamma m]$	yes	[111, 97]
Butterfly	$2\alpha n + 2\frac{(P-1)}{P}m\beta + \frac{P-1}{P}m\gamma$	yes	[22]
Parallel Ring	$(P-1+n)\alpha + \frac{(P-1)}{P}m(2\beta + \gamma)$	no	[141]
Radix-k	$\sum_{i=1}^r [(k_i - 1)] + n\alpha + \frac{(P-1)}{P}m(2\beta + \gamma)$	yes	[99]
Linear pipeline	$(P-2)\alpha + 2\sqrt{(P-2)\alpha}\sqrt{m(\beta + \gamma)} + \beta m + \gamma m$	yes	[148]
Two-tree	$4(n-1)\alpha + 4\sqrt{(n-1)\alpha}\sqrt{\beta m/2} + m\beta + 2m\gamma$	yes	[164]
Clairvoyant	$(n-1)\alpha + 2\sqrt{(n-1)\alpha}\sqrt{m(\beta + \gamma)} + m\beta + m\gamma$	no	This dissertation

Communication cost is calculated as the time required for the last process to complete execution in the worst case, for balanced PATs. Problem size is denoted by  $m$ , the number of process by  $P$  and  $n = \lceil \log_2 P \rceil$ . It is assumed that  $P = 2^k, k \in \mathbf{N} \setminus 1$ .



**Figure 6.5:** Performance prediction based on the time complexity equations presented in Table 6.2. The x-axis denotes the problem size in multiples of four bytes, while the y-axis denotes the algorithm runtime relative to algorithm Clairvoyant. Parameters  $\alpha = 2.66 \mu\text{s}$ ,  $\beta = 4.8179 \times 10^{-10} \text{ sB}^{-1}$ ,  $\gamma = 1.6654 \times 10^{-10} \text{ sB}^{-1}$ ; experimentally determined on PRACE CURIE supercomputer. For Radix-k, the same radix vector  $\{4, 4, 8\}$  was used for the entirety of the problem size range

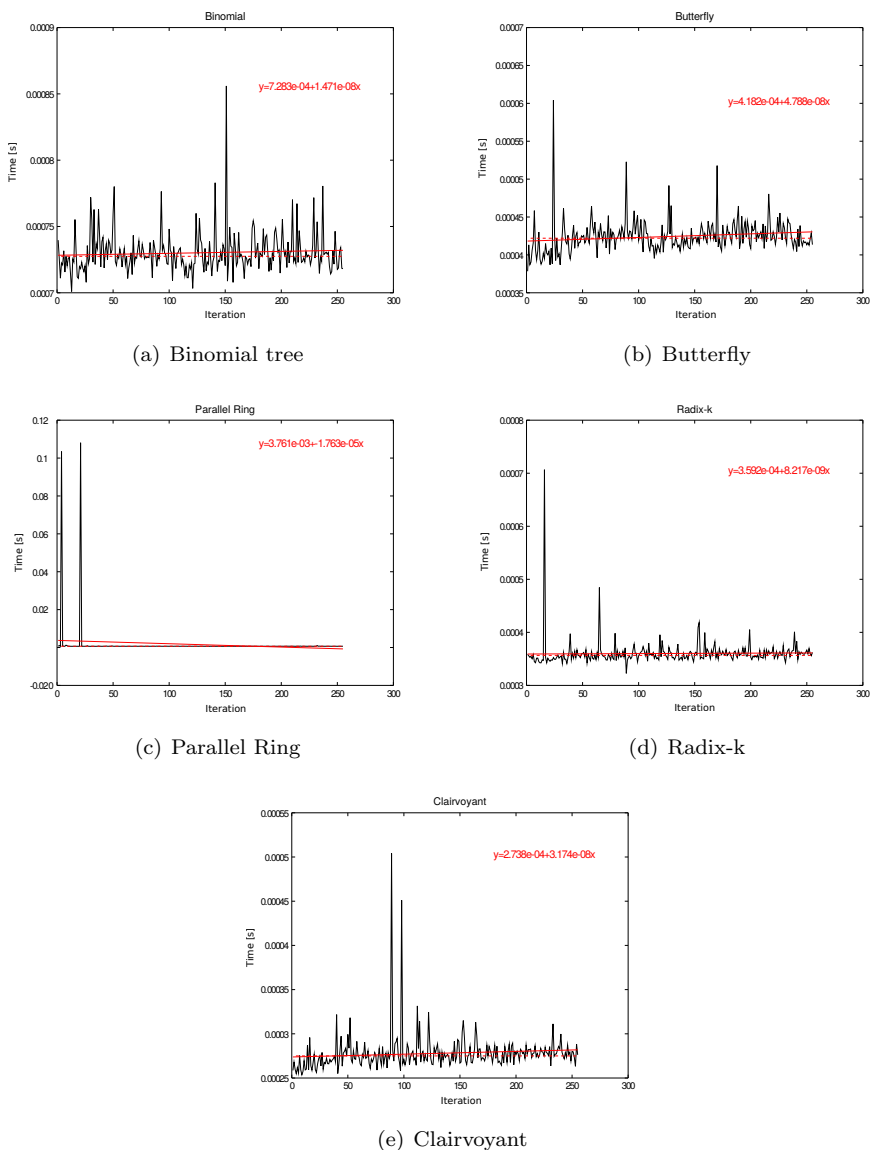
case for the sample gathered from the runtimes of algorithm Parallel Ring (outliers are two orders of magnitude above the median).

**Table 6.3:** Summary of coefficients estimated using the linear regression model  $y = 1 + x_1$  for data depicted in Fig. 6.6. The value tStat is the ratio  $\frac{\text{Estimate}}{\text{SE}}$ . The pValue is the F statistic computed for the null hypothesis test that the corresponding coefficient is zero.

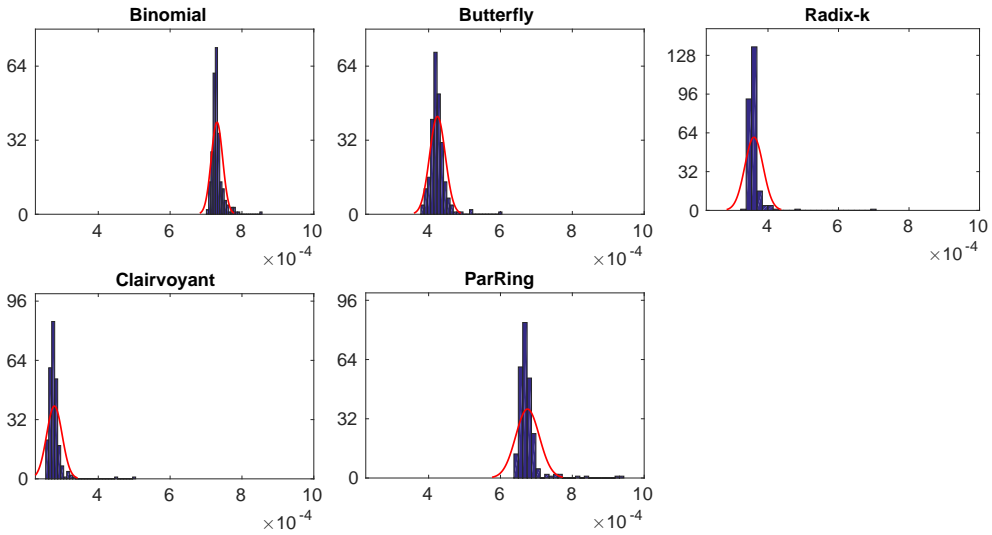
	Estimate	SE	tStat	pValue
<b>Binomial Tree</b>				
Intercept	$7.2828 \times 10^{-4}$	$1.9481 \times 10^{-6}$	$3.7383 \times 10^2$	0
$x_1$	$1.4711 \times 10^{-8}$	$1.3194 \times 10^{-8}$	1.115	$2.6592 \times 10^{-1}$
<b>Parallel Ring</b>				
Intercept	$3.7608 \times 10^{-3}$	$1.1579 \times 10^{-3}$	3.2479	$1.3198 \times 10^{-3}$
$x_1$	$1.7631 \times 10^{-5}$	$7.842 \times 10^{-6}$	-2.2483	$2.542 \times 10^{-2}$
<b>Clairvoyant</b>				
Intercept	$2.7378 \times 10^{-4}$	$2.7008 \times 10^{-6}$	$1.0137 \times 10^2$	$7.3951 \times 10^{-207}$
$x_1$	$3.1737 \times 10^{-8}$	$1.8291 \times 10^{-8}$	1.7351	$8.3937 \times 10^{-2}$
<b>Butterfly</b>				
Intercept	$4.1821 \times 10^{-4}$	$2.6903 \times 10^{-6}$	$1.5545 \times 10^2$	$4.5149 \times 10^{-253}$
$x_1$	$4.7884 \times 10^{-8}$	$1.822 \times 10^{-8}$	2.6281	$9.1125 \times 10^{-3}$
<b>Radix-k</b>				
Intercept	$3.5916 \times 10^{-4}$	$3.2535 \times 10^{-6}$	$1.1039 \times 10^2$	$5.1129 \times 10^{-216}$
$x_1$	$8.2172 \times 10^{-9}$	$2.2034 \times 10^{-8}$	$3.7293 \times 10^{-1}$	$7.0951 \times 10^{-1}$

As was discussed in Chapter 4, not all measures of locations are equally good for all data. Ideally, we want a measure robust in validity and efficiency. In presence of skewed data, the sample mean is not a robust statistic. Fig. 6.7 shows the distribution of algorithm runtime for the sample with problem size  $m = 128$  KiB and balanced PATs, the same sample used to generate Fig. 6.6. We can observe that most distributions exhibit positive skew. In light of the fact that the sample median is less susceptible to outliers in data and is a more efficient estimator of location than the sample mean for data with long tails, we decided to adopt it as the location estimator in our analysis. This decision was further supported by bootstrap uncertainty estimates (Fig. 6.8 and Fig. 6.9 for the mean and median on each of the collected samples, where the sampling distribution of the median was shown to have tighter confidence intervals compared to the sample mean).

Algorithm runtime performance is reported as ratios of the medians of sampled algorithm runtimes, normalized to algorithm Clairvoyant. As there is no simple traditional method to compute the significance test for such statistic, a resampling permutation test method was used instead. For each sample point of interest (algorithm, problem size, imbalance), we performed a permutation test with 20000 iterations. Out of all the generated permutations, we count for each sample point



**Figure 6.6:** Run sequence plots of observed algorithm runtimes for  $m = 128$  KiB. A sample of 256 observations, with the first observation removed, has been modeled with least squares linear regression to detect a trend in data. The dotted red line depicts the median runtime, while the solid red line depicts the linear fit of the data. All measurements were performed on the PRACE CURIE supercomputer;  $P=128$ , one process per node



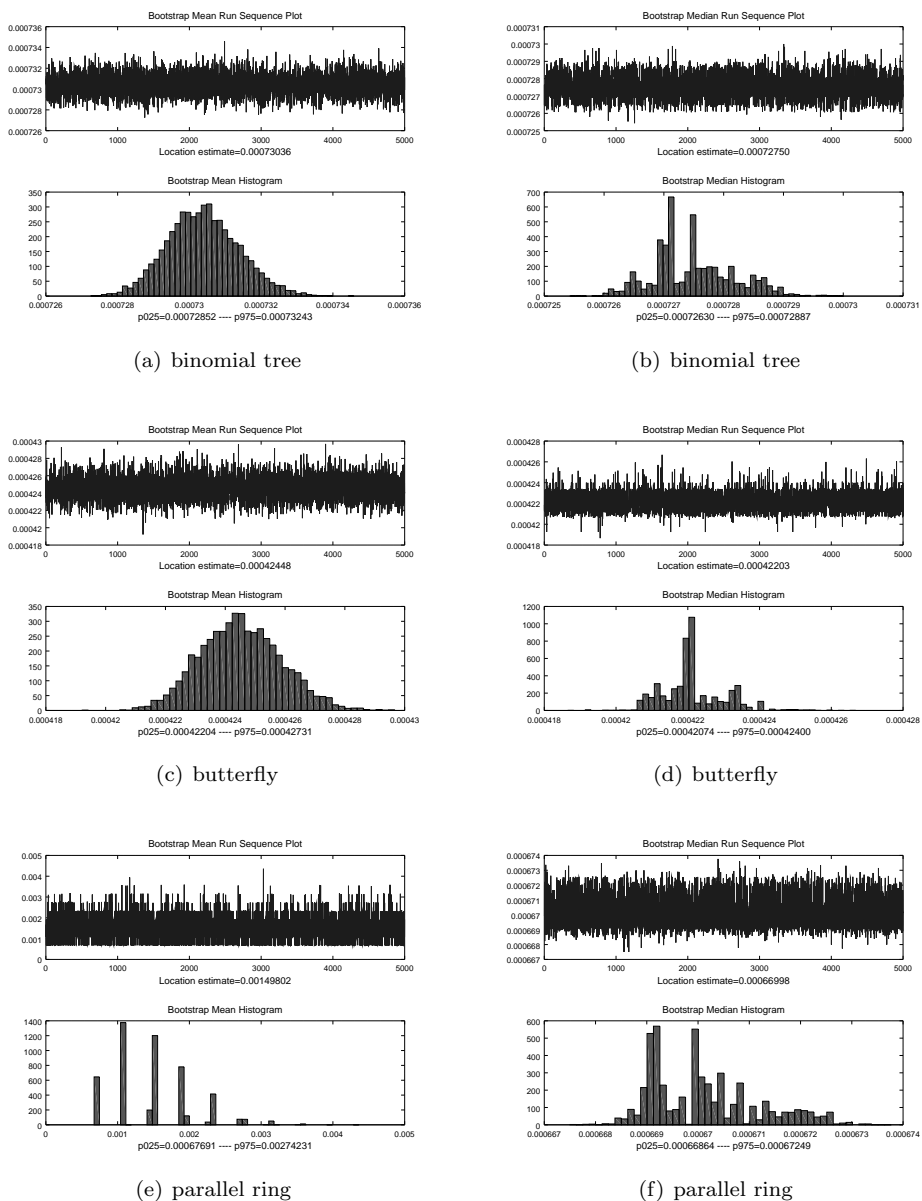
**Figure 6.7:** Distribution of algorithm runtime for balanced PATs and problem size  $m = 128$  KiB. Superimposed on top of each histogram is a fitted normal probability density function. In the plots, the x-axis denotes observed runtime in seconds, while the y-axis denotes the number of observations per bin

of interest how many have resulted with ratios higher than those observed. This gives us an estimated probability of the observed ratio occurring by chance. Then, for each sample point of interest we report the highest such probability among the four algorithms (Clairvoyant is excluded as it is the denominator in the ratios). If the computed probability  $p \geq 0.05$  then we mark the result as statistically not significant.

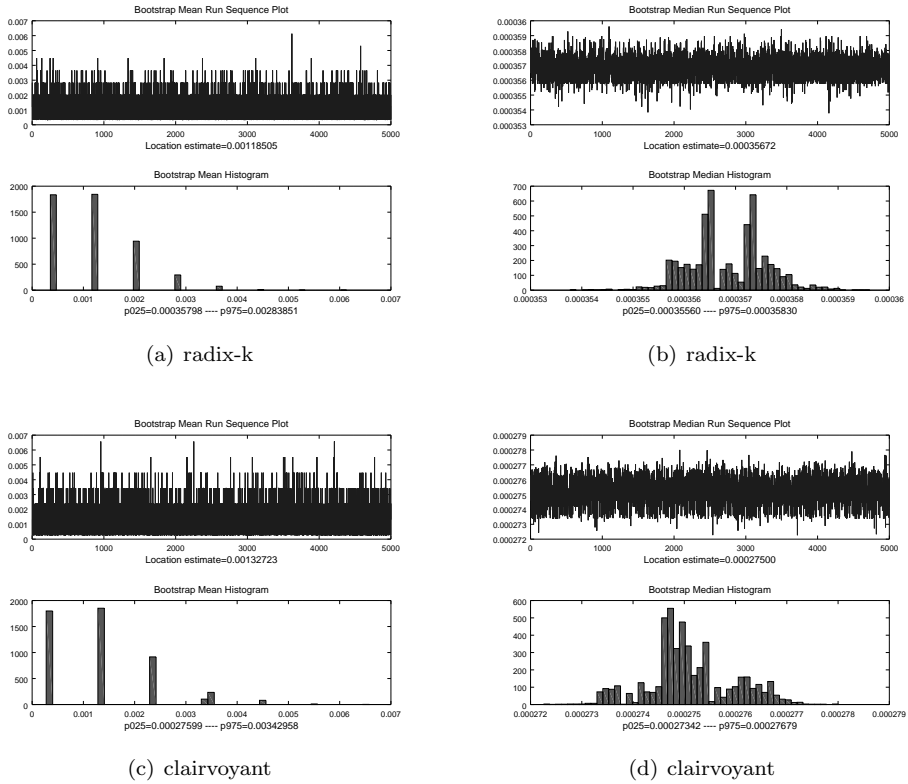
A prerequisite to this approach is that the observations are independent and come from a stationary random generation process. To determine stationarity of random generation processes we used simple quantitative methods such as linear fits to quantify trends and Levene tests to quantify the stationarity of variance. Furthermore, we conducted runs tests on all samples to determine the presence of serial correlation in the gathered data. Table 6.4 shows the results of the runs test analysis. Computing autocorrelograms and spectral plots can be further conducted to verify the presence and nature of the serial correlation in data.

## 6.5 Experiment design

To evaluate algorithm robustness to imbalanced PATs, we selected the PAT pattern where a single process at rank  $P - 1$  was delayed. Proposition 3.3.10 indicates



**Figure 6.8:** Bootstrap plots for the mean and median of observed algorithm runtimes for  $m = 128$  KiB. A sample of 256 observations was resampled for  $B = 5000$  times to produce the plots. Bootstraps of means are displayed in the left column, while those of the medians in the right. All measurements were performed on the PRACE CURIE supercomputer;  $P=128$ , one process per node



**Figure 6.9:** Bootstrap plots for the mean and median of observed algorithm runtimes for  $m = 128$  KiB. A sample of 256 observations was resampled for  $B = 5000$  times to produce the plots. Bootstraps of means are displayed in the left column, while those of the medians in the right. All measurements were performed on the PRACE CURIE supercomputer;  $P=128$ , one process per node

**Table 6.4:** Serial correlation in time series data for each algorithm and each problem size. A runs test was performed on each sample with the confidence level set to 95%. Each value in the table reports the ratio of samples that passed the runs test, i.e. where no significant serial correlation was detected.

Algorithm	128 KiB	512 KiB	2 MiB	4 MiB	40 MiB
Binomial	0.77	0.52	0.63	0.86	0.76
Butterfly	0.72	1.00	0.77	0.86	0.84
Parallel Ring	0.61	0.52	0.63	0.91	0.76
Radix-k	0.88	0.90	0.95	0.82	0.76
Clairvoyant	0.77	0.76	0.77	0.78	0.84

that this is where we should expect to observe the largest absorption time. The experiment was setup as follows: for each message size  $m \in \{128 \text{ KiB}, 512 \text{ KiB}, 2 \text{ MiB}, 4 \text{ MiB}, 40 \text{ MiB}\}$  a different allocation of  $P = 128$  nodes on the PRACE Curie machine was obtained. For each problem size a set of  $k$  absolute imbalances  $I_j, \in \{0 \dots k\}$  was produced so that the magnitudes of these imbalances spanned from  $0t_c$  to roughly  $5t_c$ , where  $t_c$  denotes the runtime of algorithm Clairvoyant. Then for each problem size and each imbalance  $I_j$ , a PAT  $\psi$  was generated, where process at rank  $P-1 = 127$  was delayed for time  $I_j$ . The experiment then proceeded in  $r$  rounds where in each round the five algorithm were executed in succession. The number of rounds  $r$  was 256 for the first two problem sizes, and 100 for the remaining. Message elements were of type `MPI_INT` and the utilized combining operator was `MPI_SUM`.

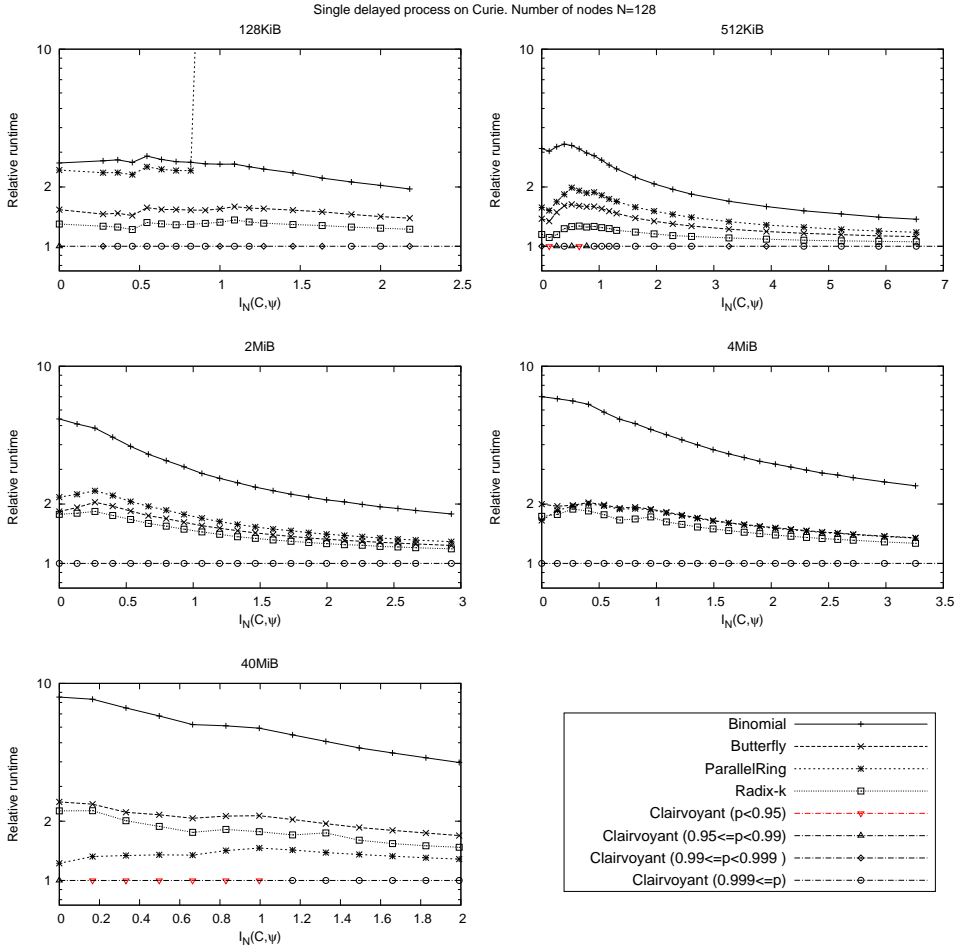
**Table 6.5:** Optimal number of blocks  $N$  as determined by the linear model and empirical measurement.

Method	128 KiB	512 KiB	2 MiB	4 MiB	40 MiB
Linear model	13.2	27.3	54.7	77.39	244.75
Empirical	16	8	16	16	40

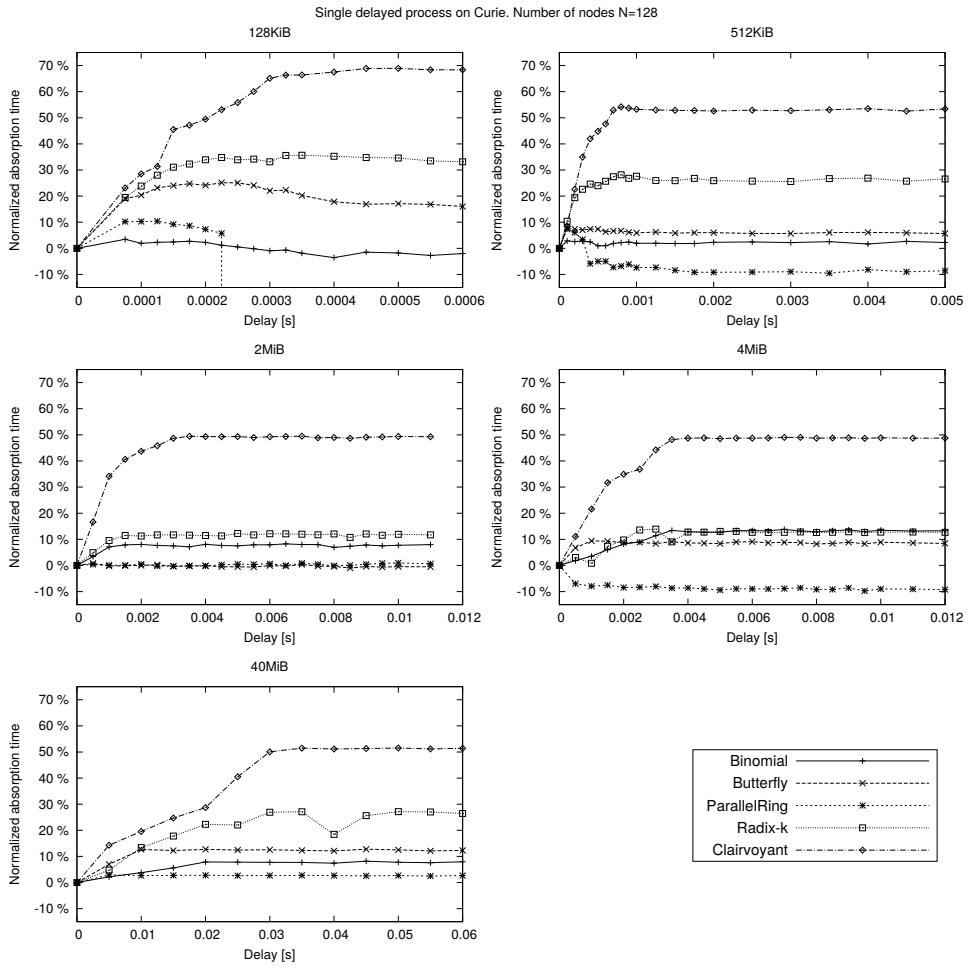
The number of blocks  $N_{opt}$  for each problem size was determined empirically, by measuring the runtime of algorithm Clairvoyant for all  $N_{opt} \in \{2, 4, 8, 16, 32, 64, 128\}$  and selecting that  $N$  which produced the shortest runtime. For  $m = 40 \text{ MiB}$  we added to the set of potential block sizes  $N_{opt}$  the values  $\{40, 80\}$ . The empirically determined value was significantly different from what the linear model predicted, as can be seen from Table 6.5. This is because the parameters  $\beta, \gamma$  are not message size invariant. In fact, the ratio  $\frac{\beta}{\gamma}$  grows with decreasing message size due to larger relative weight of message overheads. This also explains why the empirically determined optimal block sizes were smaller than what the linear model predicted.

## 6.6 Results and discussion

As the results show (Fig. 6.10), algorithm Clairvoyant dominates the surveyed algorithms in performance for all problem sizes and all absolute imbalances. The computed ratios have been shown by permutation tests to be of very high statistical significance (for most data points,  $p \leq 0.001$ ). While this behaviour was predicted (Fig. 6.5), in many cases the observed speedup exceeded the predictions. This is due to shortcomings of the simple linear model where the cost parameters  $\beta, \gamma$  are modeled as message size independent, as discussed in Section 6.5.



**Figure 6.10:** Relative algorithm performance for the PAT where a single process (rank  $P - 1$ ) is delayed. The median observed runtime of each algorithm is normalized to that of algorithm Clairvoyant. To compute the statistical significance of relative runtimes, a permutation test with  $B=20000$  iterations was conducted for each sample pair. The p-values computed by the permutation test are visualized on the line  $y = 1$ . A downward red triangle indicates that, for that particular data point, at least one of the 4 sample pairs did not meet the 95% significance threshold, i.e. that  $p \geq 0.05$ . Circles denote very high significance levels of  $p \leq 0.001$ , while upward triangles and rhombes denote levels in between as depicted in legend.



**Figure 6.11:** Normalized algorithm absorption time with single delayed process (rank  $P - 1$ ).

Observing the transition from balanced PATs to imbalanced, we see one instance of order inversion: for  $m = 4$  MiB algorithm Parallel Ring falls behind Radix-k for imbalanced PATs, while it was faster for balanced PATs. In general, however, we can state that the ordering observed for balanced PATs holds with increasing levels of imbalance. This would imply that the existing collective operation benchmarks could be used to determine the best performing algorithm even for imbalanced PATs.

Algorithm Radix-k consistently outperforms Butterfly for all tested problem sizes, contrary to/with the linear model prediction. This leads us to believe that latency plays a smaller role than that used to model the time complexity of algorithm Radix-k. For  $m = 40$  MiB, the runtime of algorithm Parallel Ring is significantly lower than the prediction.

To understand better how each algorithm responded to PAT imbalance, we have plotted the normalized absorption times in Fig. 6.11. A surprising result was that the observed normalized absorption for algorithm Clairvoyant was higher than the prediction indicated (Fig. 6.1).

For  $m = 40$  MiB the maximum observed normalized imbalance  $I_N$  was 51%, compared to 14.9% the linear model would predict. As one of its input parameters, algorithm Clairvoyant takes the time  $t_s$  to send/receive and combine one segment of size  $m/N$ . For  $m = 40$  MiB, we empirically determined that  $t_s = 6.43 \times 10^{-4}$  s. For  $I(\psi) = 60$  ms, the generated schedule length  $R = 133$ .

From proposition 3.3.10 and the fact that  $P = 2, N = 40 \Rightarrow R' = 40$ , we can determine whether the schedule length  $R = 133$  is consistent by verifying that  $t_s(R - R') = I(\psi)$ . Since  $t_s(133 - 40) = 0.0598$  s  $\approx I(\psi)$  we can conclude that the algorithm performed as expected, producing a schedule of minimum length.

However, the expected time  $t_C(128, \psi) = Rt_s = 0.0855$  s is greater than the observed runtime of 0.0747 s. From the observed runtime and the fact that  $I(\psi) = 60$  ms we can empirically estimate the time required to perform a 2-way reduction as  $t_C^e(2, \psi) = 0.0146$  s. The empirically determined runtime is considerably shorter than the projected time:  $t_C(2, \psi) = 40t_s = 0.02572$  s. It would seem that with only two processes communicating, the network bandwidth is 176% that of the bandwidth with 128 processes communicating concurrently.

The schedule generation time for  $N = 16, P = 128$  was on average 0.55 ms (Table 6.5). Extrapolating the empirically determined schedule computation time (Fig. 6.4), leads us to conclude that in real time usage scenarios, the algorithm will be competitive whenever  $m \geq 1$  MiB.

### 6.6.1 Catastrophic slowdown

An interesting phenomenon was observed with algorithm Parallel Ring for  $m = 128$  KiB. For  $I(\psi) \geq 0.2$  ms the algorithm experienced a catastrophic slowdown of two orders of magnitude. Its time series data oscillated between high and low values,

with excursions into high value territory becoming more prominent with increasing absolute imbalance (Fig. 6.12). We reproduced this behaviour by re-running the experiment for  $m = 128$  KiB on a different day with a different allocation of compute nodes.

Autocorrelograms of the time series data (6.13) depict an alternating sequence of positive and negative spikes, slowly decaying to zero and well into statistically significant territory. This makes for a strong argument that the observed phenomenon has a systematic cause, either because of some system interference or the underlying nature of the native MPI implementation.

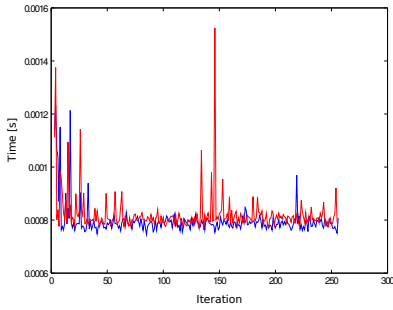
As of the time of writing this dissertation, we have no conclusive explanation for the observed phenomenon. The fact that this behaviour was observed only for the problem size  $m = 128$  KiB, but not for larger problem sizes indicates that the reason might lie in the smaller segment size of  $B = 1$  KiB and the possibility that the former were communicated with the eager protocol, while the latter with the rendezvous communication protocol. Were former the case, then for sufficiently large absolute imbalances, up to  $P - 1$  messages of size  $B = m/P$  would be sent from process rank 0 to rank  $P - 1$ . We conjecture that the manner in which these unexpected receives were handled by the implementation could explain the observed phenomenon.

## 6.7 Summary

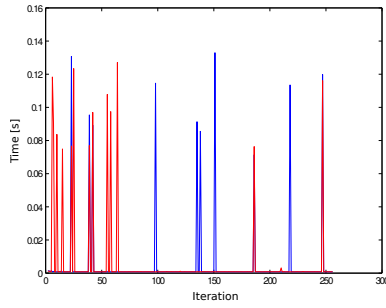
The work presented in this chapter has provided a much needed insight into the performance of MPI reduction algorithms under the presence of imbalanced process arrival times and introduced a novel, clairvoyant segmenting reduction algorithm.

We found that this algorithm universally outperforms all selected reduction algorithms in this study, even for balanced PATs. This is a very interesting and perhaps surprising result. For some problem sizes, the algorithm was found to be nearly twice as fast as the next fastest algorithm. The peak performance of the algorithm is contingent on full knowledge of PATs. Using this side-information, the algorithm constructs an optimized communication schedule prior to the execution of the reduction operation. If the schedule generation is performed at the time the collective operation is invoked, than the speedup obtained outweighs the construction costs for problem sizes larger or equal to 1 MiB. Otherwise, the algorithm can be used in iterative settings where the schedule can be precomputed once and reused multiple times.

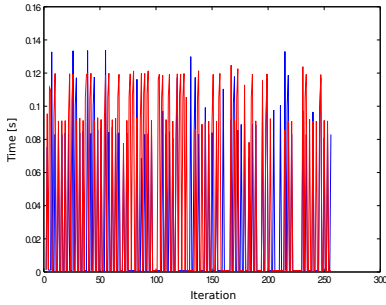
Our results indicate that, excluding the Clairvoyant algorithm, reduction algorithms have little to no resiliency to skewed PATs. Another interesting finding is that the ordering of algorithm runtime observed for balanced PATs appears to hold with increasing levels of imbalance. This is reassuring, as all known benchmarks used to optimize MPI collective operation performance ensure that PATs are balanced.



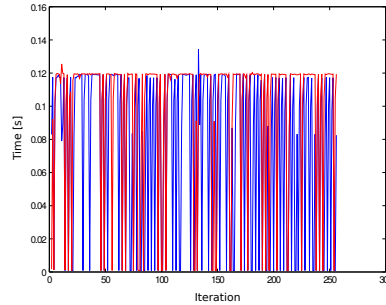
(a) Parallel ring,  $I(x) = 0.175$  ms



(b) Parallel ring,  $I(x) = 0.2$  ms

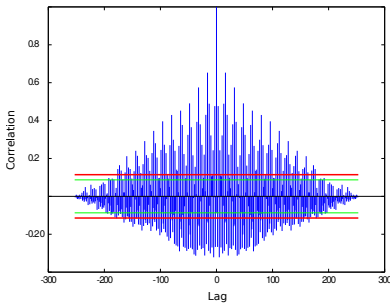


(c) Parallel ring,  $I(x) = 0.225$  ms

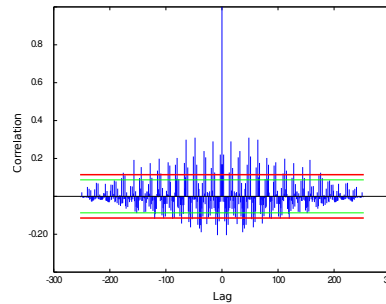


(d) Parallel ring,  $I(x) = 0.25$  ms

**Figure 6.12:** Time series data of two independent samples of algorithm Parallel Ring. Blue color denotes sample 1 and red sample 2. Axis x denotes iteration number, while axis y denotes observed algorithm runtime in seconds.



(a) Parallel ring,  $I(x) = 0.25$  ms. Sample 1.



(b) Parallel ring,  $I(x) = 0.25$  ms. Sample 2.

**Figure 6.13:** Autocorrelation of time series data of two independent executions of algorithm Parallel Ring. Green line denotes the 95% confidence level, while the red denotes the 99% confidence level.

However, one algorithm, Parallel Ring, exhibited a catastrophic slowdown of 2 orders of magnitude, for problem size  $m = 128$  KiB and increasing magnitudes of imbalance. This result was reproduced by re-running the experiment on a different day and a different allocation of compute nodes.

These findings have important implications on High Performance Computing (HPC) applications as they show that an imbalance resilient reduction algorithm can be produced to consistently outperform reduction algorithms found in library implementations of the MPI programming interface.

A potentially interesting extension of this work would be to devise an agnostic-dynamic non-atomic reduction algorithm. We suspect that the communication graph re-ordering could be done along the lines of desynchronization algorithms used in wireless sensor networks [19, 33, 170, 32].



## Chapter 7

# Comparative analysis with non-blocking reduction operations

---

In this chapter, we present a comparative experimental analysis of the algorithms introduced in Chapter 5 and Chapter 6 with the non-blocking collectives, that are part of the Message Passing Interface (MPI) standard since version 3.0 [125]. The potential of non-blocking collectives to amortize the communication latency of collective operations by overlapping them with independent computation, presents them as an interesting alternative in mitigating the impacts of imbalanced Process Arrival Times (PATs). Non-blocking collectives are already becoming an established method in mitigating the effects of system noise [47, 188].

Non-blocking collectives are operationally similar to non-blocking Point-to-point (p2p) operations. As with the latter, a call to a non-blocking collective operation initiates an operation that must be completed in a separate completion call. Their principal benefit is clearly stated in the standard: “Once initiated, the operation may progress independently of any computation or other communication at participating processes. In this manner, nonblocking collective operations can mitigate possible synchronizing effects of collective operations by running them in the “background.” In addition to enabling communication-computation overlap, nonblocking collective operations can perform collective operations on overlapping communicators, which would lead to deadlocks with blocking operations. Their semantic advantages can also be useful in combination with point-to-point communication.” [125]. The performance benefits of non-blocking collective operations have been argued and backed up by numerous papers since the mid 2000s [78, 80, 57, 47, 188]. When supported by modern Network Interface Controllers (NICs) with programmable hardware for computation offload [130], a high degree of computation-computation overlap can be achieved.

## 7.1 LibNBC

For this comparative study, we chose the LibNBC [74, 73, 72] library as the principal implementation of the non-blocking collectives against which we will compare our imbalance resilient algorithms. LibNBC is an open source implementation of non-blocking collectives and has served as a prototypic implementation of a non-blocking interface for MPI, prior to the inclusion of non-blocking collectives in the standard. It is based on MPI-1 and written in ANSI C. A strong motivation behind our choice was the ease of inspection and modification of the source code of the library to better suit our microbenchmarking requirements.

Inspecting the `MPI_IReduce` implementation of LibNBC revealed that the library implements this operation with two algorithms: Binomial Tree and Linear Pipeline. The former is an algorithm optimized for small problem sizes and atomic input data, while the latter is an algorithm optimized for large problem sizes, non-atomic input data and small to moderately large communicator sizes. Both of the algorithms have been discussed in detail in Chapter 3. We made the decision to take advantage of these underlying implementations by leveraging the native Binomial Tree implementation for the comparative study for atomic input data and the native Linear Pipeline algorithm for the comparative study for non-atomic input data.

The core of LibNBC is the collective schedule data structure. This schedule is generated on the fly by the LibNBC implementations of the collective operations and executed by the library runtime. As defined by the LibNBC technical report [71]: “A collective schedule is a process specific “execution plan” for a collective operation. It consists of all necessary information to perform the operation ... A schedule can consist of multiple rounds to model the data dependencies. Operations in round  $r$  may depend on operations in rounds  $i \leq r$  and are not executed by the scheduler before all operations in rounds  $j < r$  have been finished.”

The technical report states that through this schedule design it is entirely possible to implement transparent segmentation and pipelining.

## 7.2 Non-blocking collective operation microbenchmark

The LibNBC runtime uses handles to identify outstanding (active) collective operations and all the necessary information to ensure their progress. The implementation of LibNBC requires the user to explicitly call `NBC_Test` to progress the operation from one round to the other. However, how often the user should call the test function is not clear-cut. Calling `NBC_Test` too often will result with overhead, while not calling it at all will result with no overlap.

In our implementation of the comparative performance analysis, we adopted an approach similar to what the authors used in NBCBench [74, 73]. Thus, for each non-blocking operation call, we would issue  $N$  calls to `NBC_Test` for every *interval* bytes, where  $N$  is a function of input data size  $m$ , computed as:

$$N = \lfloor \frac{m}{interval} \rfloor + 1$$

The benchmark was setup as follows. We selected the problem size  $m = 512$  KiB and distinguished two cases: atomic input data and non-atomic input data. The microbenchmark evaluated each algorithm by performing the following four steps in succession:

1. imbalance pattern injection
2. collective reduction operation invocation
3. local computation of preset duration.
4. Blocking wait for the operation in Step 2 to complete

For each algorithm, we measured the time from the first process arriving at Step 1 till the last process exiting at Step 4.

In one iteration of the microbenchmark, each algorithm is subjected to an identical PAT vector to produce one observation per algorithm. This procedure is repeated for  $r$  iterations to generate one sample of  $r$  observations per algorithm. Depending on the type of PAT imbalance, the PAT vector in each iteration may or may not be the same.

In this experiment, we generated two types imbalanced PAT vectors and set  $r = 100$ . In the first type, a single process was randomly selected in each of the  $r$  iterations, and delayed for the preset time  $I(a)$ . In the second type, every odd ranked process was delayed for time  $I(a)$ . Thus, in the first type, the PAT vector was allowed to differ from one iteration to the next, while in the second type, all the PAT vectors were identical. The set of absolute imbalances was the same for both types of PATs and equal to

$$\Delta = \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.75, 2.0, 2.25, 2.5, 2.75, 3.0, 3.5, 4.0, 4.5, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0\} [ms].$$

Immediately after the invocation of the collective operation, a local computation of duration  $I(a) \cdot \sigma$  was started (Step 3). The values of the overlap fraction  $\sigma$  were selected in sequence from the vector  $\Sigma = \{0, 0.25, 0.5, 1.0\}$ . Thus, for each point in the space  $\Delta \cdot \Sigma$  we produced one sample of  $r = 100$  observations. The procedure to execute the local computation with overlap is provided in Listing 7.1.

**Listing 7.1:** Local computation with overlapped communication progress (adopted from LibNBC-1.1.1 [73])

```
double docompute_test(double computetime, double numTests,
                    NBC_Handle *handle, MPI_Status * status,
                    bool callTest)
{
    double testsum=0;
    double end;
    double time, testint, nextttest;
    int ret;

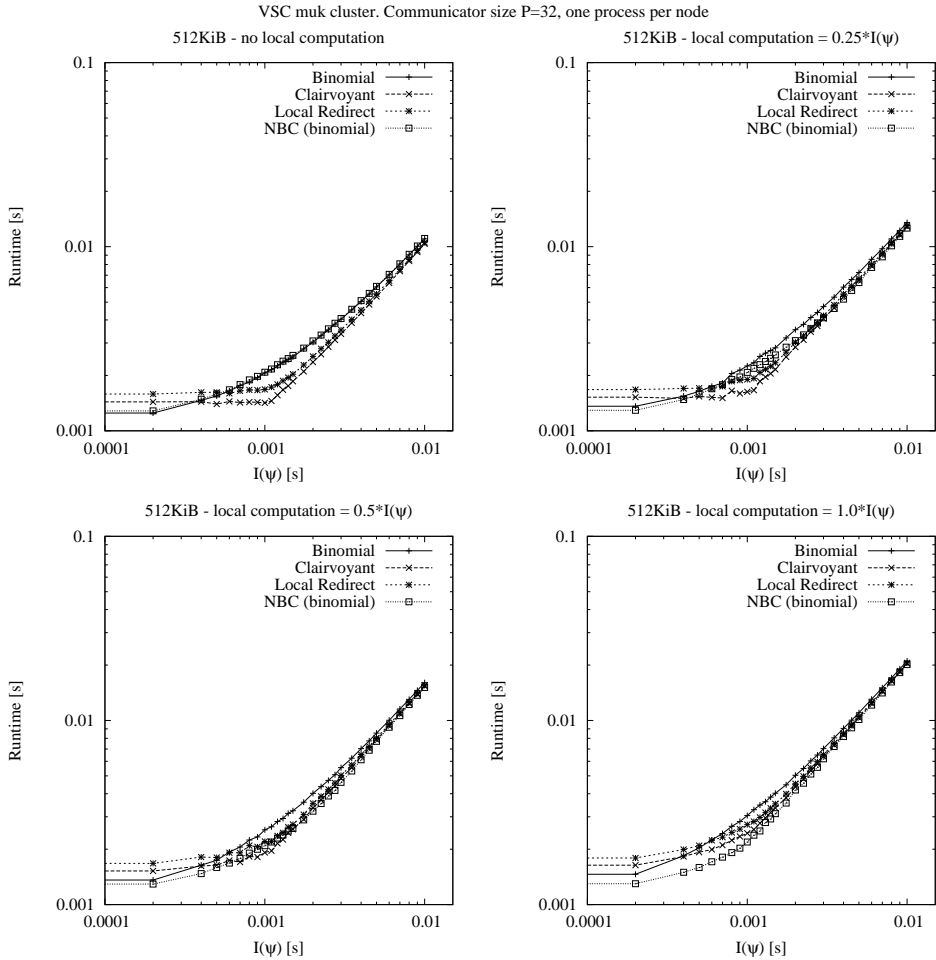
    testint = computetime/numTests;
    time = MPI_Wtime();
    end = time + computetime;
    nextttest = time + testint;
    while(time < end) {
        time = MPI_Wtime()-testsum;
        if(time > nextttest && (handle != NULL)) {
            nextttest = time + testint;
            testsum -= MPI_Wtime();
            NBC_Test(handle, &ret, status);
            testsum += MPI_Wtime();
        }
    }
    return testsum;
}
```

## 7.3 Experimental results

Using the benchmark, we conducted a comparative performance evaluation on the VSC muk cluster. This cluster was described in detail in Chapter 4. For the evaluation, a communicator of size  $P = 32$  was used throughout, and one process was assigned per compute node.

For atomic input data, we selected four algorithms: Binomial Tree, Local Redirect, Clairvoyant and NBC non-blocking binomial tree. For non-atomic input data, we selected the following four algorithms: Butterfly, Parallel Ring, Clairvoyant and the LibNBC non-blocking Linear Pipeline. In addition to these algorithms, we also included the native `MPI_IReduce` implementation from the Intel MPI library, version 5.03 .

The experimental results for the atomic input data and the PAT where a single randomly selected process was delayed is shown in Fig. 7.1. From the figure, we can observe several points. First, when no local computation is present, the performance



**Figure 7.1:** Observed runtimes for the PAT where a single randomly selected process was delayed. Communicator size  $P = 32$  and atomic input data of size  $m = 512$  KiB.

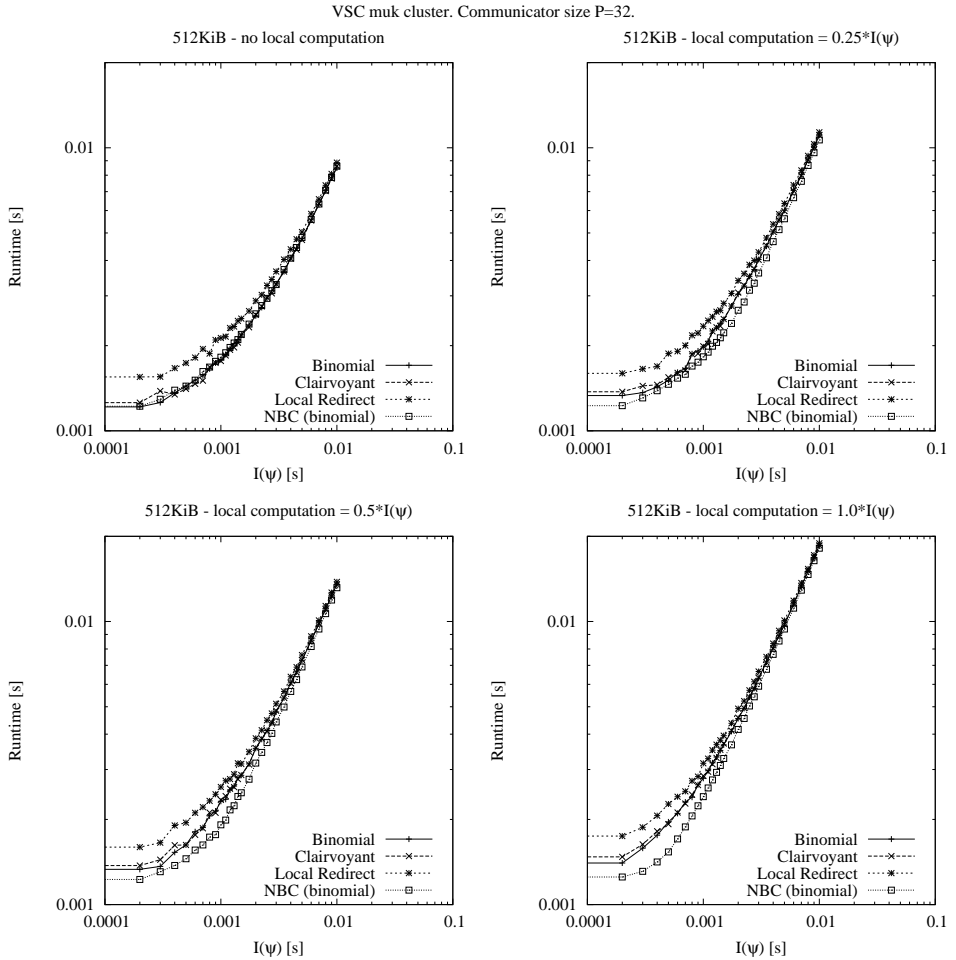
of our Binomial Tree implementation and that of the LibNBC Binomial Tree implementation are roughly identical, as expected. The performance of algorithm Clairvoyant is initially less than that of these two algorithms (due to overheads discussed in Chapter 5), but once  $I(\psi) > 0.4$  ms, the algorithm emerges as the best performing. Similarly, algorithm Local Redirect overtakes in performance the Binomial Tree implementations when  $I(\psi) > 0.6$  ms. However, with increasing magnitude of the absolute imbalance, the relative performance of the four algorithms becomes indistinguishable. This is exactly the same behaviour as observed and discussed in Chapter 5.

Once the local computation is introduced, we can begin to observe a divergence in performance of our Binomial Tree implementation and that of the LibNBC library. The non-blocking implementation is beginning to take advantage of the opportunity to overlap the reduction operation with the local computation. However, algorithm Local Redirect and Clairvoyant are still better in performance, until the local computation begins to constitute half or more of the absolute imbalance. When the local computation matches in magnitude the absolute imbalance, the LibNBC Binomial Tree implementation is never dominated by the other three algorithms.

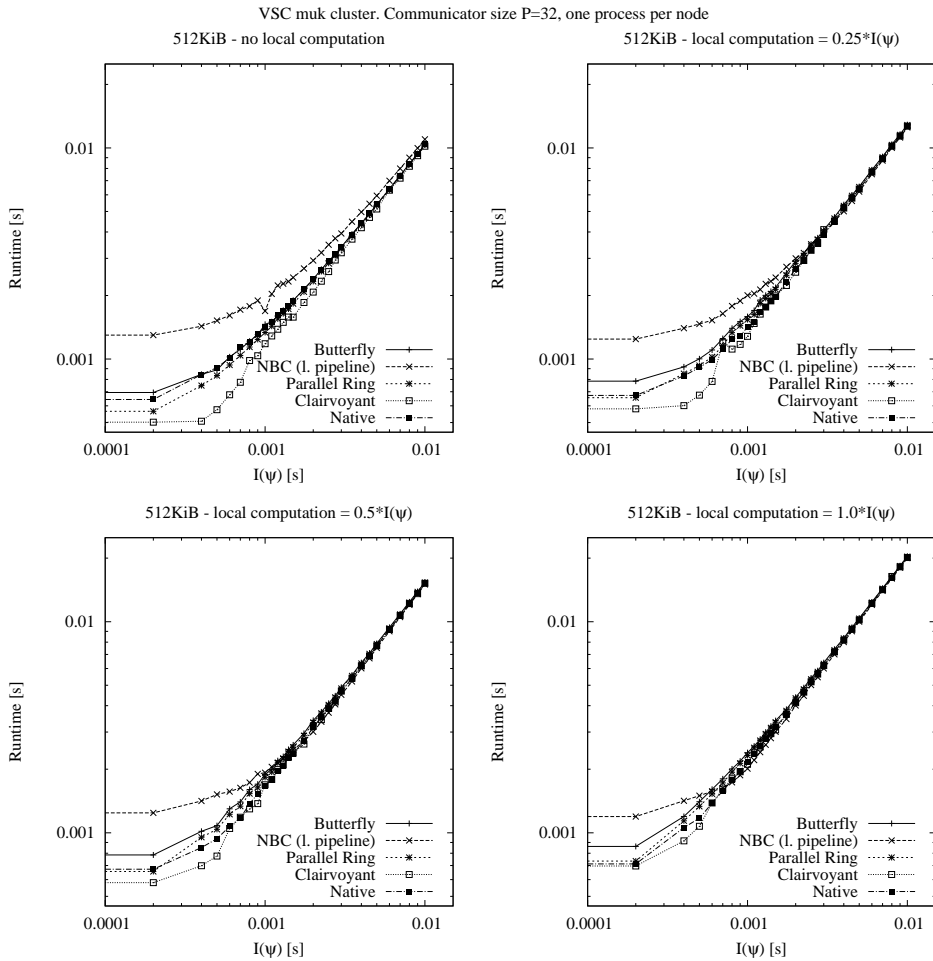
The experimental results for the atomic input data and the PAT where every odd ranked process was delayed is shown in Fig. 7.2. This is a PAT pattern where imbalance robust algorithms have no opportunity to mitigate the absolute imbalance. This is clearly reflected in the performance of algorithm Local Redirect, for which this pattern was identified to constitute the worst case (5.2.4). For this PAT pattern, the LibNBC Binomial Tree becomes non-dominated as soon as local computation is introduced.

The experimental results for the non-atomic input data and the PAT where a single randomly selected process was delayed is shown in Fig. 7.3. When no local computation is present, algorithm Clairvoyant is not dominated in any of the samples. This closely reflects what was observed on the PRACE CURIE supercomputer and discussed in Chapter 6. The performance of the LibNBC Linear Pipeline implementation is surprisingly far behind the other algorithms. This could likely be attributed to a poor implementation choice of segment size. Once the local computation is introduced, the non-blocking algorithms begin to approach algorithm Clairvoyant in performance and finally overtake it once  $I(\psi) \geq 1$  ms. However, by that point the relative performance difference is very small. We can observe that relative performance of non-blocking algorithms for non-atomic input data is inferior to that for atomic input data. This can be explained by the fact that in the non-atomic input data reduction algorithms (with segmentation), processes are spending relatively more time in computation (due to better load balancing) and there is less opportunity for overlap, compared with the atomic input data algorithms.

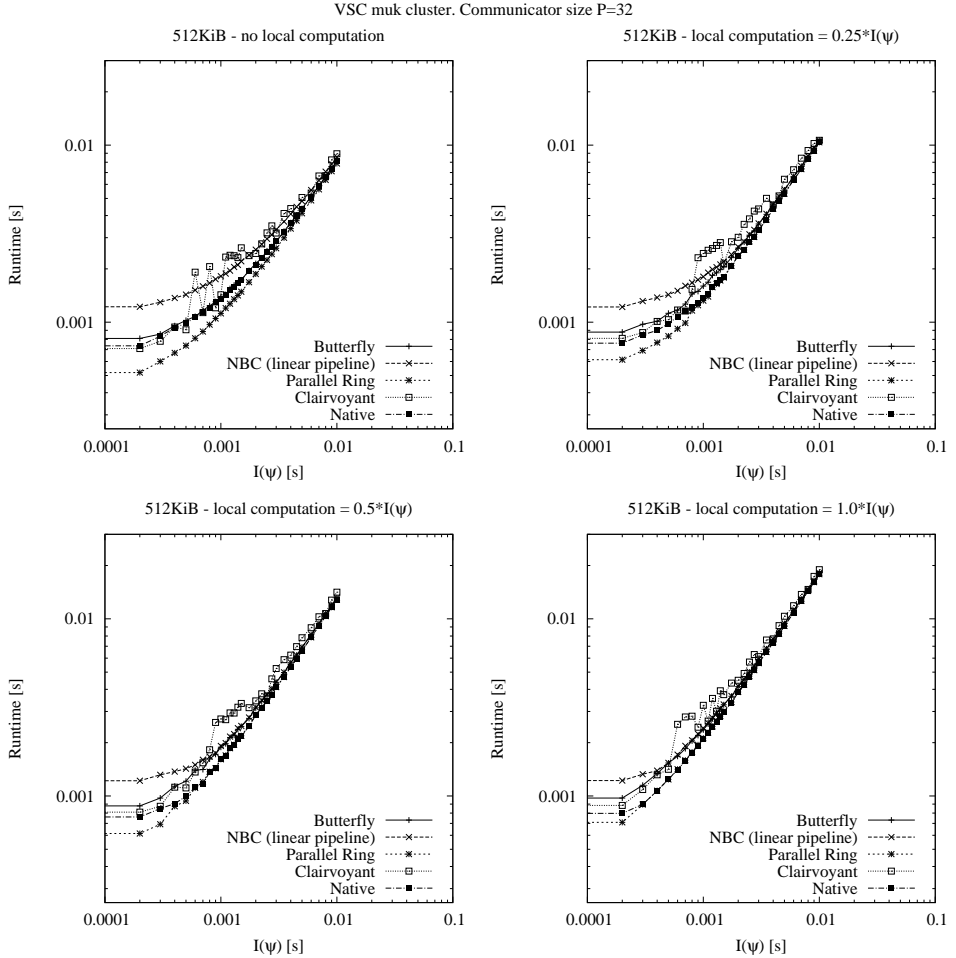
The experimental results for the non-atomic input data and the PAT where every odd ranked process was delayed is shown in Fig. 7.4. Again, this is the pattern where



**Figure 7.2:** Observed runtimes for the PAT where every odd ranked process was delayed. Communicator size  $P = 32$  and atomic input data of size  $m = 512$  KiB.



**Figure 7.3:** Observed runtimes for non-atomic input data the PAT where a single randomly selected process was delayed. Communicator size  $P = 32$  and non-atomic input data of size  $m = 512$  KiB.



**Figure 7.4:** Observed runtimes for the PAT where every odd ranked process was delayed. Communicator size  $P = 32$  and non-atomic input data of size  $m = 512$  KiB.

imbalance robust algorithms are expected to have no ability to mitigate the impact of the absolute imbalance in the PATs, which is clearly reflected in the observed performance. The fact that even in presence of local computation, the Intel MPI implementation of `MPI_IReduce` fails to exceed the performance of the Parallel Ring algorithm indicates that either the library implementation is poorly tuned to the cluster machine or that its algorithmic implementation is sub-optimal.

## 7.4 Summary

In this chapter we presented a short comparative performance analysis of the imbalance robust reduction algorithms introduced in Chapter 5 and Chapter 6 against the non-blocking algorithms found in the implementations of the LibNBC and Intel MPI libraries.

Our findings show that when the input data is atomic, the local (independent) computation has to represent at least one half of the absolute imbalance for the non-blocking algorithms to match or exceed in performance the imbalance robust algorithms. When input data is non-atomic, there appears to be less opportunity to overlap computation with communication, and consequently the non-blocking algorithms do not measurably improve upon the imbalance robust algorithms.

In conclusion, we can comfortably state that non-blocking collectives, at least in case of `MPI_IReduce` could provide a viable alternative to imbalance robust algorithms when there is sufficient opportunity for computation-communication overlap. This comes with the caveat that to achieve this, it might be necessary to perform extensive changes to legacy algorithms to extract independent computations. This is not guaranteed to be always possible. On the other hand, imbalance robust algorithms have the advantage of providing imbalance resiliency without any change to legacy code. Yet, they require modifications to the library runtime, and depending on the algorithm require that the PAT patterns be traced, modelled and the computation schedules pre-constructed, all of which brings non-trivial overheads.

## Chapter 8

# Epilogue

---

### 8.1 Conclusions

This dissertation presented an overview of my efforts and contributions to the domain of message passing algorithms by virtue of several new implementations of the Message Passing Interface (MPI) collective reduction operation. The work herein performed had as its principal objective the performance improvement of High Performance Computing (HPC) applications running on computer clusters and supercomputers worldwide. This was done through the optimization of the collective MPI reduction operation.

Traditionally, implementations of MPI collective operations were optimized only with the assumption that the Process Arrival Times (PATs) at the collective operation call site are balanced, i.e. that all the participating processes commence the operation simultaneously. In fact, all microbenchmarks would go to great measures in ensuring such starting conditions. Yet, the reality is that balanced PATs are an uncommon occurrence in the execution life of HPC applications [6, 139]. This leads to the grave implication that potentially all implementations of collective operations in the various library implementations of MPI may not be optimized for the conditions that routinely occur in the real world.

The approach undertaken in this dissertation was to design and implement collective reduction operations that would be robust to imbalanced PATs. In doing so, we have adopted two strategies: one contingent on the capacity to predict the PAT pattern at the moment of collective operation invocation and the other which is agnostic of PAT patterns. The former resulted in an optimal reduction algorithm for atomic input data, and near optimal algorithm for non-atomic data, assuming a fully connected homogeneous network with no communication-computation overlap. The latter produced a dynamic algorithm that was shown to be a clear improvement over the binomial tree reduction algorithm, which is the optimal reduction algorithm for atomic input data and balanced PATs.

The first contribution of the dissertation is technical. To measure and evaluate the performance of the novel reduction algorithms it was necessary to construct a new collective operation microbenchmark with the ability to both inject a wide range of imbalance patterns and to accurately measure collective operation execution time. The benchmark and the rationale behind it have been elaborated in Chapter 4.

The first scientific contribution of the dissertation is the optimal reduction algorithm for atomic input data. The algorithm, called *clairvoyant* pre-constructs a reduction schedule using the knowledge of the PAT pattern at the moment of collective operation's invocation. A proof of the algorithm's optimality under the assumed network properties was derived. The implementation of this algorithm was evaluated against the Binomial Tree algorithm for a range of imbalanced process arrival times on a large supercomputer machine. The implementation dominated in runtime for almost all test parameters (the chief of which were problem size and imbalance magnitude). A maximum measured speedup of 1.67 was observed compared to the Binomial Tree algorithm. This was close to the maximum possible speedup of 1.75 that the theoretical analysis predicted for  $P = 128$ . The observed performance gap was shown to be a consequence of memory allocation overheads, in absence of which the algorithm would have met the theoretical predictions. This was achieved despite the heterogeneous nature of the utilized network. For balanced PATs, the algorithm performed roughly the same as the Binomial tree reduction algorithm, as was expected. We conjecture that this algorithm could with minimal modifications be used to implement an optimal broadcast algorithm for atomic input data.

In contrast to the static clairvoyant algorithm, we have designed and implemented a dynamic schedule algorithm for atomic input data, *Local Redirect* and analyzed it in detail in Chapter 5. This algorithm is agnostic of PAT patterns and thus applicable to circumstances where PATs are unpredictable. It also has the fundamental advantage that it is equally applicable to commutative and non-commutative combining operators. The observed runtime of the algorithm was consistently smaller than that of the Binomial Tree algorithm for most of the evaluated imbalance magnitudes. However, for balanced PATs the algorithm was strictly worse in performance.

The third scientific contribution of the dissertation is the near-optimal reduction algorithm for non-atomic input data. This algorithm, just like the clairvoyant algorithm presented in Chapter 5 pre-constructs the reduction schedule by utilizing the pre-knowledge of the PAT pattern that is to occur at the collective operation call site. It uses a greedy schedule construction strategy that endeavours to keep the early arrived processes busy by exchanging and combining input data segments. Due to equi-segmentation of the input data, this algorithm cannot produce optimal schedule lengths for all problem parameters. Yet, the experimentally observed performance was universally superior to that of the five state-of-the-art algorithms included in

the comparative study, as is elaborated in Chapter 6. If the schedule generation is performed at the time the collective operation is invoked, than the speedup obtained outweighs the construction costs for problem sizes larger or equal to 1 MiB. Otherwise, the algorithm can be used in iterative settings where the schedule can be precomputed once and reused multiple times.

A noteworthy result of this research is that current implementations of the collective reduction operations have little to no resiliency to imbalanced PATs. Similar results have been reported by researchers for other collective operations, giving a clear impetus to the design and production of imbalance robust collective operations. The work presented in this dissertation represents one small contribution to this objective.

Finally, we compared the imbalance robust reduction algorithms presented in this dissertation with the established non-blocking reduction algorithms. We found that the non-blocking algorithms can offer a comparable and sometimes even superior resiliency to imbalance. However, this was achieved only for atomic input data algorithms and high ratios of independent computation to collective operation runtime. For non-atomic input data, imbalance robust algorithms were found superior to the non-blocking algorithms. While the algorithms presented in this dissertation can be transparently put to use in legacy code to bring about resiliency to imbalanced PATs, the same cannot be said of non-blocking collectives. The latter require a significant modification of the application kernels to extract local, independent computations that can be overlapped with the communication.

## 8.2 Future work

While the work presented in this dissertation answers several important questions, it leaves equally important questions unanswered. Chief of those being how applicable clairvoyant algorithms are in real-world HPC applications. The ability to accurately predict PAT patterns at collective operation call sites would require the derivation of a good-enough stochastic model. What constitutes good-enough is a very relevant question, as short of divine prescience it is inconceivable that perfect knowledge of PATs would be available to the library runtime. In this manuscript, we have presented only a preliminary study that showed how a simple moving average model could be used to fit the observed PATs for one particular application run. The results of this study are encouraging and warrant further experimental effort.

The reduction algorithms herein presented all assume commutative combining operators, yet some applications invoke reduction operations with custom non-commutative operators. It is not immediately clear whether it would be possible to derive imbalance robust algorithms in this case, where the ability to perform “out-of-order” communication is significantly more constrained.

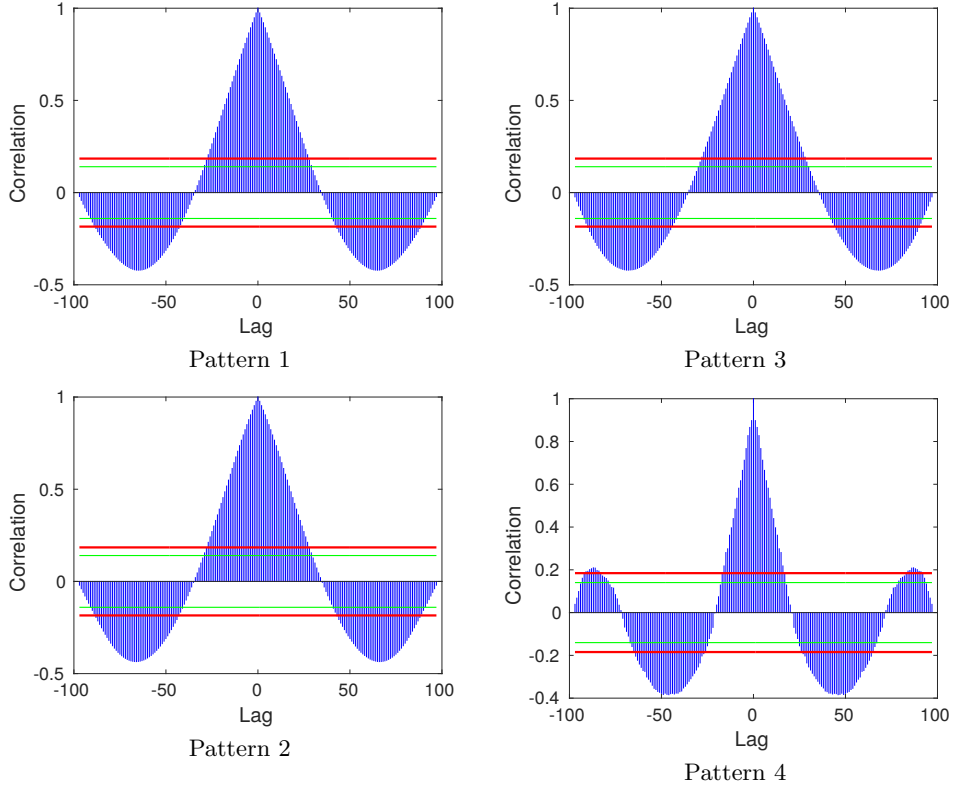
Last, while a dynamic PAT agnostic algorithm was presented for atomic input data, no such algorithm was presented for non-atomic input data. It would be interesting to derive such an algorithm and compare its performance to the clairvoyant algorithm.

Appendix A

## Supplementary figures

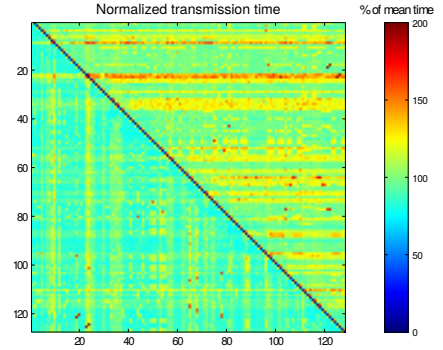
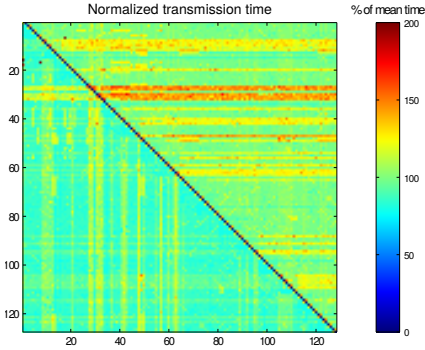
---

## A.1 Helsim trace file autocorrelograms



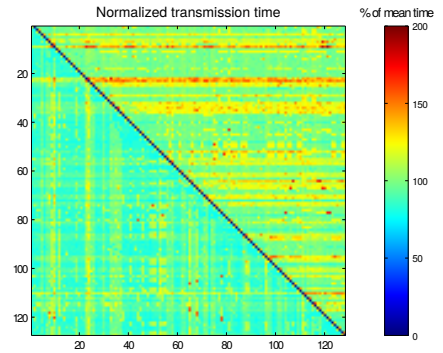
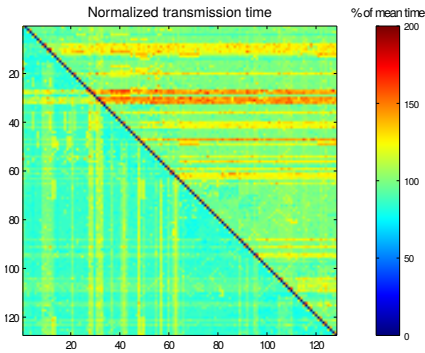
Autocorrelograms of the 4 principal clusters of image render times across the 100 iterations of the Helsim simulation, for  $p < 24$ . The data was gathered on the Lynx cluster with  $P = 128$  and 8 processes per node. In the autocorrelation plots, the green line indicates a  $p = 0.95$  significance level, and the red line indicates a  $p = 0.99$  significance level.

## A.2 Peer-to-peer transfer time experiment



Allocation 2 (iteration 0)

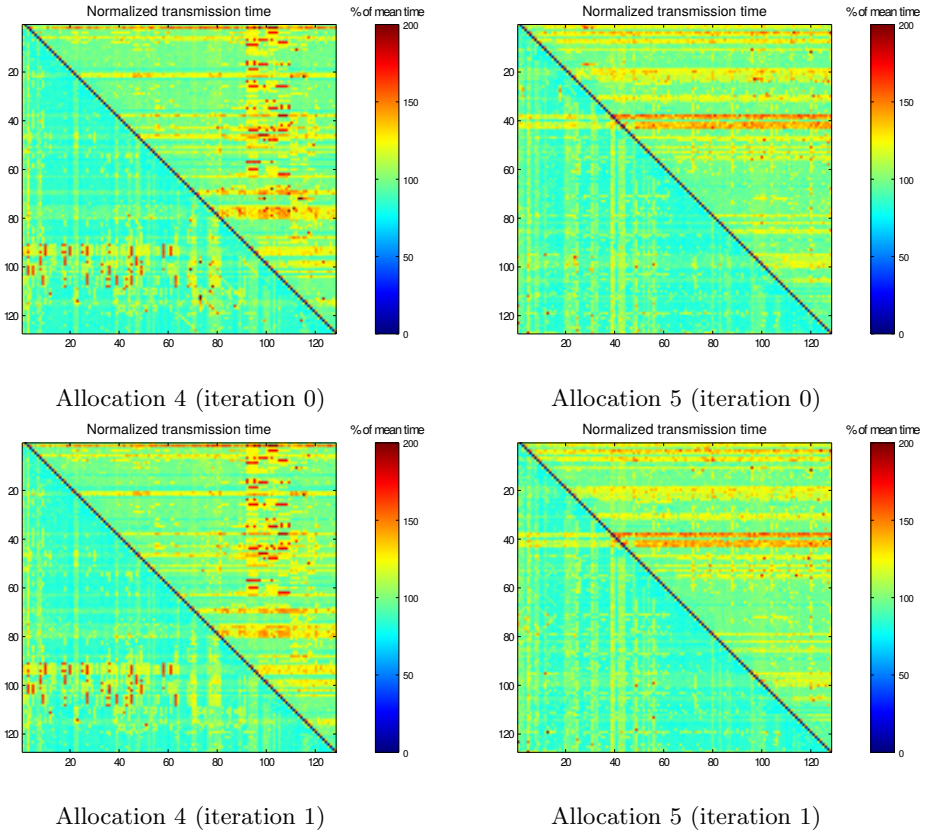
Allocation 3 (iteration 0)



Allocation 2 (iteration 1)

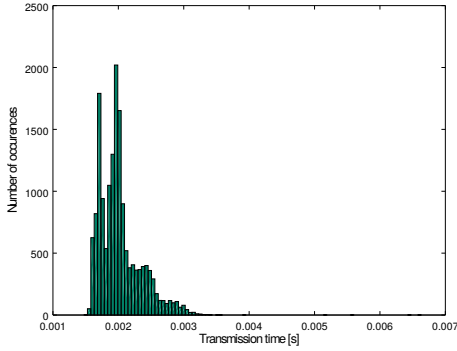
Allocation 3 (iteration 1)

p2p message transmission time across two consecutive iterations of Algorithm 6 (PRACE CURIE; 128 nodes,  $m = 4$  MB)

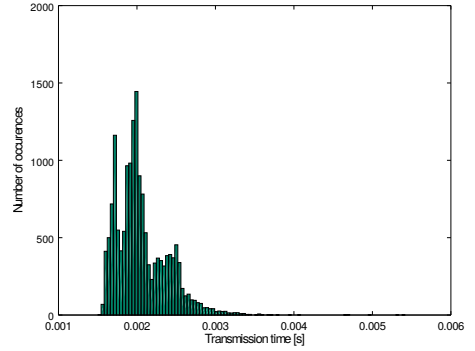


p2p message transmission time across two consecutive iterations of Algorithm 6 (PRACE CURIE; 128 nodes,  $m = 4$  MB)

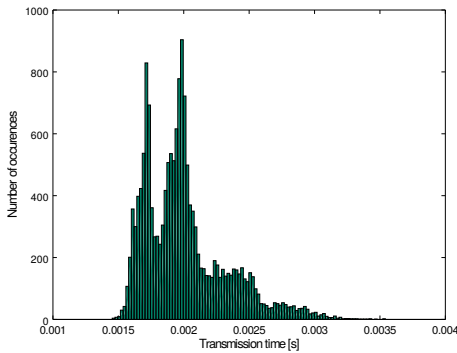
### A.3 Distribution of p2p message transfer time



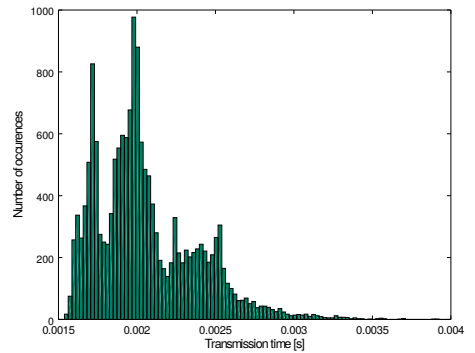
Allocation 2 (iteration 0)



Allocation 3 (iteration 0)

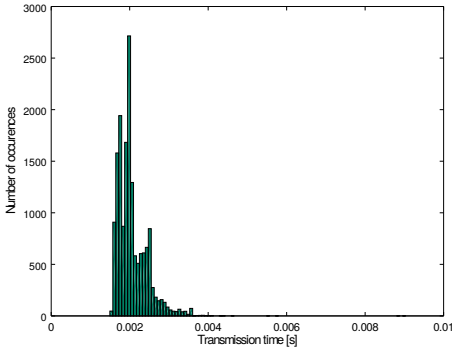


Allocation 2 (iteration 1)

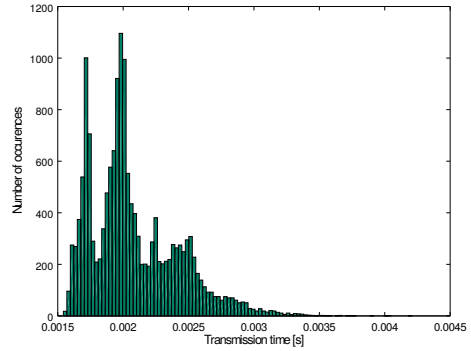


Allocation 3 (iteration 1)

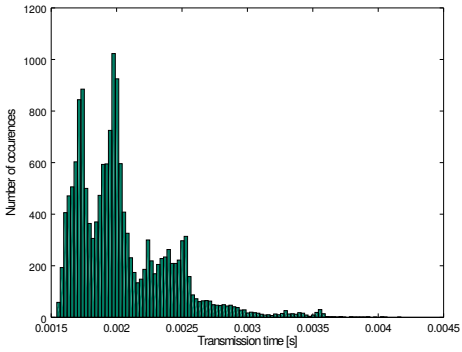
Histograms of p2p message transmission time across two consecutive iterations (PRACE CURIE; 128 nodes,  $m = 4$  MB)



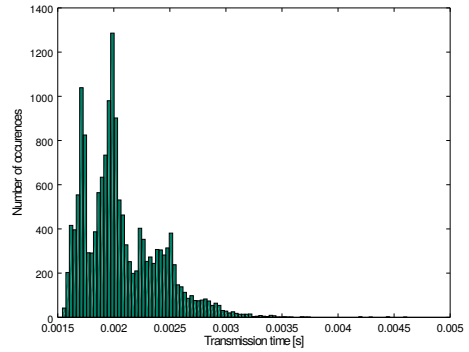
Allocation 4 (iteration 0)



Allocation 5 (iteration 0)



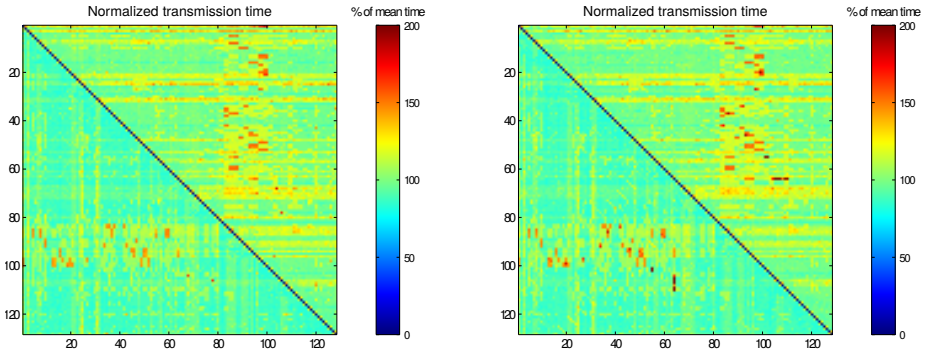
Allocation 4 (iteration 1)



Allocation 5 (iteration 1)

Histograms of p2p message transmission time across two consecutive iterations (PRACE CURIE; 128 nodes,  $m = 4$  MB)

## A.4 Variance in time to complete one round of reduction

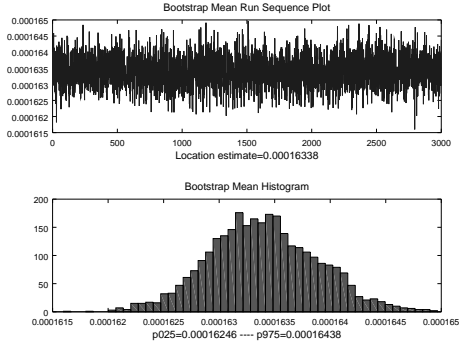


Time to receive and combine (iteration 0)

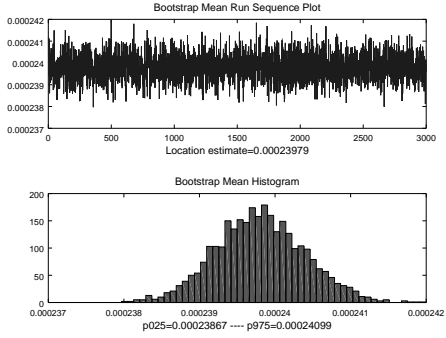
Time to receive and combine (iteration 1)

Execution time of one round of tree reduction algorithms: time to receive a msg of size  $m$ , plus the time to combine a msg of size  $m$  (PRACE CURIE; 128 nodes,  $m = 4$  MB)

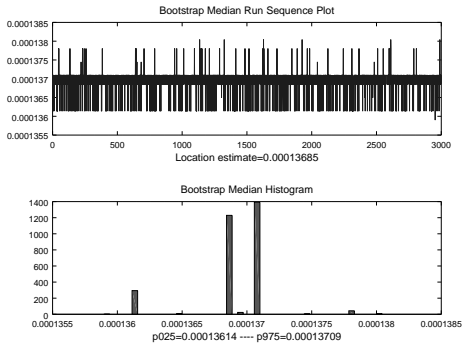
## A.5 Bootstrap plots



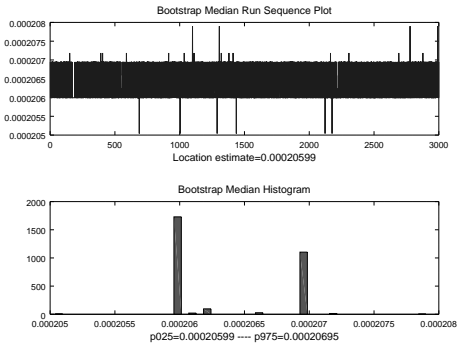
bootstrap of mean,  $m = 200\text{kB}$



bootstrap of mean,  $m = 400\text{kB}$

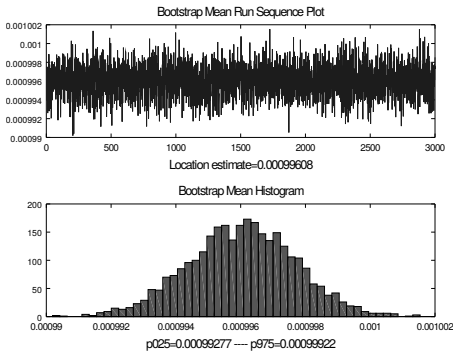


bootstrap of median,  $m = 200\text{kB}$

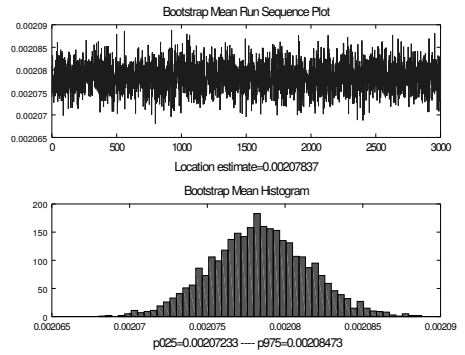


bootstrap of median,  $m = 400\text{kB}$

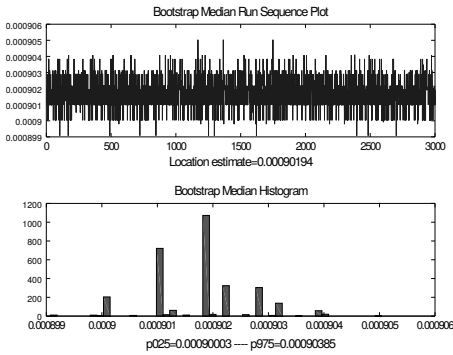
Bootstrap plots for the mean and median of observed message transfer times for  $m \in \{200\text{kB}, 400\text{kB}\}$ . A sample of  $128 \times 128$  p2p message transfer observations was resampled for  $B = 3000$  times to produce the plots. All measurements were performed on the PRACE CURIE supercomputer;  $P=128$ , one process per node



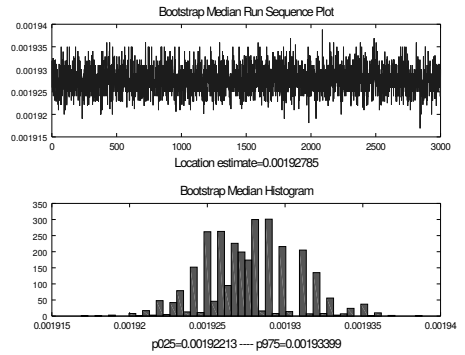
bootstrap of mean,  $m = 2\text{MB}$



bootstrap of mean,  $m = 4\text{MB}$



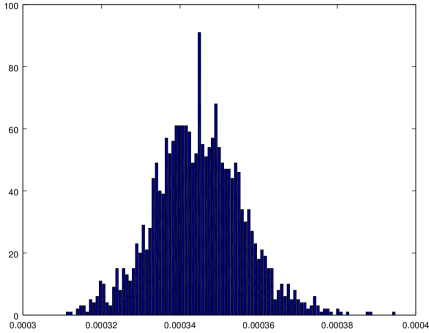
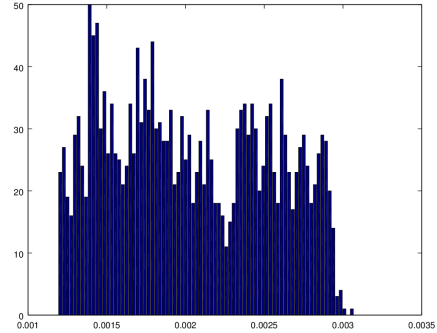
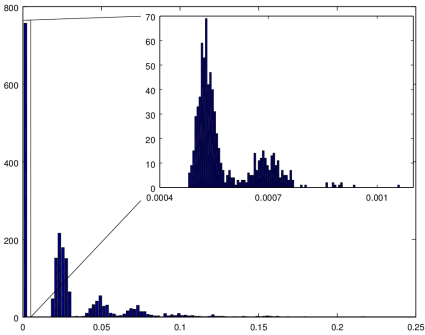
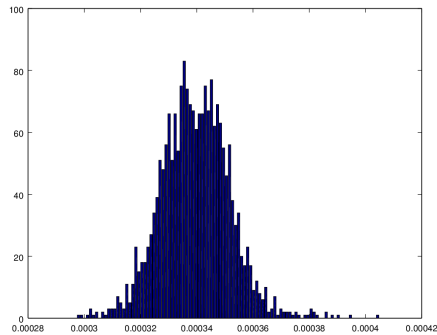
bootstrap of median,  $m = 2\text{MB}$



bootstrap of median,  $m = 4\text{MB}$

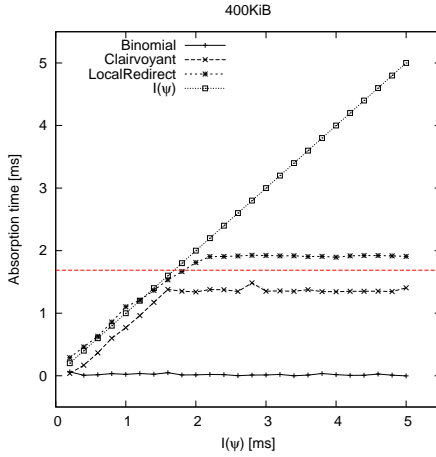
Bootstrap plots for the mean and median of observed message transfer times for  $m \in \{2\text{MB}, 4\text{MB}\}$ . A sample of  $128 \times 128$  p2p message transfer observations was resampled for  $B = 3000$  times to produce the plots. All measurements were performed on the PRACE CURIE supercomputer;  $P=128$ , one process per node

## A.6 Histograms of algorithm runtime for atomic input data

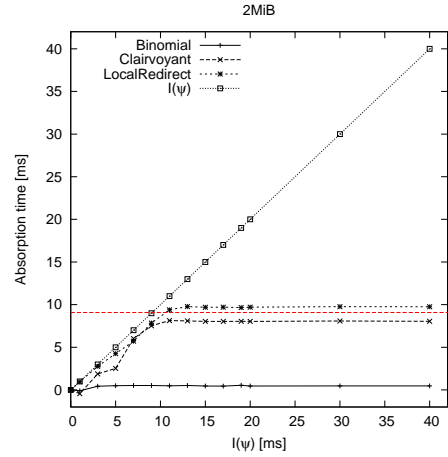
Clairvoyant  $p_{50} = 3.45 \times 10^{-4} s$ Redirect (optimized)  $p_{50} = 2.48 \times 10^{-3} s$ Local Redirect  $p_{50} = 2.29 \times 10^{-2} s$ Binomial  $p_{50} = 3.39 \times 10^{-4} s$ 

Histograms of algorithm runtimes for balanced PATs and  $m = 40$  KiB. Observed runtime in seconds is denoted on the x-axis, while the y-axis denotes the number of occurrences per bin. Sample size  $n = 2048$ . The distributions are not shown in scale. In the captions,  $p_{50}$  stands for median.

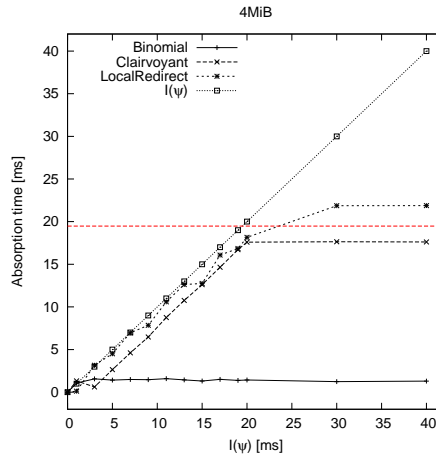
## A.7 Absorption time for the PAT pattern with a single delayed process



(a)  $m = 400KiB$



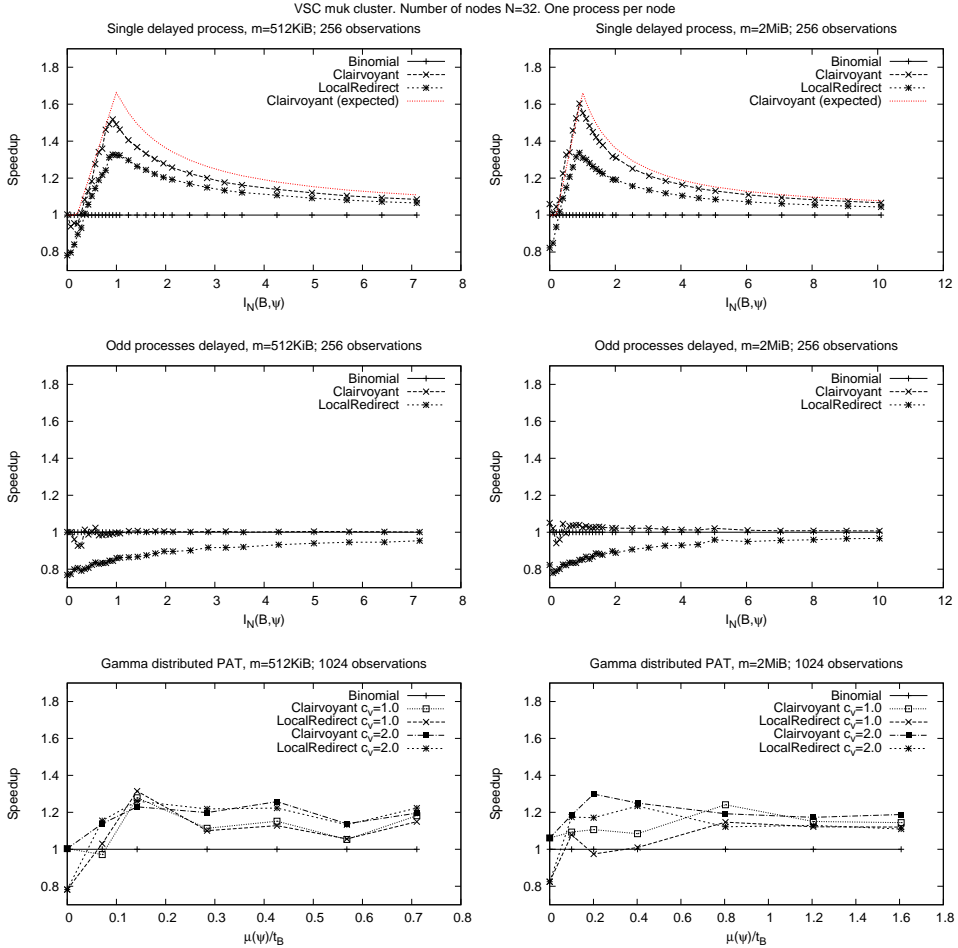
(b)  $m = 2MiB$



(c)  $m = 4MiB$

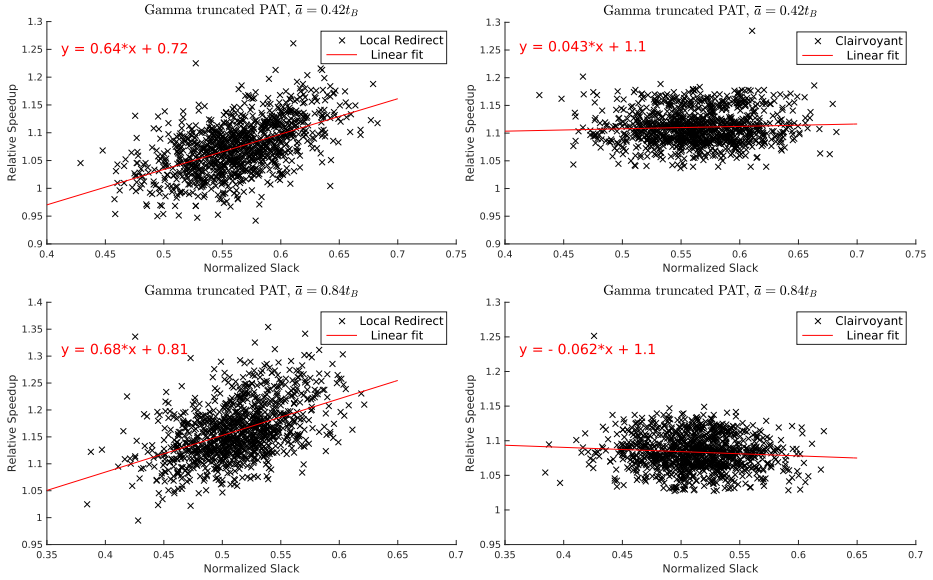
Absorption time (CURIE,  $N=128$ ). Red arrow marks the maximum expected absorption time for algorithm  $\mathcal{C}$  (Clairvoyant), assuming  $t_{\mathcal{C}}(\pi) = t_{\mathcal{B}}(\pi)$  and  $\mathcal{B}=\text{BinomialTree}$ .

## A.8 Observed runtime on the VSC muk cluster



Observed performance on the VSC muk cluster for three different PAT patterns: single delayed process, every odd process delayed and gamma distributed PAT pattern. All runtimes are normalized to algorithm Binomial Tree. The red line (first row) denotes the modelled runtime of algorithm Clairvoyant. The modelled runtime was based on the speedup predicted by Theorem 5.2.2 and the assumption that  $t_B(\pi) = t_C(\pi) \wedge t_B(\psi) = t_B + I(\psi)$ .

## A.9 Truncated gamma distributed PAT experiment



Truncated gamma distributed PATs. In this experiment, two series of 1024 PAT patterns were produced, one with  $\bar{a} = 2$  ms and the other with  $\bar{a} = 4$  ms. For each PAT pattern, 90 runtime observations were recorded and the median runtime was computed to produce the relative speedup, compared to algorithm Binomial Tree. The normalized slack of each PAT pattern was computed according to Definition 3.3.4.



# List of publications

---

## ISI journal publications

1. Marendic, Petar, Jan Lemeire, Dean Vucinic and Peter Schelkens. “On the optimality of reduction algorithms for atomic messages with imbalanced process arrival times”, submitted to the Journal of Parallel and Distributed Computing and under review. Impact factor: 1.179
2. Marendic, Petar, Jan Lemeire, Dean Vucinic and Peter Schelkens. “A novel MPI reduction algorithm resilient to imbalances in process arrival times”, accepted in the Journal of Supercomputing. Impact factor: 0.858

## Conference publications with peer review

1. Marendic, Petar, Tom Haber and Jan Lemeire. “An investigation into the performance of reduction algorithms under load imbalance.” Euro-Par 2012 Parallel Processing. Springer Berlin Heidelberg, 2012. 439-450

## Posters

1. Tom Haber, Petar Marendic, Dean Vucinic, Jan Lemeire, and Philippe Bekaert. 2011. Poster: exascale in-situ visualization using raytracing. In Proceedings of the 2011 companion on High Performance Computing Networking, Storage and Analysis Companion (SC '11 Companion). ACM, New York, NY, USA, 15-16. DOI=10.1145/2148600.2148609 <http://doi.acm.org/10.1145/2148600.2148609>



# References

---

- [1] Sage, version 6.8. available at: <http://www.sagemath.org/> (Aug2015).
- [2] SKaMPI: A comprehensive benchmark for public benchmarking of MPI, author=Reussner, Ralf and Sanders, Peter and Träff, Jesper Larsson, journal=Scientific Programming, volume=10, number=1, pages=55–65, year=2002, publisher=Hindawi Publishing Corporation.
- [3] J. Dongarra W. Jiang R. Manček V. Sunderam A. Geist, A. Beguelin. *PVM: A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.
- [4] CA Addison, VS Getov, AJG Hey, Roger W Hockney, and IC Wolton. The genesis distributed-memory benchmarks. *Computer Benchmarks*, 8:257–271, 1993.
- [5] Saurabh Agarwal, Rahul Garg, and Nisheeth K. Vishnoi. The impact of noise on the scaling of collectives: a theoretical approach. In *Proceedings of the 12th international conference on High Performance Computing*, HiPC'05, pages 280–289, Berlin, Heidelberg, 2005. Springer-Verlag.
- [6] Xin Yuan Ahmad Faraj, Pitch Patarasuk. A Study of Process Arrival Patterns for MPI Collective Operations.
- [7] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauer, and Chris Scheiman. LogGP: incorporating long messages into the LogP model - one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.
- [8] George Almási, Charles Archer, José G Castanos, John A Gunnels, C Christopher Erway, Philip Heidelberger, Xavier Martorell, José E Moreira, Kurt Pinnow, Joe Ratterman, et al. Design and implementation of message-passing services for the blue gene/l supercomputer. *IBM Journal of Research and Development*, 49(2.3):393–406, 2005.
- [9] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 483–485. ACM, 1967.
- [10] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S

- Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.
- [11] Amotz Bar-Noy and Shlomo Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 13–22. ACM, 1992.
- [12] Michael Barnett, R Littlefield, David G Payne, and R Van de Geijn. Global combine on mesh architectures with wormhole routing. In *Parallel Processing Symposium, 1993., Proceedings of Seventh International*, pages 156–162. IEEE, 1993.
- [13] M. S. Bartlett. Properties of sufficiency and statistical tests. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 160(901):268–282, 1937.
- [14] Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, and Paul G. Spirakis. BSP vs LogP. In *SPAA*, pages 25–32, 1996.
- [15] James F Blinn. Compositing. 1. theory. *Computer Graphics and Applications, IEEE*, 14(5):83–87, 1994.
- [16] Shahid H Bokhari and Harry Berryman. Complete exchange on a circuit switched mesh. In *Scalable High Performance Computing Conference, 1992. SHPCC-92, Proceedings*, pages 300–306. IEEE, 1992.
- [17] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, et al. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 29–29. IEEE, 2002.
- [18] Ron Brightwell and Keith D Underwood. An analysis of NIC resource usage for off-loading MPI. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 183. IEEE, 2004.
- [19] Dujdow Buranapanichkit, Nikos Deligiannis, and Yiannis Andreopoulos. Convergence of desynchronization primitives in wireless sensor networks: A stochastic modeling approach. *CoRR*, abs/1411.2862, 2014.
- [20] Robin Calkin, Rolf Hempel, H-C Hoppe, and Peter Wypior. Portable programming with the parmacs message-passing library. *Parallel Computing*, 20(4):615–632, 1994.
- [21] John M Chambers, William S Cleveland, Beat Kleiner, and Paul A Tukey. Graphical methods for data analysis. *Wadsworth, Belmont, CA*, 35, 1983.
- [22] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.
- [23] Ernie Chan, Robert van de Geijn, William Gropp, and Rajeev Thakur. Collective communication on architectures that support simultaneous communication over multiple links. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 2–11. ACM, 2006.

- 
- [24] Ernie W Chan, Marcel F Heimlich, Avi Purkayastha, and Robert A Van De Geijn. On optimizing collective communication. In *Cluster Computing, 2004 IEEE International Conference on*, pages 145–155. IEEE, 2004.
- [25] WenGuang Chen, JiDong Zhai, Jin Zhang, and WeiMin Zheng. LogGPO: An accurate communication model for performance prediction of MPI programs. *Science in China Series F: Information Sciences*, 52(10):1785–1791, 2009.
- [26] Yi-Jen Chiang and Cláudio T Silva. I/O optimal isosurface extraction. In *Visualization'97., Proceedings*, pages 293–300. IEEE, 1997.
- [27] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, March 1953.
- [28] Intel corporation. Intel MPI benchmarks, 2013.
- [29] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauer, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, July 1993.
- [30] Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. High Performance Linpack Benchmark: A Fault Tolerant Implementation Without Checkpointing. In *Proceedings of the International Conference on Supercomputing, ICS '11*, pages 162–171, New York, NY, USA, 2011. ACM.
- [31] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [32] Nikos Deligiannis, João FC Mota, George Smart, and Yiannis Andreopoulos. Decentralized multichannel medium access control: Viewing desynchronization as a convex optimization method. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, pages 13–24. ACM, 2015.
- [33] Nikos Deligiannis, Joao FC Mota, George Smart, and Yiannis Andreopoulos. Fast desynchronization for decentralized multichannel medium access control. *Communications, IEEE Transactions on*, 63(9):3336–3349, 2015.
- [34] Thomas H Dunigan. Hypercube clock synchronization. *Concurrency: Practice and Experience*, 4(3):257–268, 1992.
- [35] Alexandre E. Eichenberger and Santosh G. Abraham. Impact of load imbalance on the design of software barriers. In *In Proceedings of the 1995 International Conference on Parallel Processing*, pages 63–72, 1995.
- [36] Stefan Eilemann and Renato Pajarola. Direct send compositing for parallel sort-last rendering. In *Proceedings of the 7th Eurographics conference on Parallel Graphics and Visualization*, pages 29–36. Eurographics Association, 2007.
- [37] Exascience. Space weather modeling and simulation, October 2015. available at: [http://www.exascience.com/researchold/space-weather/\(Aug2015\)](http://www.exascience.com/researchold/space-weather/(Aug2015)).
- [38] Scott Pakin Fabrizio Petrini, Darren J. Kerbyson. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing, SC '03*, pages 55–, 2003.

- [39] A.A. Faraj. Determining communications latency for transmissions between nodes in a data communications network, 2009. US Patent App. 11/838,969.
- [40] Ahmad Faraj. *Automatic empirical techniques for developing efficient mpi collective communication routines*. Florida State University, 2006.
- [41] Ahmad Faraj, Sameer Kumar, Brian Smith, Amith Mamidala, John Gunnels, and Philip Heidelberger. MPI collective communications on the blue gene/p supercomputer: algorithms and optimizations. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 489–490, New York, NY, USA, 2009. ACM.
- [42] Ahmad Faraj and Xin Yuan. Automatic generation and tuning of MPI collective communication routines. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 393–402. ACM, 2005.
- [43] Ahmad Faraj, Xin Yuan, and David Lowenthal. STAR-MPI: self tuned adaptive routines for MPI collective operations. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 199–208. ACM, 2006.
- [44] D.A. Faraj and B.E. Smith. Runtime optimization of an application executing on a parallel computer, November 25 2014. US Patent 8,898,678.
- [45] Dror G. Feitelson. *Workload Modeling for Computer Systems Performance Evaluation*. draft version, 2013.
- [46] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 19:1–19:12, Piscataway, NJ, USA, 2008. IEEE Press.
- [47] Kurt B Ferreira, Patrick G Bridges, Ron Brightwell, and Kevin T Pedretti. The impact of system design parameters on application noise sensitivity. *Cluster computing*, 16(1):117–129, 2013.
- [48] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept 1972.
- [49] Partnership for Advanced Computing in Europe (PRACE). Prace-11p deliverable 7.4.1: Applications and user requirements for tier-0 systems, May 2012. available at: [http://www.prace-ri.eu/IMG/pdf/d7.4.3\\_1ip.pdf](http://www.prace-ri.eu/IMG/pdf/d7.4.3_1ip.pdf) (Aug2015).
- [50] Working Group for the National Space Weather Program (WG/NSWP) of the Committee for Space Environment Forecasting (CSEF) of the Office of the Federal Coordinator for Meteorological Services and Supporting Research (OFCM). National space weather program strategic plan, August 1995. available at: <http://www.ofcm.gov/nswp-sp/text/a-cover.htm> (Aug2015).
- [51] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.
- [52] The MPI Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

- 
- [53] Michael L Fredman, Robert Sedgewick, Daniel D Sleator, and Robert E Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1-4):111–129, 1986.
- [54] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.
- [55] Tim Kindberg George Coulouris, Jean Dollimore. *Distributed Systems – Concepts and Design*. Addison Wesley, 1994.
- [56] A.V. Gerbessiotis and L.G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251 – 267, 1994.
- [57] Pieter Ghysels, Thomas J. Ashby, Karl Meerbergen, and Wim Vanroose. Hiding Global Communication Latency in the GMRES Algorithm on Massively Parallel Machines. *SIAM J. Scientific Computing*, 35(1), 2013.
- [58] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):47–56, 2004.
- [59] William Gropp and Ewing Lusk. Reproducible measurements of MPI performance characteristics. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 11–18. Springer, 1999.
- [60] Duncan Grove and Paul Coddington. Precise MPI performance measurement using mpibench. In *Proceedings of HPC Asia*, pages 24–28. Citeseer, 2001.
- [61] Duncan A Grove and Paul D Coddington. Communication benchmarking and performance modelling of MPI programs on cluster computers. *The Journal of Supercomputing*, 34(2):201–217, 2005.
- [62] Christopher B Harrison and Klaus Schulten. Quantum and classical dynamics simulations of ATP hydrolysis in solution. *Journal of chemical theory and computation*, 8(7):2328–2335, 2012.
- [63] Alan Heirich and James Arvo. A competitive analysis of load balancing strategies for parallel ray tracing. *The Journal of Supercomputing*, 12(1-2):57–68, 1998.
- [64] Jim Held. Single-chip cloud computer: an IA Tera-Scale Research Processor. In *Euro-Par-Workshop*, page 85, 2010.
- [65] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.
- [66] Tim Hesterberg, David S Moore, Shaun Monaghan, Ashley Clipson, and Rachel Epstein. Bootstrap methods and permutation tests. *Introduction to the Practice of Statistics*, 5:1–70, 2005.
- [67] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

- [68] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.
- [69] Roger W Hockney. Performance parameters and benchmarking of supercomputers. *Parallel computing*, 17(10):1111–1130, 1991.
- [70] Roger W. Hockney. The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Comput.*, 20(3):389–398, March 1994.
- [71] T. Hoefler and A. Lumsdaine. Design, Implementation, and Usage of LibNBC. Technical report, Open Systems Lab, Indiana University, Aug. 2006.
- [72] T. Hoefler and A. Lumsdaine. Message Progression in Parallel Computing - To Thread or not to Thread? In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008.
- [73] T. Hoefler and A. Lumsdaine. Optimizing non-blocking Collective Operations for InfiniBand. In *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium, CAC'08 Workshop*, Apr. 2008.
- [74] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007.
- [75] T. Hoefler and D. Moor. Energy, Memory, and Runtime Tradeoffs for Implementing Collective Communication Operations. *Journal of Supercomputing Frontiers and Innovations*, 1(2):58–75, Oct. 2014.
- [76] T. Hoefler, T. Schneider, and A. Lumsdaine. Multistage switches are not crossbars: Effects of static routing in high-performance networks. In *Cluster Computing, 2008 IEEE International Conference on*, pages 116–125, 2008.
- [77] Torsten Hoefler and Roberto Belli. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 73. ACM, 2015.
- [78] Torsten Hoefler, Peter Gottschling, Wolfgang Rehm, and Andrew Lumsdaine. Optimizing a conjugate gradient solver with non-blocking collective operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 374–382. Springer, 2006.
- [79] Torsten Hoefler, William Gropp, Rajeev Thakur, and JesperLarsson Trff. Toward performance models of MPI implementations for understanding application scaling issues. In Rainer Keller, Edgar Gabriel, Michael Resch, and Jack Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6305 of *Lecture Notes in Computer Science*, pages 21–30. Springer Berlin Heidelberg, 2010.
- [80] Torsten Hoefler, Prabhanjan Kambadur, Richard L Graham, Galen Shipman, and Andrew Lumsdaine. A case for standard non-blocking collective operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 125–134. Springer, 2007.

- 
- [81] Torsten Hoefler, Torsten Mehlan, Andrew Lumsdaine, and Wolfgang Rehm. Net-gauge: A network performance measurement framework. In *HPCC*, volume 7, pages 659–671. Springer, 2007.
- [82] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Multistage switches are not crossbars: Effects of static routing in high-performance networks. In *Cluster Computing, 2008 IEEE International Conference on Cluster Computing*, pages 116–125. IEEE, 2008.
- [83] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Accurately measuring overhead, communication time and progression of blocking and nonblocking collective operations at massive scale. *Int. J. Parallel Emerg. Distrib. Syst.*, 25(4):241–258, August 2010.
- [84] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10*, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.
- [85] Chao Huang, Orion Lawlor, and Laxmikant V Kale. Adaptive MPI. In *Languages and Compilers for Parallel Computing*, pages 306–322. Springer, 2004.
- [86] Chao Huang, Gengbin Zheng, Laxmikant Kalé, and Sameer Kumar. Performance evaluation of adaptive MPI. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 12–21. ACM, 2006.
- [87] Wei Huang, Gopalakrishnan Santhanaraman, Hyun-Wook Jin, Qi Gao, and Dhaleswar K Panda. Design of high performance MVAPICH2: MPI2 over InfiniBand. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 43–48. IEEE, 2006.
- [88] Sascha Hunold, Alexandra Carpen-Amarie, and Jesper Larsson Träff. Reproducible MPI micro-benchmarking isn't as easy as you think. In *Proceedings of the 21st European MPI Users' Group Meeting*, page 69. ACM, 2014.
- [89] Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara. LogGPS: a parallel computational model for synchronization analysis. *SIGPLAN Not.*, 36(7):133–142, June 2001.
- [90] Intel. Intel 64 and IA-32 Architectures Software Developers Manual Volume 2B: Instruction Set Reference: N-Z. *Part, 2*, 2015.
- [91] MPI Intel. Benchmarks: Users guide and methodology description. *Intel GmbH, Germany*, 452, 2004.
- [92] Nikhil Jain and Yogish Sabharwal. Optimal bucket algorithms for large MPI collectives on torus interconnects. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 27–36. ACM, 2010.
- [93] Rick Jones et al. NetPerf: a network performance benchmark. *Information Networks Division, Hewlett-Packard Company*, 1996.

- [94] Wall Street Journal. Intel rechisels the tablet on Moores law, July 2015. available at: <http://blogs.wsj.com/digits/2015/07/16/intel-rechisels-the-tablet-on-moores-law/> (Aug2015).
- [95] Wall Street Journal. U.s. agencies block technology exports for supercomputer in china, April 2015. available at: <http://www.wsj.com/articles/u-s-agencies-block-technology-exports-for-supercomputer-in-china-1428561987> (Aug2015).
- [96] Laxmikant V Kale. The chare kernel parallel programming language and system. In *ICPP (2)*, pages 17–25, 1990.
- [97] Richard M Karp, Abhijit Sahay, Eunice E Santos, and Klaus Erik Schauer. Optimal broadcast and summation in the LogP model. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 142–153. ACM, 1993.
- [98] Amit Karwande, Xin Yuan, and David K Lowenthal. An MPI prototype for compiled communication on ethernet switched clusters. *Journal of Parallel and Distributed Computing*, 65(10):1123–1133, 2005.
- [99] Wesley Kendall, Tom Peterka, Jian Huang, Han-Wei Shen, and Robert Ross. Accelerating and benchmarking radix-k image compositing at large scale. In *Proceedings of the 10th Eurographics conference on Parallel Graphics and Visualization*, EG PGV'10, pages 101–110, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.
- [100] Ajay D Kshemkalyani and Mukesh Singhal. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2008.
- [101] Sameer Kumar, Amith Mamidala, Philip Heidelberger, Dong Chen, and Daniel Faraj. Optimization of MPI collective operations on the ibm blue gene/q supercomputer. *Int. J. High Perform. Comput. Appl.*, 28(4):450–464, November 2014.
- [102] Sandia National Laboratories. CTH, Version 10.3, February 2013. available at: <http://www.sandia.gov/CTH/> (Aug2015).
- [103] Leslie Lamport. Interprocess communication. Technical report, DTIC Document, 1985.
- [104] Giovanni Lapenta, Stefano Markidis, Stefaan Poedts, and Dean Vucinic. Space weather prediction and exascale computing. *Computing in Science & Engineering*, 15(5):68–76, 2013.
- [105] Alexey Lastovetsky, Vladimir Rychkov, and Maureen OFlynn. Mpiplib: Benchmarking MPI communications for parallel computing on homogeneous and heterogeneous clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 227–238. Springer, 2008.
- [106] Charles E Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 100(10):892–901, 1985.
- [107] Howard Levene. Robust tests for equality of variances1. *Contributions to probability and statistics: Essays in honor of Harold Hotelling*, 2:278–292, 1960.

- 
- [108] Xiang-Ke Liao, Zheng-Bin Pang, Ke-Fei Wang, Yu-Tong Lu, Min Xie, Jun Xia, De-Zun Dong, and Guang Suo. High performance interconnect network for Tianhe system. *Journal of Computer Science and Technology*, 30(2):259–272, 2015.
- [109] Zhi-Qiang Liu, Jun-Qiang Song, Feng-Shun Lu, and Fen Xu. Optimizing method for improving the performance of MPI broadcast under unbalanced process arrival patterns. *Journal of Software*, 22(10), 2011.
- [110] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *ACM siggraph computer graphics*, volume 21, pages 163–169. ACM, 1987.
- [111] Gabriel Antoniu Louis-Claude Canon. Scheduling associative reductions with homogenous costs when overlapping communications and computations. Technical Report 7898, Inria, 2012.
- [112] Kwan-Liu Ma, James S. Painter, Charles D. Hansen, and Michael F Krogh. Parallel volume rendering using binary-swap compositing. *Computer Graphics and Applications, IEEE*, 14(4):59–68, 1994.
- [113] Amithand Mamidala, Jiuxing Liu, and Dhabaleswar K. Panda. Efficient barrier and allreduce on infiniband clusters using multicast and adaptive algorithms. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing, CLUSTER '04*, pages 135–144, Washington, DC, USA, 2004. IEEE Computer Society.
- [114] Petar Marendic, Jan Lemeire, Tom Haber, Dean Vucinic, and Peter Schelkens. An investigation into the performance of reduction algorithms under load imbalance. In Christos Kaklamanis, Theodore Papatheodorou, and PaulG. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 439–450. Springer Berlin Heidelberg, 2012.
- [115] Stefano Markidis, Giovanni Lapenta, et al. Multi-scale simulations of plasma with iPIC3D. *Mathematics and Computers in Simulation*, 80(7):1509–1519, 2010.
- [116] D.R. Martinez, J.C. Cabaleiro, T.F. Pena, F.F Rivera, and V. Blanco. Accurate analytical performance model of communications in MPI applications. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8, 2009.
- [117] Vikas Mehta. LeanMD: A Charm++ framework for high performance molecular dynamics simulation on large parallel machines. *Master's thesis, Univeristy of Illinois at Urbana-Champaign*, 2004.
- [118] Esteban Meneses and Laxmikant V. Kal. Camel: collective-aware message logging. *The Journal of Supercomputing*, 71(7):2516–2538, 2015.
- [119] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 1.1, June 1995. available at: <http://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/mpi-report.html> (Feb 2016).
- [120] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 1.2, July 1997. available at: <http://www.mpi-forum.org/docs/mpi-2.0/mpi-20-html/mpi2-report.html> (Feb 2016).

- [121] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 2.0, July 1997. available at: <http://www.mpi-forum.org/docs/mpi-2.0/mpi-20-html/mpi2-report.html> (Feb 2016).
- [122] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 1.3, May 2008. available at: <http://www.mpi-forum.org/docs/mpi-1.3/mpi-report-1.3-2008-05-30.pdf> (Feb 2016).
- [123] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 2.1, June 2008. available at: <http://www.mpi-forum.org/docs/mpi-2.1/mpi-report-2.1-2008-06-23.pdf> (Feb 2016).
- [124] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 2.2, September 2009. available at: <http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf> (Feb 2016).
- [125] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 3.0, September 21st 2012. available at: <http://www.mpi-forum.org/docs/mpi-3.0/> (Mar 2015).
- [126] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 3.1, June 4th 2015. available at: <http://www.mpi-forum.org/docs/mpi-3.1/> (Feb 2016).
- [127] Jace A Mogill and David J Haglin. A comparison of shared memory parallel programming models. *Pacific Northwest National Laboratory, Richland, WA, USA*, 2010.
- [128] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *Computer Graphics and Applications, IEEE*, 14(4):23–32, July 1994.
- [129] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *Computer Graphics and Applications, IEEE*, 14(4):23–32, 1994.
- [130] Adam Moody, Juan Fernandez, Fabrizio Petrini, and Dhabaleswar K Panda. Scalable NIC-based reduction on large-scale clusters. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 59–59. IEEE, 2003.
- [131] Kenneth Moreland, Wesley Kendall, Tom Peterka, and Jian Huang. An image compositing solution at scale. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 25. ACM, 2011.
- [132] Csaba Andras Moritz and Matthew I. Frank. LoGPC: Modeling Network Contention in Message-Passing Programs. *IEEE Trans. Parallel Distrib. Syst.*, 12(4):404–415, April 2001.
- [133] Frederick Mosteller and John Wilder Tukey. Data analysis and regression: a second course in statistics. *Addison-Wesley Series in Behavioral Science: Quantitative Methods*, 1977.
- [134] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

- 
- [135] National Science Foundation, US. A report of the national science foundation advisory committee for cyberinfrastructure task force on grand challenges, March 2011. available at: [https://www.nsf.gov/cise/aci/taskforces/TaskForceReport\\_GrandChallenges.pdf](https://www.nsf.gov/cise/aci/taskforces/TaskForceReport_GrandChallenges.pdf) (Aug 2015).
- [136] Ulrich Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *Parallel Rendering Symposium, 1993*, pages 97–104. IEEE, 1993.
- [137] Ulrich Neumann. Communication costs for parallel volume-rendering algorithms. *Computer Graphics and Applications, IEEE*, 14(4):49–58, 1994.
- [138] NIST/SEMATECH. e-handbook of statistical methods, 2012. Available at: <http://www.itl.nist.gov/div898/handbook/> (Jun 2015).
- [139] Benjamin S. Parsons and Vijay S. Pai. Exploiting process imbalance to improve MPI collective operations in hierarchical systems. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS '15*, pages 57–66, New York, NY, USA, 2015. ACM.
- [140] Pitch Patarasuk and Xin Yuan. Efficient MPI bcast across different process arrival patterns. *Parallel and Distributed Processing Symposium, International*, 0:1–11, 2008.
- [141] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.
- [142] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.
- [143] Tom Peterka, David Goodell, Robert Ross, Han-Wei Shen, and Rajeev Thakur. A configurable algorithm for parallel image-compositing applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09*, pages 4:1–4:10, New York, NY, USA, 2009. ACM.
- [144] Antoine Petit. HPL-a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. <http://www.netlib.org/benchmark/hpl/>, 2004.
- [145] Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg, and Adolfo Hoisie. Hardware- and software-based collective communication on the Quadrics network. In *Network Computing and Applications, 2001. NCA 2001. IEEE International Symposium on*, pages 24–35. IEEE, 2001.
- [146] Fabrizio Petrini, Juan Fernandez, Eitan Frachtenberg, and Salvador Coll. Scalable collective communication on the ASCI Q machine. In *High Performance Interconnects, 2003. Proceedings. 11th Symposium on*, pages 54–59. IEEE, 2003.
- [147] Seth Pettie. Towards a final analysis of pairing heaps. In *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*, pages 174–183. IEEE, 2005.

- [148] Jelena Pjesivac-Grbovic, Thara Angskun, George Bosilca, Graham E. Fagg, and Gabriel Jack J. Dongarra. Performance analysis of MPI collective operations. In *IEEE International Parallel and Distributed Processing Symposium*, 2005.
- [149] The Platform. The Strange Story of the U.S. Weather Supercomputer Police in China, July 2015. available at: <http://www.theplatform.net/2015/07/01/how-the-u-s-supercomputer-police-patrolled-chinese-weather-systems/> (Aug2015).
- [150] PRACE. Prace resources, 2011. available at: <http://www.prace-ri.eu/prace-resources/> (Aug2015).
- [151] Ying Qian. *Design and Evaluation of Efficient Collective Communications on Modern Interconnects and Multi-core Clusters*. PhD thesis, Queen's University, 2010.
- [152] Rolf Rabenseifner. Optimization of collective reduction operations. In *Procs. of Int. Conf. on Computational Science (ICCS)*, pages 1–9, 2004.
- [153] Rolf Rabenseifner and Jesper Larsson Trff. More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems. In *EuroPVM/MPI*, pages 36–46, 2004.
- [154] Alexey Andreyevich Radul. *Propagation Networks: A Flexible and Expressive Substrate for Computation*. PhD thesis, Massachusetts Institute of Technology (MIT), 2009.
- [155] Mohammad J. Rashti and Ahmad Afsahi. Assessing the ability of computation/communication overlap and communication progress in modern interconnects. In *Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects, HOTI '07*, pages 117–124, Washington, DC, USA, 2007. IEEE Computer Society.
- [156] Nornadiah Mohd Razali and Yap Bee Wah. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *Journal of Statistical Modeling and Analytics*, 2(1):21–33, 2011.
- [157] Shaolei Ren, N. Deligiannis, Y. Andreopoulos, M.A. Islam, and M. van der Schaar. Dynamic scheduling for energy minimization in delay-sensitive stream mining. *Signal Processing, IEEE Transactions on*, 62(20):5439–5448, Oct 2014.
- [158] Ralf Reussner, Peter Sanders, Lutz Prechelt, and Matthias Mller. Skampi: A detailed, accurate MPI benchmark. In Vassil Alexandrov and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 52–59. Springer Berlin Heidelberg, 1998.
- [159] Jorge L Reyes-Ortiz, Luca Oneto, and Davide Anguita. Big data analytics in the cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf. *Procedia Computer Science*, 53:121–130, 2015.
- [160] Nicolas Richart, Aurélien Esnard, and Olivier Coulaud. Toward a computational steering environment for legacy coupled simulations. In *Parallel and Distributed Computing, 2007. ISPDC'07. Sixth International Symposium on*, pages 43–43. IEEE, 2007.

- 
- [161] Rolf Riesen, Ron Brightwell, and Arthur B Maccabe. Measuring MPI latency variance. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 112–116. Springer, 2003.
- [162] Eduardo R. Rodrigues, Philippe O. A. Navaux, Jairo Panetta, Alvaro Fazenda, Celso L Mendes, and Laxmikant V Kale. A comparative analysis of load balancing algorithms applied to a weather forecast model. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, pages 71–78. IEEE, 2010.
- [163] Diego Rossinelli, Babak Hejazialhosseini, Panagiotis Hadjidoukas, Costas Bekas, Alessandro Curioni, Adam Bertsch, Scott Futral, Steffen J Schmidt, Nikolaus Adams, Petros Koumoutsakos, et al. 11 pflop/s simulations of cloud cavitation collapse. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–13. IEEE, 2013.
- [164] Peter Sanders, Jochen Speck, and Jesper Larsson Trff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581 – 594, 2009. Selected papers from the 14th European PVM/MPI Users Group Meeting.
- [165] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.
- [166] R Schwenn, JC Raymond, D Alexander, A Ciaravella, N Gopalswamy, R Howard, H Hudson, P Kaufmann, A Klassen, D Maia, et al. Coronal observations of cmes. *Space Science Reviews*, 123(1-3):127–176, 2006.
- [167] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, pages 591–611, 1965.
- [168] David E Shaw, JP Grossman, Joseph A Bank, Brannon Batson, J Adam Butts, Jack C Chao, Martin M Deneroff, Ron O Dror, Amos Even, Christopher H Fenton, et al. Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 41–53. IEEE Press, 2014.
- [169] Mohak Shroff and Robert A Van De Geijn. Collmark: MPI collective communication benchmark. In *International Conference on Supercomputing*, volume 2000, page 10. Citeseer, 1999.
- [170] George Smart, Nikos Deligiannis, Rosario Surace, Valeria Loscri, Giancarlo Fortino, and Yiannis Andreopoulos. Decentralized Time-Synchronized Channel Swapping for Ad Hoc Wireless Networks. *IEEE Transactions on Vehicular Technology*, 2016.
- [171] George Snecdecor and William Cochran. *Statistical Methods, 8th Edition*. Wiley, 1989.
- [172] Quinn O Snell, Armin R Mikler, and John L Gustafson. Netpipe: A network protocol independent performance evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, volume 6. Washington, DC, USA), 1996.

- [173] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.
- [174] IEEE Spectrum. Why aren't supercomputers getting faster like they used to?, July 2015. available at: <http://spectrum.ieee.org/tech-talk/computing/hardware/supercomputing-list-reveals-performance-development-is-down> (Aug 2015).
- [175] Bharath Sundararaman, Ugo Buy, and Ajay D Kshemkalyani. Clock synchronization for wireless sensor networks: a survey. *Ad Hoc Networks*, 3(3):281–323, 2005.
- [176] Ivan E Sutherland, Robert F Sproull, and Robert A Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys (CSUR)*, 6(1):1–55, 1974.
- [177] T. Schneider T. Hoefler and A. Lumsdaine. Accurately measuring overhead, communication time and progression of blocking and nonblocking collective operations at massive scale. *International Journal of Parallel, Emergent and Distributed Systems*, 25(4):241–258, 2010.
- [178] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.
- [179] Jesper Larsson Träff. mpimicroscope: Towards an MPI benchmark tool for performance guideline verification. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *Recent Advances in the Message Passing Interface - 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, volume 7490 of *Lecture Notes in Computer Science*, pages 100–109. Springer, 2012.
- [180] Jesper Larsson Träff and Andreas Ripke. Optimal broadcast for fully connected processor-node networks. *Journal of Parallel and Distributed Computing*, 68(7):887–901, 2008.
- [181] Shu-Ju Tu and Ephraim Fischbach. A study on the randomness of the digits of pi. *International Journal of Modern Physics C*, 16(02):281–294, 2005.
- [182] Keith D Underwood and Ron Brightwell. The impact of MPI queue usage on message latency. In *Parallel Processing, 2004. ICPP 2004. International Conference on*, pages 152–160. IEEE, 2004.
- [183] Sathish S Vadhiyar, Graham E Fagg, and Jack Dongarra. Automatically tuned collective communications. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, page 3. IEEE Computer Society, 2000.
- [184] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.
- [185] Manjunath Gorentla Venkata, Pavel Shamis, Rahul Sampath, Richard L Graham, and Joshua S Ladd. Optimizing blocking and nonblocking reduction operations for multicore systems: Hierarchical design and implementation. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.
- [186] Abraham Wald and Jacob Wolfowitz. On a test whether two samples are from the same population. *The Annals of Mathematical Statistics*, 11(2):147–162, 1940.

- 
- [187] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1):3–35, 2001.
- [188] Patrick Widener, Kurt B Ferreira, Scott Levy, and Torsten Hoefer. Exploring the effect of noise on the performance benefit of nonblocking allreduce. In *Proceedings of the 21st European MPI Users' Group Meeting*, page 77. ACM, 2014.
- [189] Martin B Wilk and Ram Gnanadesikan. Probability plotting methods for the analysis for the analysis of data. *Biometrika*, 55(1):1–17, 1968.
- [190] Barry Wilkinson and Michael Allen. *Parallel programming*, volume 999. Prentice hall New Jersey, 1999.
- [191] HPC Wire. Advancing applications toward exascale target, February 2014. available at: <http://www.hpcwire.com/2014/02/11/advancing-applications-toward-exascale-target/> (Aug 2015).
- [192] Thomas Worsch, Ralf Reussner, and Werner Augustin. On benchmarking collective MPI operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 271–279. Springer, 2002.
- [193] Thomas Worsch, Ralf Reussner, and Werner Augustin. Benchmarking collective operations with SKaMPI. In *High Performance Computing in Science and Engineering02*, pages 491–502. Springer, 2003.
- [194] Ming Xue, Kelvin K Droegemeier, and Daniel Weber. Numerical prediction of high-impact local weather: A driver for petascale computing. *Petascale Computing: Algorithms and Applications*, 200:103–124, 2007.
- [195] Don-Lin Yang, Jen-Chih Yu, and Yeh-Ching Chung. Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. *The Journal of Supercomputing*, 18(2):201–220, 2001.
- [196] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32. ACM, 2007.
- [197] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.
- [198] Hongfeng Yu, Chaoli Wang, Ray W Grout, Jacqueline H Chen, and Kwan-Liu Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications*, (3):45–57, 2010.
- [199] Hongfeng Yu, Chaoli Wang, and Kwan-Liu Ma. Massively parallel volume rendering using 2–3 swap image compositing. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.

- [200] Xin Yuan, Scott Daniels, Ahmad Faraj, and Amit Karwande. Group management schemes for implementing MPI collective communication over ip-multicast. In *JCIS*, pages 309–313, 2002.
- [201] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.
- [202] Eitan Zahavi. Fat-tree routing and node ordering providing contention free traffic for MPI global collectives. *Journal of Parallel and Distributed Computing*, 72(11):1423–1432, 2012.

# Index

---

- absolute imbalance, 53
- absorption time, 58
  - limit, 109
- atomic, 36
- automatic tuning, 32
- benchmarking, 71
- bootstrapping
  - bootstrap plots, 94
  - uncertainty estimate, 94
- BSP, 60
- clairvoyancy, 101
- clairvoyant, 13
- clairvoyant schedule generator
  - non-atomic, 126
- collective communication, 29
- CRESTA, 7
- CUDA, 22
- CURIE, 72
- distributed memory system, 16
- effective bisection bandwidth, 19
- exascale, 2
- global clock, 74
- grand challenges, 1
- Hadoop, 19
- hardware clock, 73
- HPC, 1
- image rendering, 34
  - image compositing, 34
  - over-operator, 34
  - sort-last, 34
- imbalance
  - worst-case, 114
- independent progress, 64
- Infiniband, 18
- job scheduler, 18
- linear cost model, 63
- load balancing, 51
- load imbalance, 49
  - microbenchmark, 57
- LogP, 61
- Lynx, 72
- MapReduce, 19
- mesh topology, 18
- microbenchmark
  - all-pairs p2p, 76, 79
  - Intel MPI, 86
  - MPBench, 85
  - MPIBench, 86
  - MPIBlib, 86
  - MPPTEST, 85
  - NBCBench, 86
  - p2p latency, 75
  - PAT imbalance injection, 87
  - SKaMPI, 86
- MPI, 2

- one-sided communication, 24
- two-sided communication, 24
- MPI\_Reduce, 31
- MPICH, 25
- network interconnect, 18
  - central-switch, 18
  - multi-stage interconnect, 18
- non-blocking collective operation, 32
- Open MPI, 25
- OpenCL, 22
- OpenMP, 22
- parallel programming model, 15
  - data-parallel, 16
  - message-passing, 20
  - shared memory, 21
  - task-parallel, 15
- PAT, 53
  - definition, 53
  - local information, 113
- permutation test, 95
- persistent communication, 21
- PGAS, 22
- point-to-point communication, 25
- PRACE, 22
- Process Arrival Time, 12
- process synchronization, 74
- Pthreads, 22
- reduction, 11
  - binomial tree, 109
  - clairvoyant atomic, 110
  - local redirect, 111
  - operation, 11
  - optimal atomic, 107
- reduction problem
  - definition, 35
- reduction schedule
  - dynamic, 113
- runtime, 54
  - global measure, 54
  - local measure, 54
- Spark, 19
- state-of-the-art, 38
- synchronization delay, 58
- system noise, 52
- Tianhe-2, 17
- Titan, 17
- tree topology, 18
  - fat-tree, 18
- visualization, 9, 33
  - in-situ, 10
  - isosurface rendering, 33