PERFORMANCE TUNING AND MODELING OF COMMUNICATION IN PARALLEL APPLICATIONS

A Dissertation Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Shweta Jha

May 2017

PERFORMANCE TUNING AND MODELING OF COMMUNICATION IN PARALLEL APPLICATIONS

Shweta Jha

APPROVED:

Dr. Edgar Gabriel, Chairman
Dept. of Computer Science, University of Houston

Dr. Jaspal Subhlok
Dept. of Computer Science, University of Houston

Dr. Weidong Shi
Dept. of Computer Science, University of Houston

Dr. Deniz Gurkan
Dept. of Engineering Tech, University of Houston

Dean, College of Natural Sciences and Mathematics

Acknowledgments

First and foremost I would like to thank my dissertation advisor Dr. Edgar Gabriel for his support, and encouragement. His contagious enthusiasm towards research have been a motivation for me throughout my Ph.D. It is a privilege to learn under your guidance. I really enjoyed our discussions, and thank you for being so approachable.

I will take this opportunity to show my gratitude to my committee members, Dr. Jaspal Subhlok, Dr. Larry Shi, and Dr. Deniz Gurkan for agreeing to be in my committee. Your valuable inputs and insightful discussions during my Ph.D proposal defense was of immense help in planning and structuring my work. I would like to thank Total E&P and Peirre-Yves Aquilanti who gave me the opportunity to intern with them and provided me insight on how HPC is used in Oil and Gas.

I would like to thank Anant, for his constant assurance and belief in me. I would also like to show my appreciation to my friends - Amrita, Ananya, Pooja, Prajakta Tejas for making my life comfortable here. All other friends from Houston and back home thank you for all the encouragement and belief in me. The last five years in PSTL were amazing, and it would not have been possible without former and current members of group: Vish, Kshitij, Jyothi, Youcef, Priya, and Sonia.

Last but definitely not the least I would like to thank my family. My parents who always have been there for me. Choti Didi, you are my inspiration to come back to grad school. Badi Didi, Soeham, Naman and both jijus for providing me unconditional love and support throughout my journey here.

PERFORMANCE TUNING AND MODELING OF COMMUNICATION IN PARALLEL APPLICATIONS

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Shweta Jha

May 2017

Abstract

The goal of high performance computing is executing very large problems in the least amount of time, typically by deploying parallelization techniques. However, introducing parallelization to an application also introduces synchronization and communication overhead, which in turn creates a performance bottleneck. Performance modeling and tuning can be used to predict and ease this bottleneck to improve the overall performance of the application.

There are two aspects of an application which can be improved from performance point of view, namely, the computational section and the communication section. The time spent in communication operations is a major factor in determining the scalability of parallel applications. Tuning the parameters of a communication library can be used to adapt its characteristics to a particular platform, minimizing the communication time of an application. On the other hand performance modeling can be used to predict the performance using the network and application attributes.

The goal of this dissertation is to improve the performance of a parallel application by performance tuning and performance modeling. Specifically, we introduce the notion of a personalized MPI library, highlighting the necessity and the methodology each application needs to have a communication library tuned for the particular platform. Secondly, this dissertation contributes towards the theoretical understanding of impact and limitations of point-to-point communication performance on collective communication and the overall application. This study has been further extended to develop performance models for communication aspect of collective I/O for one and two dimensional data decomposition, and for two file partitioning strategies, namely even and static partitioning.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High Performance Computing (HPC) is used to solve complex problems in numerous fields like quantum mechanics, climate research, oil and gas, and molecular modeling. The focus of the research in HPC is to aggregate the maximum computing power and use this to improve the performance of applications. In the last few years, on account of various advancements in technology, large volumes of data have been aggregated. Analyzing and interpreting this humongous volume of data, with complexity much higher than previously available data has been a driving force for the development of HPC. The three aspects of HPC that are important in the context of the present work are: hardware platform, algorithms (parallel computing theoretical basis), and software support. In the next section, we discuss hardware trends and various programming models.

1.1 Recent trends in HPC: hardware

Hardware research is primarily focussed on the development of device technology, memory capacity, communication bandwidth and latency, the system architecture, and the interconnect topology among the cores. This development is especially conspicuous when we compare the current trend with few of the initial supercomputers. The first supercomputer, Cray-1 had a peak performance of 160 Megaflops and 8 MB of main memory [1], whereas the fastest supercomputer as per the list of the top 500 supercomputers as on November 2014, was Tianhe-2, Guangzhou, China, has a peak performance of 33.86 petaflop per second and a memory of 1,375 TiB [2]. This improvement in performance can be attributed to some of the features discussed below.

1.1.1 Architecture

Introduction of heterogeneous systems have resulted in huge performance improvements through the use of accelerators in commodity systems like Nvidia GPUs. It is important to take into account architectural differences like cache, core connectivity, and I/O channels during development for systems to be backward compatible.

1.1.2 Inter-nodal connection

A few of the most accepted inter-nodal connections are: InfiniBand [3], Ethernet [4] and Omnipath [5]. Statistically in the list of top 500 supercomputers, that was released during Supercomputing 2014, 224 systems currently use InfiniBand, 100 systems use Gigabyte Ethernet, and 88 supercomputers use 10 Gigabyte Ethernet [2].

FDR (Fourteen Data Rate), the latest version of InfiniBand, offers 56 Gbps of bandwidth. The next generation, EDR (Enhanced Data Rate), is expected to offer 100 Gbps of bandwidth. The recent 100 Gigabit Ethernet card can also transfer data at 100 Gbps. A few of the InfiniBands that are about to be released are HDR with 200 Gbits/sec, and NDR with 400 Gbit/sec of bandwidth.

1.2 Programming models in HPC

The hardware improvements, discussed above, required to have an updated programming model. These new programming models seek portability, comprehensiveness, are easy to use and most importantly, one which could deliver the best performance.

1.2.1 Shared memory

In shared memory, parallelism is achieved by using the concept of threads. The commonly used shared memory programming languages like POSIX threads [6], and OpenMP [7], are able to efficiently utilize the fact that the address space is shared and each thread has access to the common address pool. POSIX threads (also known as Pthreads) is a Unix/Linux based library. OpenMP is a compiler directive based programming model.

1.2.2 Distributed memory

In distributed memory programming models, each process has its own address space and data is exchanged among the processes through explicit message passing, such

as the Message Passing Interface (MPI) [8] and Parallel Virtual Machine (PVM) [9]. MPI is the de facto standard for message passing systems. the first version of MPI was released in 1992. In 2015, the MPI forum released the lastest version MPI-3.1.

Two of the most popular implementations of MPI are: Open MPI [10] and MPICH [11]. Open MPI is an open source implementation of the MPI-3 specification, which focuses on component concepts. It is developed and maintained by a consortium of academic, research, and industry partners. Its component architecture provides a stable platform for third-party research and enables the run-time composition of independent software add-ons. Open MPI has three main functional areas:

- Modular Component Architecture (MCA), which provides management services for all other layers,
- Component frameworks, which manages modules,
- Self-contained software units, which export well defined interfaces that can be deployed and composed with other modules at run-time.

A few interesting features of OpeMPI are thread safety and concurrency, runtime instrumentation, and dynamic process spawning.

MPICH is another implementation that uses the MPI specification. One of the design goals of MPICH is merging high performance with portability, which is achieved by the introduction of the Abstract Device Interface (ADI), an abstraction layer, which separates platform specific functionality from user function calls. MPI functions are implemented as ADI macros and functions, which provides portability, ease

of implementation, and an incremental approach to trading portability for performance.

Hadoop MapReduce [12] and Apache Spark [13] are distributed programming models introduced by the big data community. Both of these frameworks target the analysis of a huge amount of data. However, the frameworks differ in the way data are accessed in the memory: the apache spark processes the data in-memory whereas the hadoop MapReduce pushes the data back to the disk after each map/reduce step. Hence, the performance of both frameworks vary as per the requirements and resources available. For example if an application has iterative computation, sparks performs more efficiently than MapReduce, however if the data is too big to fit in the memory available, there is a major degradation in the performance of spark.

1.2.3 Hybrid model

The introduction of clusters of shared memory systems pioneered the hybrid programming language which uses (i) distributed programming for parallelization across node interconnect, and (ii) shared memory programming for parallelization inside each node. This approach is targeted to exploit the scalability of distributed programming model, memory saving, and efficiency of shared memory programming model.

1.2.4 PGAS (Partitioned Global Address Space)

Thus far, all past efforts have concentrated solely on the hybrid models that involve the use of threads on a single node to efficiently use shared memory and message

passing for inter-node communication. However, there was a departure from this notion in the PGAS (Partitioned Global Address Space) languages [14]. In PGAS, it is assumed that the global memory address space is logically partitioned and that a portion of the global memory space is local to each process or thread. The PGAS languages provide locality of reference, since a portion of the shared memory has access to a particular process. The PGAS languages decouple data transfer from synchronization, and create an abstraction over MPI. A few of the commonly used PGAS languages are Unified Parallel C (UPC) [15], Global Arrays (GA) [16], Co-Array Fortran (CAF) [17], and Titanium [18].

1.2.5 Accelerators

Accelerators [19] contain a large number of processing cores and internal memory, and are used in conjunction with the CPUs of the node. They are used to clear bottlenecks faced during computational performance for problems involving algebraic operations. Examples of such accelerators are graphics processing units (GPUs) [20], Cell Broadband Engines (Cell BEs) [21], field-programmable gate arrays (FPGAs) [22], Xeon Phi [23] and other data-parallel or streaming processors. Compared to conventional CPUs, the accelerators can offer an order-of-magnitude improvement in performance for certain operations.

Programming models like CUDA [24] and OpenACC [25] are designed to exploit accelerators to achieve performance benefit. However, the bus bandwidth and the latency between CPU and GPU might create bottlenecks since the accelerator and the CPU do not share the main memory.

1.3 Challenges in HPC

In 1967, computer architect G. Amdahl tried to estimate the upper-limit of performance in a parallel system. As per the Amdahl's law [26], if the fraction F of the application is sequential and $(1-F)$ is a fraction that can be parallelized, then the maximum speed-up $S(n)$ achievable by P processes is given by $S(n) = 1/(F+(1-F)/P)$. Therefore, the performance of an application is constrained by the sequential fraction.

Over the past few years, the number of cores in a system has increased to tens of thousands in number. Unfortunately, if there is a sequential component in the application, the breaking of a problem into multiple tasks does not always lead to better performance.

1.3.1 Performance tuning

The performance of parallel applications depends on application as well as system level characteristics. A set of characteristics which are optimal for one application on a particular platform, are not necessarily optimal on another platform. With the increase in the number of processes, the necessity to tune an application also increases. However, tuning is not a trivial task, and one may face many challenges while attempting to do so. Some of these challenges are:

- Large search space: For many applications the number of the parameters that needs be optimized is very high. Though an empirical search over the parameter space results in the best performance, it is expensive to do so. Consequently,

it is important to have a search strategy that prunes the search space without compromising the quality of the search result, resulting in a reduction of performance overhead.

- Portability: Considering the pace at which developments have recently occurred in HPC, it is impossible to have a programming paradigm which supports all the architectures efficiently. Manual tuning of the application is not feasible as it consumes a substantial amount of time and also restricts the number of systems for which an application can be tuned. Therefore, it is important to have an auto-tuning mechanism which is portable across systems.

There are two aspects in a parallel application which can be tuned, namely, computation and communication. Tuning a communication library for a particular application requires multiple steps, namely: (i) identifying the set of individual and collective operations used by an application (ii) identifying the message lengths used by the application for a particular application scenario; and (iii) tuning the parameters of the occurring individual and collective operations for the message length observed.

The tuning procedure is not as straightforward as it initially appears. Generally, communication libraries such as OpenMPI have a huge number of parameters that can be adjusted to control the communication performance. Deciding on the subset of these parameters to be tuned requires an educated guess from the end-user, since tuning all parameters is not an option due to the time this would take. Applications have more than one relevant message length, each of which would lead to a separate 'optimal' set of parameters for the communication library. Open MPI, for example,

can however only handle one set of parameters per a job, i.e. changing the value of a parameter after launching the MPI job is not possible for most parameters. Thus, the benchmark used during the tuning process has to incorporate all the message lengths used by the application. Ideally, this could be achieved by using the application itself for the tuning. This is however unrealistic in a vast majority of the cases, since the tuning step requires the re-execution of the benchmark/application hundreds or even thousands of times, necessitating execution times of a few seconds at most to keep the time spend in the tuning procedure within reasonable limits. Consequently, most tuning tools rely on simple communication benchmarks such as NetPipe [27] for point-to-point operations and SkaMPI [28] for collective operations.

1.3.2 Performance analysis

To achieve most of the performance benefit in software, it is important to isolate the hot spot and then resolve it. Performance analysis tools in HPC provide a visual understanding of the behavior of machines and their performance by displaying the performance characteristics of the applications. This is performed in three basic steps: data collection, data transformation, and data visualization. Data collection is done using three techniques, namely

- profiles- noting the time spent in different parts of code,
- Counters- noting the number of times an event occurs,
- Traces- collecting continuous details of the application.

Important issues to be considered while selecting the performance analysis tool are

accuracy, simplicity, abstraction, intrusiveness, and flexibility. Some of the challenges faced by performance analysis are discussed below.

- As mentioned previously, high performance systems have become vast, with thousands of nodes, and with each node having one or more multicore processors. The performance analysis tool used for these systems are required to handle systems with a large number of nodes or threads efficiently.

- Increase in system size and the scientific requirements have resulted in increased complexity of codes. Performance analysis tools are required to identify performance bottlenecks in complex codes and tune accordingly to guide re-engineering of applications.

1.3.3 Performance modeling

Predicting future performance of an application is an integral part of HPC, and involves extrapolating the performance. Performance prediction is therefore important to optimize, schedule, verify performance, and procure systems. Performance modeling can be done through the following three approaches, namely,

- Based on mathematical or analytic methods, such as LogP [29], LogGP [30], LogPC [31] and in [32] for I/O.

- Based on tool support and simulation, such as WARPP [33], PACE [34] for communication and PIOsimHD [35] for I/O.

- A combination of the two approaches discussed above. This type of performance modeling is done in POEMS [36], Performance Prophet [37].

Some of the challenges faced in performance modeling are:

- System size, system architecture, processor speed, multi-level cache latency, interprocessor network latency, bandwidth, system software efficiency, type of application, algorithms used, programming language used, problem size, amount of I/O and others affect the performance of an application. Ideally, a performance model should cover each of these factors. This is difficult to design. Additionally, an increase in the level of concurrency (threads, cores, nodes), deeper and more complex memory hierarchies (register, cache, disk, network), mixed hardware sets (CPUs and GPUs) and the increase in scale (tens or hundreds of thousands of processing elements) makes the building of an accurate performance model even more difficult.

- Performance model are based on the assumption that the influencing factors have a steady impact on the application, and therefore fail to handle unreliable performance, caused by shared resources, networks, file system and caching.

1.4 Research goals

Communication operations often represent sections with limited scalability in parallel applications. The main challenge for optimizing communication operations stems from the fact that certain parameters influencing the performance of communication operations are dependent on application as well as platform specific characteristics and can not be easily generalized.

A personalized MPI library [38] allows users to store and retrieve optimal parameter sets for a particular application on a platform. One approach to provide a personalized communication library is to determine a parameter set of the communication library which minimizes the execution time of a given application. To achieve this goal, two fundamental problems have to be solved: (i) efficient optimization of a very large parameter set, since an exhaustive evaluation of all possible parameter combinations might not be feasible (ii) managing, storing and retrieving the optimized parameter sets

An HPC application consists of point-to-point communication as well as collective communication. The parameter set that tunes the point-to-point application might have a different effect on performance of a collective algorithm. Using the performance model [39] we have attempted to arrive at a better understanding both theoretically and practically as to how improvements in the communication time of individual data transfer operations translate to improvements in the communication time of collective operations.

Once we develop an elaborate model on collective communication operations, we can extend it further to the collective I/O operation [40]. The collective I/O comprises mainly of communication among the processes, and the actual reading from or writing to the disk. This often reduces the time spent in I/O by reorganizing data across processes to match the layout of the data on the file system. Goal of this work is to derive models considering the underlying file domain partitioning strategy of the file system and data decomposition of the application.

1.5 Dissertation outline

The organization of the remainder of this dissertation is as follows. Chapter 2 presents a state of art of projects applying dynamic optimization to HPC applications, various performance models and performance model pertaining exclusively to I/O. In Chapter 3 we introduce the concept of a personalized MPI library. In chapter 4 we discuss the impact and limitation of collective communication based on point to point communication. Further chapter 5 covers the modeling of communication in collective I/O. Chapter 6 summarizes the scientific contributions of this work and presents the future perspectives in this research area.

Chapter 2

Related Work

This chapter reviews projects that focus on the tuning or modeling of high performance applications or lay down a foundation that helps in these. We discuss various autotuners, search strategies that can be used to prune the search space and the performance models that represent a part of HPC application mathematically.

2.1 Autotuners

Auto-tuning systems empirically evaluate a search space with all possible implementations and identify the best implementation that meets the optimization criteria. An aggressive tuning process considers a number of optimization features for example input data, machine architecture etc.

Experiments show that the process of tuning depends not only on the platform of the application but also on the software aspects. Thus having a domain specific tuning results in a huge search space. A significant amount of effort has been spent on

autotuning specific applications like Automatically Tuned Linear Algebra Software (ATLAS) [41] and Fastest Fourier Transform in the West (FFTW) [42]. We also have the generalized open framework like OpenTuner [43] that can be re-used across platforms and applications. Auto-tuning projects aims at incorporating the following three features:

- Generality: As pointed out earlier, auto-tuning depends on system-based as well as program-based characteristics, and a seamless integration of both these characteristics are desirable for the tuning.

- Managing overhead: In parallel applications, the search space of the possible optimization contestants is vast and empirically going through this search space takes a huge amount of time. Overhead created by this has to be compensated by improved application implementation, either in the same execution or considering the number of times the application is executed.

- Usability: Auto-tuners are easy to use, as the developer might not be the domain expert.

In this chapter we discuss various auto-tuning strategies, tools available for performance analysis, available predictive models for communication, and in the final section we cover the state of art for I/O performance analysis.
Autotuning can be classified in the following ways:

1. Computational and communication tuning

   In HPC, state of art computers are a collection of compute-nodes clustered together and connected via networks. One way to exploit multiple processes

is to distribute the computational tasks, and have them communicate to each other via a network. This arrangement allows tuning through computation and communication.

Computational tuning involves having a code variant such as loop distribution to divide the matrices efficiently. For communication tuning, parameters that influence the communication among the processes are tested for various values to achieve the best-cost communication.

2. Static and dynamic tuning

   In static tuning, the application is optimized statically prior to the execution. This can be achieved either by having a planner-step where each implementation is tested or by using prediction models to estimate the most optimal version. Though this approach has been widely used in several projects like ATLAS [41] or Automatically tuned Collective communication (ATCC) [44], they do have a few fundamental drawbacks:

   - Often, the tuning of the application takes much longer than the application itself.
   - Several characteristics on which execution of the application depends are available only once the application starts executing, for example process placement, communication volume etc. This delays the tuning.
   - Irregular communication patterns are ignored

Dynamic tuning, on the other hand, tunes the application along with the actual execution, therefore it also considers runtime feature for the tuning operation. Some operations have more than one implementation. In dynamic tuning the best implementation can be selected. Projects like FFTW [42], and ADAPT [45] use dynamic tuning.

3. Library-specific, application-specific and compiler-specific

   According to Basu et al., (July 1, 2013) in [46], autotuners can also be classified as:

   - Library-specific or Kernel-specific autotuners are aimed at deriving tuned versions of commonly used dense linear algebra library functions for various architectures. The focus then switched to kernel specific code generators and finally to domain-specific autotuners. One example of domain specific autotuner is FFTW, this provides algorithms tuned for the underlying architecture and input data. These library and kernel specific auto-tuners lacked the ability to tune the applications that fall outside the existing library.
   - Application-directed autotuner are aimed at for general application support, and have code variants and parameter optimizations at the application level, for example language extensions such as annotations on type of functions. Projects like Active harmony [47] added a separate script containing optimization parameters and their search space, other projects like Intel Software Autotuning Tool (ISAT) [48] uses pragma for tuning regions. Pragma is a preprocessor directive used to provide additional

information to the compiler. Despite fulfilling the goal of generality this approach lacks availability as approach relies solely on the programmer.

- Compiler-base autotuners: Use of a compiler based autotuner is the easiest way of autotuning, since it relies on the compiler to automatically generate code variants to explore parameters that affect parallelism such as loop nest computation, data layout etc.

In this chapter we have followed the classification by Basu et al., (July 1, 2013) to study about different type of autotuners.

2.1.1 Self tuning library generator

Some of the previous projects aim at deriving the tuned versions of algebraic library. In this section, we discuss some of these projects.

Automatically Tuned Linear Algebra Software (ATLAS)

Automatically Tuned Linear Algebra Software (ATLAS) [41] is an implementation of BLAS API [49], to automatically generate the most optimized implementation. This is done using a paradigm called Automated Empirical Optimization of Software (AEOS). AEOS uses empirical timings to decide the best method for a given architecture. ATLAS uses three types of optimization, namely,

1. Parametrization: In an application there are a number of parameters that influence the performance of the application. Parameter optimization is used to examine the parameter space for optimal performance.

2. Multiple Implementation: The tuning of a function is performed by an empirical search over the available implementations of the function.

3. Code generation: A program called code generator modifies the code instructions at the compiler level by considering various parameter and source code adaptations.

Different levels of BLAS use combinations of the above discussed optimization methods. In level 1 and 2 BLAS, each function has its own kernel, making it difficult to have architecture specific optimization. Therefore, parameterization and multiple implementation optimization is used. Optimization of level 3 BLAS uses all the above methods.

FFTW: Fastest Fourier Transform in the West

All the initial attempts of auto-tuning were aimed at reducing the number of operations. It was soon realized that it was also important to study the underlying architecture of the microprocessor to make the application adapt to the hardware details. FFTW [42, 50] aims at tuning the Fast Fourier transform (FFT) as per the underlying hardware.

In this project a flexible implementation of Cooley-Tukey Fast Fourier transform was developed. Planner is the dynamic algorithm, which determines at the runtime the combination of codelets that would result in minimum execution time and floating point operation. These codelets are generated by a smart codelet generator which simplifies the generation of blocks of optimized codelets.

Sometimes a planner might consume a significant amount time, in which case the

planner can eliminate implementations that tend to increase the planner time. In further implementation of FFTW3 'wisdom' a utility to generate wisdom files that saves the information about ways to optimally compute fourier transforms of various sizes was used to add historic learning. The main drawback of this implementation is its lack of flexibility as it could be used only for one problem size. Wisdom also lacks the ability to understand outdated or invalid platforms.

Self-Adapting Large-scale Solver Architecture (SALSA/SANS)

Both SALSA and SANS [51, 52] are aimed at solving large-scale numerical problems by algorithmic selection and its tuning through heuristic-decisions and the incorporation of data-mining and machine learning algorithms.

SALSA maintains a knowledge database and uses this database to estimate the best solver for an application matrix. First, the structural properties of the matrix are used to select the solver, thereafter, various machine algorithms boosting and alternating decision tree are used to improve the tuning. Though this is a statically tuned application, its knowledge base makes it re-usable. In SANS algorithm choices are made dynamically based on input problem data.

2.1.2 Application-directed autotuner

Various approaches have been developed to have code variants and optimization parameters at the application level.

Active harmony

Active harmony [47] is an automated runtime performance tuning system, which help applications to adapt better to a runtime environment resulting in better performance. This is done by modifying the application and library source code. Adaptation controllers along with the tuning algorithm are responsible for adaptability of various parameters.

Each parameter is treated as an independent variable. In any real application, the number of tunable parameters is very high. In these cases the search space is reduced by first prioritizing the parameters. Parameter prioritizing is performed only once per new load, hence it can be reused over several executions. Active harmony performs smart tuning by utilizing previously performed tuning. Even where the exact characteristics are not found, the closest characteristic using linear Euclidean distance is chosen. However in cases where there are no matches with any historic data, extensive tuning is required and this is expensive.

RK-Suite

RK-Suite is a prediction framework [53] for HPC application using a single small-scale application. Important aspects of execution called execution signatures are extracted and to predict the performance an automatic phase identification is done by comparing the signatures to the reference kernels which are benchmarks from various application domains. In case there is no match found a full static analysis is done. The framework called RK-Suit consists of

- RK-Collection: This is a collection of reference kernels.

- RK-profiles: RK-profiles consist of execution profiles, cache hit and miss, obtained by benchmark run of reference kernel for a fixed problem size and a fixed number of processes.

- RK-model: This is a performance model to predict execution time of kernel implementation of different problem size and processors.

RK-suite is a database that stores information like execution time of kernel benchmark, performance-model, and the execution signature.

Application signature matching is done by comparing normalized instruction histogram of kernels and the application, and then calculating the distance, in order to determine similarity. For the prediction of execution time, the ratio of problem size is taken into consideration. For out-of-reference application phases, compiler based analysis is done to derive the most accurate performance model.

OpenTuner

OpenTuner [43] is an open source framework for tuning multi-domain projects. It is an ensemble of search techniques run at the same time. Techniques which perform badly are either entirely shut off or given fewer tasks. OpenTuner was tested by building tuners for seven projects with 16 benchmarks. Recently, OpenTuner was used to autotune Java Virtual Machine (JVM) [54].

2.1.3 Compiler-directed autotuner

In compiler-directed autotuning, the compiler automatically derives the code variants and optimization parameters.

ADAPT

ADAPT [45] is framework for dynamic program optimization. ADAPT decouples the compilation of code-variants from a dynamic selection of these variants. Dynamic compilation overhead is avoided by overlapping the generation of code variants and their execution.

The new code variants are also available to dynamic selection mechanism. At runtime, the dynamic selection mechanism tests the variants to determine the fastest implementation. ADAPT prunes the search space of program variant using the conditions defined by the user.

Optimization is done using granularity of intervals. For example loops which have a single entry and exit point are optimized using operations like loop distribution, loop unrolling, loop tiling, automatic parallelization, and other compilers optimizations. ADAPT also provides a number of services like:

- Code triage- the hot-spots are optimized,
- Environment monitoring- to adapt to the environment efficiently,
- Dynamic selection: to select the code variants as per the interval descriptor and code variant descriptor.

Though the main advantage of ADAPT lies in its capability to generate code variants and load them at the runtime, this can be done only on sequential or multi-threaded application and it lacks functionality at a parallel application level.

2.2 Autotuning of MPI libraries

Most of the parallel applications in HPC use the MPI library for inter-nodal communication. Implementations of the MPI libraries have parameters that provide a platform to optimize performance as per the characteristics of the architecture and the application. In this section, we discuss some of the projects that concentrate on the optimization of the MPI parameters.

ANOVA

One way of optimizing the performance of an MPI application is by tuning the runtime parameters. In [55], analysis of variance (ANOVA) is performed on randomly explored MPI parameter values. Most of the optimization techniques involve either iterative feedback-driven tuning (IFT) or iterative compilation (IC). However both of these take a large amount of compilation/optimization time and are also input-data sensitive. To overcome these drawbacks, architecture-specific parameter values based on computational kernels like stencil operations, Fourier transforms, are optimized for certain parallel architectures. Applications using the same communication pattern as these kernels can use this information for performance improvement. This method consist of two phases:

- Exploration phase: In the exploration phase, runtime parameters for a set of computational kernels are tuned on a target architecture. This is a one-time step, which is performed only when the cluster is deployed,
- Parameter tuning phase: In the parameter tuning phase, gathered data is analyzed using statistical method of ANOVA (Analysis of Variance).

The output parameter set replaces the default settings of MPI library.

STAR MPI

STAR MPI [56] works on a set of MPI collective implementations that can adapt to both system architecture and application workload. Applied AEOS (Automatic Empirical Optimization of Software) is used to select the optimal collective algorithm. Until the platform and application is known algorithm selection is withheld. This is called delayed finalization. MPI program are linked to STAR-MPI (that has implementation of all the routines). Each collective routine is supported by N implementations. Decisions are taken in two stages:

- Measure-select- One routine is selected to realize the operation and performance,
- Monitor update- All the algorithms are executed, and a decision is made using an all-reduce operation across processes.

If there is a deterioration in the performance, 'monitor-adapt' is used, i.e. STAR-MPI continues monitoring the performance of the selected algorithm and adapts (changes the algorithm) accordingly.

MPI advisor

While tuning the MPI application, MPI advisor [57] addresses four potential bottlenecks in an application: point to point protocol (eager or rendezvous), collective communication algorithm, MPI task-to-core mapping, and Infiniband transport protocol.

The framework automates the following steps:

- process of data collection using PMPI and MPI_T tools (described in detail in section 2.4),
- analysis that translates collected data into performance metrics and,
- recommendation for optimization.

Architecture specific data are collected during installation, and application specific data are collected during profiling. To make the tuning decision application specific data is compared to pre-defined heuristics.

For each collective operation employed, a table is built at the time of installation using CE script that gives the best algorithm in each MPI library installed on the system. Though the MPI advisor is extremely efficient in tuning the application, it ignores computation and communication patterns, and the load condition of the system.

Automatically Tuned Collective communication (ATCC)

ATCC (Automatically Tuned Collective Communication) [44] optimizes collective communication pattern. It has an extensive optimization step to extract the most optimal implementation from the available pool of algorithms, on a particular the platform.

Abstract Data and Communication Library (ADCL)

ADCL [58–60] is a communication library that aims at providing the best performing implementation of the communication operation in a parallel application on a given

platform at the runtime. The library tests a number of implementations of an operation, using runtime selection logic. There are three selection strategies presently available in ADCL, namely, (i) Brute force selection strategy, (ii) Attribute based selection logic, and (iii) 2k factorial selection strategy. By default, brute force search strategy is used to test the implementations, i.e. each and every implementation is tested.

ADCL first extracts the information regarding application level communication pattern, i.e. repeatedly occurring communication. To switch across various implementations of the communication operation with an application, ADCL API is used. API is high level interface of communication operation and consists of:

- `ADCL_topology`- `ADCL_topology` which contains information like process topology and neighborhood relations in the given application.
- `ADCL_vector`- `ADCL_topology` which is a data structure to be used during communication.
- `ADCL_request`- `ADCL_topology` which is a bridge between ADCL-topology and ADCL-vector.

Most of the parallel applications have the following two characteristics that ADCL takes advantage of:

- Iterative execution- Applications having large loops and the same code sequence is executed repeatedly.
- Collective execution- Applications based on data decomposition, that is the same code is executed for different sets of data.

The most interesting feature of ADCL is the integration of the planner stage and execution of the application itself. Libraries like FFTW and ATLAS also aim at optimization but they invoke a separate 'planner' stage, prior to the actual execution of the application, thus creating overhead.

To minimize the time spent in the testing stage, ADCL also supports historic learning [61] for pre-defined operations, i.e. if the operation is executed multiple times ADCL stores the problem characteristic and performance data in an xml .adcl file.

Open Tool for Parameter Optimization (OTPO)

Open MPI is an open source implementation of MPI, and is jointly maintained by consortium of academia, research institutions and industry. One of the interesting features of Open MPI is the availability of a software layer called MCA (Modular Component Architecture), which handles the management services for the Open MPI framework, for example the crossover point of the collective operation algorithm, message length at which point to point communication switches from eager to rendezvous protocol. One of the interesting feature of MCA parameters is that it can be passed onto the MPI library and influence the application, without recompiling the library, thus has been used extensively.

OTPO aims at the optimization of these runtime parameters by the end user. Internally, OTPO relies on the **Abstract Data and Communication Library (ADCL)** for the dynamic tuning procedure, including the search methodology used, data filtering to exclude outliers and the decision logic. For this, OTPO translates Open MPI MCA parameters into ADCL attributes, and creates a function-set which

executes a benchmark/application with different values for the requested MCA parameters.

OTPO first parses the input file, and stores all the parameters and their probable values in a structure. Only a subset of the parameter set, which satisfy the RPN (Reverse Polish Notation) are tested for the optimization. Latency of all the successful executions are updated by the ADCL. Once finished, OTPO gathers the results and saves them in separate files. OTPO then extracts the parameter set with the most optimized latency as the output.

Quadtree

Various methods have been studied to select the most optimized collective operations implementation. In [62] quadtree was used to tune the collective algorithm for the particular system. Quadtree is a decision tree where each node has four children nodes. System profiles and communication models were used to analyze and generate the decision function in-memory. One of the limitations is its lack of performance for single communicator value communicator sizes, which are power of 2. The same problem worsens if the performance measurement data used to construct trees is too sparse.

2.3 Search strategy

The search space in the auto tuning of parallel application is often very vast and requires a large amount of time to test. Some of the methods which are available to prune the search space are discussed in this section.

Brute force search strategy

Brute force search strategy is an exhaustive search that enumerates all the possible candidates for the solution and checks if each of them satisfies the condition. This search strategy guarantees the best available solution.

Real world parallel applications tend to have a huge search space and to evaluate all the possibilities tend to take a huge amount of time and cost. Though in many scenarios, this can be done for the first time, but it is highly impractical for usage in the long run. There are various other methods that can be used to restrict the search space, still extracting the 'optimal' possible solution.

Statistical models for empirical search-based performance tuning

In [63] the author address two problems, developing a method to stop the exhaustive search once the near-optimal solution is achieved, and the construction of the runtime decision rules based on varying features. The search space is generated by collecting system based and application based features.

Instead of pruning the search space using heuristics or a performance model, the early stop decision is preferred, which is based on knowledge gathered during the search. On-line estimation of the distribution of the search space output is made. This provides the user an option to stop the search once the implement output is within the acceptable limit.

Sometimes more than one implementation can result in an optimal performance. This issue can be resolved by using three types of statistical models, which are discussed below:

- Parametric data modeling- Execution time is estimated using linear regression on training data.

- Parametric-geometric modeling- Hyperplanes are used to create boundaries as per the input parameters. Parameters such as slope and intercept of each boundary is determined using training data.

- Nonparametric-geometric modeling- The support vector method is used to construct a nonparametric model. Nonparametric is a representation of the implicit models of the boundary.

Machine learning to optimize MPI runtime parameter settings

In [64] the author discusses two machine learning mechanisms to extract the optimal MPI runtime parameter. In the offline training phase, a predictor is built using machine characteristics. The predictor estimates the best setting of any MPI program. These MPI programs have static features extracted by studying the source code and dynamic features extracted by executing the program once. The author uses a 'decision tree' and an 'artificial neural network' to predict optimal MPI parameters. The whole process is divided into two phases:

- Learning phase- A set of training programs is executed on target architecture using several sets of chosen MPI parameters.

- Optimization phase- An MPI program is fed to the predictor, and the program features are extracted. The optimized configuration is then determined using the machine learning algorithm.

Attribute based/orthogonal search strategy

Attribute based/orthogonal search strategy is an iterative search strategy, where one attribute is optimized while keeping the other attributes constant. Once the most optimized value of one attribute is achieved, the next attribute is tested for the optimization, while maintaining the most optimized value for the first attribute and a constant value for the others. The major drawbacks of this method are

- It assumes that all the attributes are independent.
- The order in which each attribute is optimized is very important.

2k factorial design algorithm

The aim of this method is to reduce the number of attribute to only the important ones. For uni-directional attributes, two extreme values are selected and application is executed for them. Using a non-linear regression formula, algorithm calculates the weightage that each parameter carries in the system. However, a threshold of the minimum weightage has to be pre-decided as removing parameters beyond a threshold compromises the most optimal attribute set.

Genetic algorithm

Genetic algorithm is a adaptive heuristic search, and is generally used for optimization and search problems based on biological evolution. A population of candidate solutions are made to go through the iterative process of evolution to fit better. The genetic algorithm has two important features:

- A genetic representation of solution domain.
- A fitness function to evaluate the solution domain.

The genetic algorithm starts with a set of solutions, called population. Using the genetic function a new population is generated, how well the new solution fits is determined by the fitness factor. In case the fitness function does not satisfy the pre-determined condition, same steps are applied to another generation.

Genetic algorithms are very robust, i.e. even if inputs change slightly, or in a reasonable amount of noise, it performs well. Additionally, it performs better than other optimization techniques in large dimensional spaces.

Tabu search

Tabu search [65] is a metaheuristic search method employing local search methods for optimization. The local search method or the neighborhood search is a method of improving the solution by comparing the achieved solution with its immediate neighbors. In other words, solutions that are similar save some minor details.

One of the drawbacks of tabu search is its tendency to get stuck in local minima. As a solution, moves that worsen the solution are accepted initially. To avoid the algorithm coming back to same neighbor, a rule of prohibition was introduced, i.e. the neighbors already tested are marked as 'tabu'.

The traveling salesman problem is the most commonly used example for tabu search, in which the shortest route that visits every city is calculated.

Discussion

ADCL and OTPO follow the same search strategies: Brute force, Attribute based and 2k factorial. ADCL has the ability to select the fastest available implementations for a given communication pattern during the regular execution of the application, however the decision has to be pre-determined as value of MCA parameter cannot be modified during the execution.

2.4 Performance analysis tools

With the increase in complexity of HPC applications, it is important to have advance performance analysis tools. These tools are used to:

- compare system performance,
- detect anomalies,
- optimize and to predict the cost.

In this section, we will discuss some of the performance tools that were used to understand the performance of parallel applications.

Tuning and Analysis Utilities (TAU)

Tuning and Analysis Utilities [66] is a portable profiling and tracing toolkit for analyzing parallel programs. TAU framework architecture has three layers: instrumentation, measurement, and analysis. TAU provides a flexible instrumentation model that allows the user to insert performance instrumentation at various compilation

and execution stages. An interesting TAU instrumentation is the 'selective instrumentation', a facility to record a list of performance events to be included or excluded by the instrumentation in a file.

TAU profile characterizes the behavior of an application in terms of aggregate performance metrics. TAU profiles do not capture the time related aspects during execution. To study these spatial and temporal aspects, event traces are used. However the choice of performance events is very crucial and alters the way a program behaves.

Vampir

Vampir (Visualisation and Analysis of MPI Resources) [67] is a trace visualisation tool used to analyze message events when data is communicated among the processes during the execution of a parallel program. Some events that can be analyzed using Vampir are event ordering, message lengths, and times. Its latest version (5.0) features support for OpenMP events and hardware performance counters.

The tool comes with two components - VampirTrace and Vampir. VampirTrace has a library which when linked and called from an application, produces an event tracefile. Common events include the entering and leaving of function calls and the sending and receiving of MPI messages. Traces can be generated for the whole application, or the user can manually select the critical area by adding calls to `VT_USER_START` and `VT_USER_END`. Vampir is used to convert the trace information into graphical views, e.g. timeline displays showing state changes and communication, profiling statistics displaying the execution times of routines, and communication statistics indicating volumes and transmission rates.

Scalasca

Scalasca [68] is a software tool used to optimize the performance of parallel programs by measuring and analyzing their runtime behavior.

In [69] the author discusses the effect of a wait state or delay on large scale operations. The delays of a single process may increase as the number of processes increase. In parallel applications, delays manifest as wait-states, i.e. temporal intervals during which processes are waiting for synchronization. Scalasca measures temporal displacement between matching communications that have been already recorded in the event traces. To perform this operation, all the traces are analyzed in parallel, and a global analysis report is made.

PMPI

PMPI [70] is an MPI standard profiling interface. Each standard MPI function can be called with a PMPI_ prefix, for example, MPI_Send() can be called as `PMPI_Send()`. This facilitates MPI performance analysis in following two ways:

- Many performance analysis tools use PMPI. This is done by associating each MPI call with PMPI function, which captures the performance data,
- This can also be used to customize MPI function by using wrapper functions.

MPI_T

MPI standard MPI 3.0 includes a new interface called the MPI Tools Information Interface, or MPI_T [71]. MPI_T provides access to both internal performance information and runtime settings. It allows each implementation to decide what information to expose and then provides an interface for users to query what information is available, how it is offered, and what metadata is associated with it.

2.5 Performance models

Collective operations are a major part of any parallel application. A large number of implementations of these collective algorithms are available. The optimal selection of the algorithm depends on a number of factors. For example, message size or the number of processes that are involved. Any extensive tuning process of the whole search space would take an enormous amount of time, thus one of the faster methods is to use the predictive model to estimate the communication cost. Some of these models like Hockney, LogP, LogGP, and PLogP are discussed below.

Hockney's model

In 1993, Roger Hockey [72] emphasized on the criticality of speed of communication among the nodes. He also pointed out the gaps that are present between the consecutive messages. Thus, to improve the performance of message-passing in parallel applications, we need to increase the communication speed, decrease the communication delay, and recognize and shorten the gap between the messages.

In Hockey's model two major factors decide the cost:

- The startup time, which determines the short-message performance,
- The asymptotic bandwidth which determines the long-message performance.

Mathematically, time spend to send a message of size m bytes between two interconnected nodes is given by $L + Bm$, where L is the network latency for each message, and B is the transfer time per byte or reciprocal of the network bandwidth. Though this model efficiently represents the simple communication, it lacks the ability to model complex communication and network congestion.

LogP model

In [73] the author aims to have a simple communication model which also represents the critical technology trends that might create bottlenecks in the communication. The LogP model emphasizes on number of processors (P), communication bandwidth (g), communication delay (L), and communication overhead (o). Some of the assumptions maintained to analyze this model are

- Only small, constant-size message are transmitted,
- A finite capacity of network: (L/g) is the maximum number of messages that can be transmitted at any point of time,
- The model is asynchronous, as latency is unpredictable, message might not arrive in the same order as it was sent.

Mathematically, the time taken to communicate a message between two node is $L + 2o$. Contrary to the assumption made in the LogP model, generally communication messages are not short. The logGP model [30] is an extension of the LogP model with a linear model for longer message length. An additional parameter 'G', the gap per byte for long messages was added. LogGP model predicts the time to send a message of size m between two nodes as $L + 2o + (m - 1)G$.

One of the extension of LogP model is PlogP (Parameterized LogP) model [29]. As per PlogP model, a network is a function of five parameters:

- P- number of processes,
- L- end to end latency, includes time required to copy data to and from network interfaces and to transfer over physical network,
- $o_s(m)$- send overhead,
- $o_r(m)$- receive overhead,
- $g(m)$- minimum time interval between consecutive message transmission or reception.

Here m is the message length. In case of long messages, it is assumed that receive operation might start before the end of send operation, in such cases o_s and o_r might overlap. As g is considered to include o_s and o_r, it can be concluded that $g\ (m) \geq o_s(m)$ and $g(m) \geq o_r(m)$. The time to send a message of size m between two nodes in the PLogP model is $L + g(m)$. If we assume $g(m)$ to be a linear function of message size m and the latency excludes the sender overhead, then the PLogP model

is identical to the LogGP model which distinguishes between sender and receiver overheads.

In [74] the author explains the lack of a parameter that attributes to synchronization required for long messages in the rendezvous protocol. Hence in LogGPS, an additional parameter, S, threshold for message length above which messages are sent in synchronous mode was added. In [75] the authors further extended the LogGPS as LogGOPS, with parameter O, which models the overhead per byte. LoGPC [31] is another extension of LogP and LogGP with a parameter that captures the network contention and network DMA behavior.

2.6 Collective I/O performance modeling

According to Ken Batcher "A supercomputer is a device for converting a CPU-bound problem into an I/O bound problem". As I/O is extensively used in HPC, we often face I/O as one of the limiting factors in the overall performance improvement of the application. To overcome this limitation, we need to understand and tune the I/O. Historically, collective I/O operations have mostly been tuned through empirical testing, only recently research has focused on cost models to tune these operations.

A Cost-intelligent application-specific data layout scheme for parallel file systems

Song et al., (June 08 - 11, 2011) [76] present a method to predict the cost of data access for various file systems and chose the data layout strategy accordingly. Their focus was on investigating various data layout methods, i.e. the algorithm

used by the parallel file system to decide where to store a given data block. The authors evaluated three data layout methods, 1- dimensional vertical, 1- dimensional horizontal, and 2- dimensional layout, and developed a cost model that takes both communication and I/O into account. The main shorting coming however is, it does not utilize state-of-the-art models for communication operations.

Non-exclusive and hierarchical I/O scheduling

In [77] the authors consider location of aggregators when tuning collective I/O operations for minimizing the communication costs and network congestion. Liu et al., (13-16 May 2013) [78] developed a model to schedule communication such that the slowest aggregator is served first. The authors of [79] developed algorithms that take memory utilization by individual processes and nodes into account, and thus minimize the memory pressure due to collective I/O operations.

Analytical and machine learning models

The work closest to our approach is discussed in [80]. A performance model for collective I/O operations is presented that encompasses both communication and storage operations. The total communication time during the data exchange is modeled as the summation of individual times taken by collective communication operations (`MPI_Allreduce`, `MPI_Alltoall`, and `MPI_Alltoallv`) and the actual data exchange. However, the authors do not take the impact of the data distribution strategy used by the application into account, nor the file domain partitioning strategy used by the MPI I/O library.

PIOsimHD

Some researchers work on understanding the performance on complex high end systems through simulation environments. PIOsimHD [35] is an event based simulator for parallel applications. The simulator helps in evaluating the performance of collective communications, parallel I/O, cluster hardware and allows us to extrapolate this to the performance of an application.

Chapter 3

Tuning of communication parameters in parallel applications

Communication operations often represent sections with limited scalability in parallel applications. For example, a stencil communication pattern has constant execution time, assuming that the number of communication partners per process does not change with the overall process count. Even worse, an all-to-all communication operation scales quadratically with the number of processes. Taking into account that the time spent in computation decreases for a fixed problem size with increasing number of processes, both scenarios represent a challenge to Amdahl's law and highlights the necessity to minimize the time spent in these operations. The main challenge for optimizing communication operations stems from the fact that certain parameters influencing the performance of communication operations are dependent on application- and platform-specific characteristics and can not be easily generalized. For example, the cross over point between various algorithms of a collective

operation, or the switching point from an eager to a rendezvous protocol for point-to-point operations have typically been determined using very limited testing, and have not explored all feasible combinations (of message length or number of processes). Similarly, the optimal lengths of internal send/receive queues and temporary buffers utilized by the MPI library might depend on the actual number of communication partners of a process, message lengths and the number of messages within a time period. Ultimately, a one-size-fits-all solutions for MPI libraries will be difficult to maintain for exascale environments: MPI libraries will have to be customized for specific applications and platforms to maximize their performance.

3.1 Tuning of parallel applications

There are multiple different strategies that can be applied for tuning MPI applications. The first method, the **direct tuning** of the MPI application assumes that the application itself can be used as a 'benchmark' for the tuning tool. In this approach, all parameters that the user considers relevant are tuned simultaneously, assuming that the search algorithm correctly identifies the most relevant ones. The downside of the direct tuning is that each individual execution of the benchmark (i.e. the application) takes significantly longer than in the hierarchical tuning, thus increasing the overall time it takes to tune the application.

The second approach, the **hierarchical tuning**, relies on micro-benchmarks to tune a particular aspect of the application. The results of each optimization is a smaller set of parameters. In the final step, the application itself might still be used for evaluating a smaller set parameter values. To explain the hierarchical tuning,

consider a scenario in which the end user identified initially the most commonly used message length in his application, as well as the collective operations used within the code. This can be done either manually, i.e. using some code written by the user, or using a tool such as TAU [66] or Vampir [67]. Point-to-point operations for the relevant message length can now be the NetPipe benchmark for all network transports of a given platform, e.g., TCP, InfiniBand, and shared memory, resulting in tens of possible parameter sets. These parameter sets are then used as an input to a benchmark that uses more than just two processes for the same message length, e.g., SKaMPI, resulting in a further reduction of the parameter sets. In addition, SKaMPI can be used to tune the occurring collective operations. The resulting set of InfiniBand, Shared Memory and collective parameters could be used to perform a final tuning step using the application itself.

One of the foremost problems when using the hierarchical tuning stems from the fact that most applications are using more than one message length. The intuitive approach to tune all occurring message lengths using NetPipe is impractical for two reasons: first, the number of different message lengths might be very large, and second the resulting configuration files might lead to contradicting parameters. Ultimately, one has to determine a small number of message lengths that are considered to be the most influential ones for the application performance. The direct tuning on the other hand uses a benchmark, i.e. the application itself, which incorporates exactly the message lengths used. Thus, the resulting parameter set should, under optimal circumstances, result in the lowest execution time for the entire application. Note, that the resulting parameter set will minimize the combination of all

message lengths used, and might not be optimal for any single message length. In addition, direct tuning allows the end user to skip the initial step of identifying the most frequently used message lengths. It might, however, be beneficial to identify the collective operations being used by the application to minimize the initial input parameter set to explored by OTPO.

3.2 Tuning of communication operations

Tuning the performance of communication operations in a parallel application requires multiple steps. In the first step, users have to generate a profile of their application in order to identify the most commonly used individual and collective operations as well as the dominant message lengths used in a particular application scenario. The second step consists of tuning parameters of the communication library for this particular application (scenario) and retrieve a set of parameters that minimizes the communication time. For the actual production runs, the user provides the parameter sets that were deemed to be optimal to the application manually or in a semi-automatic manner.

Tuning the parameters of Open MPI poses, however, multiple challenges. Deciding which of the over 400 parameters to tune requires some knowledge of the internals of the communication library and platform, since despite of using advanced search algorithms, tuning all parameters is not feasible. Furthermore, applications will have more than one relevant message length, each of which would lead to a separate 'optimal' set of parameters for the communication library. Open MPI can, however, only handle one set of parameters within a job, i.e. changing the value

of a parameter after the job has been launched is not an option except for very few parameters. Thus, the benchmark has to utilize all the message lengths used by the application. This could be achieved by using the application itself for the tuning. This is, in the vast majority of the cases, unrealistic, since the tuning step requires the re-execution of the benchmark/application hundreds or even thousands of times, necessitating benchmarks that take a few seconds per execution at most to keep the time spend in the tuning procedure within reasonable limits. Hence, most tuning tools – including OTPO – rely on simple communication benchmarks such as NetPipe [27] for point-to-point operations and SkaMPI [28] for collective operations.

Using microbenchmarks for the tuning step reduces the time spent in the tuning operation itself, the resulting parameter sets are not necessarily optimal from the application perspective. Optimizing parameters of the InfiniBand `btl` component of Open MPI for a given message lengths show in very few scenarios the expected performance improvement, despite of significant performance improvements observed for a simple ping-pong benchmark. Even for relatively simple scenarios, e.g., a simple benchmark executing an All-to-all communication operation, performance benefits could be observed for some algorithms used to implement the collective operation, but not for others.

3.3 Sensitivity of the network parameters to the message length

In this section, we present a use-case scenario in which we analyze the sensitivity of the Open MPI point-to-point performance to a set of runtime parameters depending on the message length. For this, OTPO was used to tune a set of seven parameters of the `openib` module using the NetPipe benchmark for various message lengths, and the improvement was compared to the default performance of Open MPI when these parameters were set to their default values. Tests in this subsection have been executed on the crill cluster at the University of Houston using Open MPI 1.8.1. The crill cluster consists of 16 nodes with four 12-core AMD Opteron (Magny Cours) processor cores each (48 cores per node, 768 cores total) and 64 GB of main memory per node. Each node further has two 4X DDR InfiniBand Host Channel Adapters (HCAs), although only one HCA has been used in the subsequent tests. The openib and shared memory (sm) parameters tuned and the ranges/values explored for each parameter are shown in Table 3.1.

Table 3.1: Open MPI parameters tuned in the sensitivity analysis

openib parameters		
Parameter Name	Default value	Value range
btl_openib_eager_limit	12K	1.5K:48K:*2
btl_openib_rdma_num	16	1:32:*2
btl_openib_rdma_threshold	16	4:32:*2
btl_openib_use_eager_rdma	1	0,1
btl_openib_use_message_coalescing	1	0,2
btl_openib_free_list_num	8	2:32:*2
btl_openib_free_list_inc	32	8:64:*2
sm parameters		
Parameter Name	Default value	Value range
btl_sm_max_send_size	32K	16K:128K:*2
btl_sm_fifo_size	4K	1K:16K:*2
btl_sm_num_fifos	1	1,2,3,4
btl_sm_free_list_num	8	4:64:*2

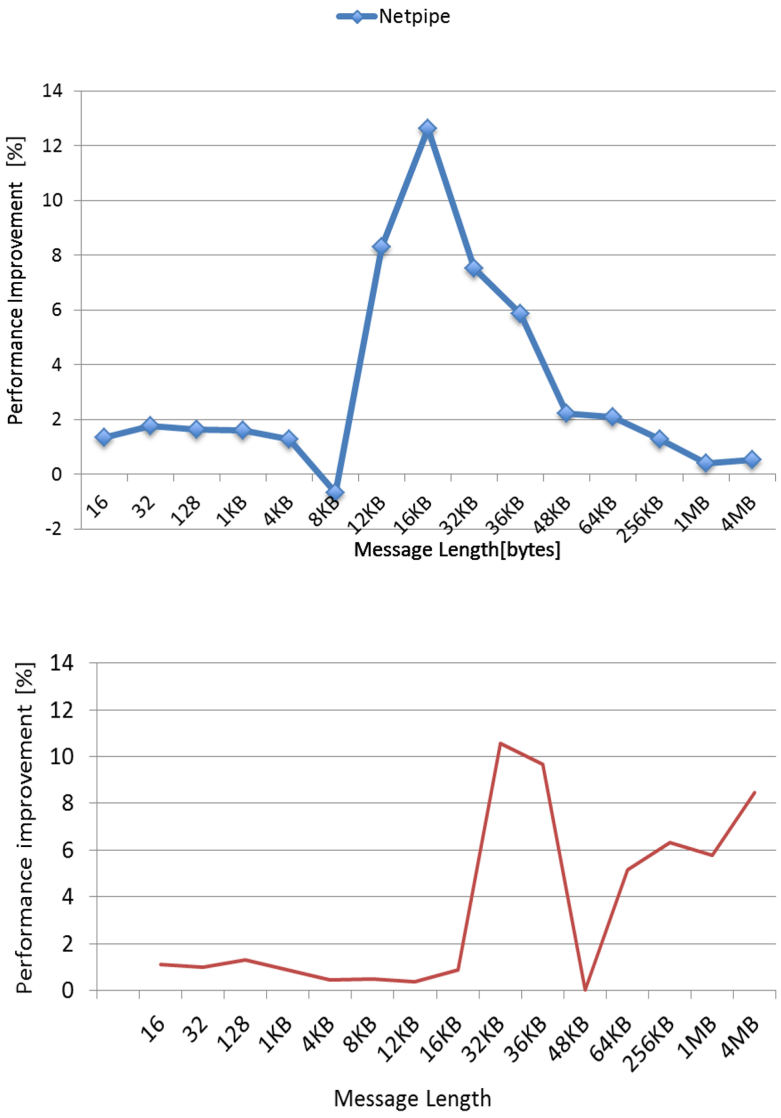

Figure 3.1: Sensitivity of the Open MPI point-to-point performance to openIB (top) and SM (bottom) parameters.

The results shown in the top part of Figure 3.1 indicate, that the performance of Open MPI over this InfiniBand network interconnect can be improved by up to 10% using optimized values for these parameters, but only for messages in the range of 12 KByte to 36 KByte. The performance of Open MPI can not be improved significantly using these parameters for very short and very long messages. The bottom part of Figure 3.1 shows the results of the same test for shared memory communication between two processes using the sm component of Open MPI. Four parameters of this component were tuned in tests shown above. The results differ from the openib results since the performance of the sm component showed some sensitivity to the parameters starting from 32 KByte message length. The performance improvement is in the range of 5%, with the 48 KByte messages being a notable (but reproducible) exception.

To demonstrate the implications of improving the performance of a point-to-point operation by a small percentage, consider the results shown in Table 3.2. The table documents the execution time of three different all-to-all communication algorithms (linear, pairwise exchange, and Bruck's algorithm) for a message length of 16 KByte, the message length which showed the largest sensitivity for the openib component according to Figure 3.1. Results are shown for alltoall operations using 32 and 128 processes and the relative performance improvement when using the optimized parameter sets. Each data point shown in Table 3.2 is the average of multiple runs (typically between two and four runs), with minimal variations seen between individual executions of SKaMPI. The results indicate significant performance improvements when using the optimized openib parameter set for the linear and the

pairwise algorithm, with performance benefits ranging from 20% to over 60%. However, Bruck's algorithm does not show improvement when using the optimized openib parameters. This is discussed in subsequent sections. This observation compelled us to further study the performance benefit for various implementation of collection operations, which is discussed in the next chapter.

Table 3.2: performance benefits of SKaMPI all-to-all with modified runtime parameters compared to the default.

Algorithm	no. of procs.	Default Exec. Time [ms]	Tuned Exec. Time [ms]	Relative Gain [%]
All-to-all linear	32	40.7	27.1	33.2%
All-to-all pairwise	32	31.9	25.1	20.6%
All-to-all bruck	32	56.6	56.3	0.4%
All-to-all linear	128	918.6	320.9	64.4%
All-to-all pairwise	128	35.9	26.5	25.9%
All-to-all bruck	128	57.8	57.7	0.2%

3.4 A personalized MPI library

In order to close the loop from the end-users perspective, a mechanism has to be developed that allows users to store and retrieve optimal parameter sets for a particular application. For this, a database has been deployed. A parameter set can be uploaded into the database using a tool developed as part of the OTPO framework. The tool connects to the database, and uploads a parameter file and additional metadata which will be used to query for a parameter set subsequently. The metadata consists of:

- **host key**: a unique identifier for a host or cluster. This key is required to distinguish between different parallel platforms which might all connect to the same database system to store/retrieve parameter values. The host key can be as simple as the name of the cluster or front-end node, or the public ssh host key of the front-end node (which might not always be applicable),

- **application key**: a key uniquely identifying a parallel application. In the initial stages, it is assumed that the application key will typically be a string containing the application name provided by the end-user, or a combination of an application name and additional qualifiers which allow to identify the use-case that the parameter set has been tuned for. It can be extended to provide automatic application key generation by utilizing a combination of checksum calculation on the source code/executable and of the communication profile of the application use-case,

- **application characteristics**: list of characteristics necessary to identify a particular use case. This could include the number of processes, message length or the application problem size.

The Open MPI parameters are stored as simple key-value pairs, the key representing the MCA parameter stored as a string. To retrieve a particular parameter set, the `mpiexec` command of the Open MPI library has been extended by a number of additional parameters. The parameters indicate the address of the database, login information as well as the host key and the application key to be used. When parsing the parameter files, Open MPI will try to contact the database and retrieve the parameter set for this particular scenario. The resulting parameter set is written

to a local file, and will be processed by all MPI processes similarly to the default parameter file.

While the current implementation is a proof-of-concept demonstrating the usefulness of the approach, the system is being extended to allow for more generic querying of parameters. For example, by introducing different categories of runtime parameters in the database, one could query for networking parameters for a given application, even if no parameter set is available for the current number of processes being utilized. Similarly, one could query for settings of collective algorithms of other applications utilizing the same number of processes and message length.

Chapter 4

Impact and Limitations of Point-to-Point Performance on Collective Algorithms

The goal of this work is to establish better understanding both theoretically and practically on how improvements in the communication time of individual data transfer operations propagate to collective operations. The initial focus is on collective operations since they (i) represent important building blocks of many application, (ii) are well understood multi-process communication patterns, and (iii) use a single message lengths and thus allows us to isolate the problem that we tackle from the multi message-length problem described above.

The starting point of the analysis is a parameter set for Open MPI that improves the execution time t of a point-to-point communication operation of message length m by a factor of i^{p2p}. The problem that we address can be formulated as follows:

given m, i^{p2p}, and the number of processes p, what is the expected and the observed performance improvement for a collective operation for using the 'optimal' parameter set for the message length m vs. using the default values. We focus on three collective operations with multiple algorithms implementing the operation, namely:

- *broadcast*: chain, binary tree, binomial tree,
- *all-gather*: ring, neighbor exchange, recursive doubling,
- *all-to-all*: linear, pairwise exchange, Bruck's algorithm.

4.1 Modeling improvements of collective operations

In the following, formulas to estimate the expected improvement of a collective operation, given an improvement in the performance of point-to-point operations are derived. For this, we focus on the (simple) models that allow us to derive formulas for the expected improvement of a collective operation. Specifically, we used Hockney's communication model [72] and the LogGP [30] communication model. Both models have been introduced in section 2.5, the most important aspects of the models are presented here for the sake of completeness. Although some other models such as LogGPS [74] or LogGOPS [75] have shown higher accuracy in modeling, the performance of communication operations, providing a consistent and simple formula for a particular operation is often challenging in those models.

In the Hockney's model, the time to send a message of size m between two processes is $l + m/b$, where l is the network latency, and b is the network bandwidth. Though very simple, this model has its limitations when modeling complex communication patterns.

The LogGP model is a more detailed model with four parameters, namely:

- L: communication delay, which is the hardware latency of the network (in contrary to the latency in Hockney's model, which is an end-to-end latency),
- o: the software communication overhead i.e. the time spent in the MPI library before injecting data into the network,
- g: the gap parameter, which is a hardware parameter dictating the minimum time before being able to inject two subsequent messages into the network,
- G: Gap per byte, which is the reciprocal value of the network bandwidth.

According to the LogGP model, the time to send a message of size m between two nodes is given by $L + 2o + (m - 1)G$.

The starting point of this work is to estimat the costs of all algorithms mentioned previously, and is based on previously published work by multiple research groups. The formulas for this are listed in Table 4.1. Two minor modifications were made for the cost estimates taken from [81]: the memory bandwidth parameter δ has been eliminated, and for the sake of simplicity it is assumed that the number of segments used in some algorithm is one (which matches the implementation used).

Table 4.1: Communication costs of various collective algorithms using Hockney's and the LogGP model.

Collective Operation	Algorithm	Model	Communication Cost	Source
Broadcast	Chain	Hockney	$t = (P-1)(l + \frac{m}{b})$	[81]
		logGP	$t = (P-1) \cdot (L + 2 \cdot o + (m-1) \cdot G)$	[81]
	Binary	Hockney	$t = 2 \cdot (\lceil log_2(P+1) \rceil - 1) \cdot (l + \frac{m}{b})$	[81]
		logGP	$(\lceil log_2(P+1) \rceil - 1) \cdot (L + g + 2 \cdot (o + (m-1) \cdot G))$	[30, 73]
	Binomial	Hockney	$t = \lceil log_2(P) \rceil \cdot (l + \frac{m}{b})$	[82, 83]
		logGP	$t = \lceil log_2(P) \rceil \cdot (L + 2 \cdot o + (m-1) \cdot G)$	[30, 73]
Allgather	Ring	Hockney	$t = (P-1)(l + \frac{m}{b})$	[82, 83]
		logGP	$t = (P-1) \cdot (L + 2 \cdot o + (m-1) \cdot G)$	[81]

Table 4.1: Communication costs of various collective algorithms using Hockney's and the LogGP model. (Continued)

	Neighbor exchange	Hockney	$t = l + \frac{m}{b} + (\frac{p}{2} - 1) \cdot (l + 2 \cdot \frac{m}{b})$	[84]
		logGP	$t = \frac{p}{2} \cdot (L + 2 \cdot o + (2 \cdot m - 1) \cdot G) - m \cdot G$	[81]
	Recursive Doubling	Hockney	$t = log_2(P) \cdot l + (P - 1) \cdot \frac{m}{b}$	[82, 83]
		logGP	$t = log_2(P) \cdot (o + max\{g, L + o\} - G) + (P - 1) \cdot m \cdot G$	[81]
Alltoall	Linear	Hockney	$t = (P - 1) \cdot (l + \frac{m}{b})$	[82]
		logGP	$t = L + 2 \cdot o + (m - 1) \cdot G + 2 \cdot (P - 1) \cdot g$	[73]
	Pairwise	Hockney	$t = (P - 1) \cdot (l + \frac{m}{b})$	[82]
		logGP	$t = (P - 1) \cdot (L + o + (m - 1) \cdot G + g)$	[81]

Table 4.1: Communication costs of various collective algorithms using Hockney's and the LogGP model. (Continued)

	Bruck	Hockney	$t = \lceil log_2(P) \rceil \cdot l + \lfloor log_2(P) \rfloor \cdot \frac{m \cdot P}{2 \cdot b} + P \cdot m + (P - 2^{\lfloor log_2(P) \rfloor}) \cdot \frac{m}{b}$	[82, 83]
		logGP	$t = log_2(P) \cdot (o + (\frac{P \cdot m}{2} - 1) \cdot G + \frac{P \cdot m}{2} + max\{g, L + o\} + P \cdot m)$	[81]

4.1.1 Estimating the improvement of collective operations

The following section discusses the approach to derive an estimate for the improvement in the execution time of a collective communication operation given the improvement of the execution time of a point-to-point as a result of the tuning step. The ubiquitous formula for computing a relative improvement in a data point from an untuned value to a tuned value is

$$i = \frac{t_{untuned}(m) - t_{tuned}(m)}{t_{untuned}(m)}, \tag{4.1}$$

where i is the performance improvement, $t_{tuned}(m)$ and $t_{untuned}(m)$ is the execution time of communication operation of a message of length m, using an optimized parameter set respectively default settings.

Using Hockney's communication model

The fundamental assumption that the improvement observed when tuning a point-to-point operation only affects *one* parameter of the model at any given point in time. For Hockney's model it is assumed that the improvement i^{p2p} either comes from improvement of the latency, l, or the bandwidth, b, but not both at the same time. The binary tree broadcast operation is used to demonstrate how to derive the formula for the expected improvement, i^{coll}. According to Hockney's model, the execution time of a binary tree broadcast operation is

$$t_{untuned} = 2 \cdot (\lceil log_2(P+1) \rceil - 1) \cdot (l + \frac{m}{b}) \tag{4.2}$$

with the latency l and bandwidth b being the original values before the tuning. Let us denote the new, tuned latency and bandwidth as l^* and b^*, respectively. Therefore, the tuned execution time is

$$t_{tuned} = 2 \cdot (\lceil log_2(P+1) \rceil - 1) \cdot (l^* + \frac{m}{b^*}) \tag{4.3}$$

Substituting (4.3) and (4.2) in (4.1)

$$i^{coll} = \frac{l + \frac{m}{b} - l^* - \frac{m}{b^*}}{l + \frac{m}{b}}$$

In the first case, it is assumed that the bandwidth remains unchanged, and the improvement has been reflected entirely in the latency,

$$l^* = x \cdot l \text{ where } x<1$$

$$b^* = b$$

Hence,

$$i^{coll}_{Hockney(l)} = \frac{l - x \cdot l}{l + \frac{m}{b}} \tag{4.4}$$

Equation (4.4) therefore shows the expected improvement of the binary tree broadcast operation given an improvement of the latency by a factor of x. Note, that x is not the observed improvement of the point-to-point operation i^{p2p}. Section 4.1.2 shows how derive l^* (and therefore x) given i^{p2p}.

Similarly, assuming that the improvement only comes from the bandwidth parameter, the latency remains unchanged:

$$b^* = y \cdot b \text{ where } y>1$$

$$l^* = l$$

Hence, performance improvement in this case is

$$i^{coll}_{Hockney(b)} = \frac{m}{l \cdot b + m} \cdot (1 - \frac{1}{y}) \tag{4.5}$$

Using LogGP communication model

Deriving the theoretical improvements of a collective operation using the LogGP model follows the same pattern as demonstrated in the previous subsection for Hockney's mode. Starting point is the formula for the binary algorithm of broadcast operation:

$$t_{untuned} = (\lceil log_2(P+1) \rceil - 1) \cdot (L + g + 2(o + (m-1)G)). \tag{4.6}$$

The LogGP model has four parameters that could be affected by the tuning. We focus on the overhead o and gap per byte G parameters for two reasons: first,

deriving formulas for the other two parameters (L and g) follows the same pattern as demonstrated for o and G; second, the Latency L and the gap g are hardware parameters which are less likely to be influenced by Open MPI parameters. Let us denote the new latency and bandwidth as o^* and G^*. Therefore, the tuned execution time is

$$t_{tuned} = (\lceil log_2(P+1) \rceil - 1) \cdot (L + g + 2(o^* + (m-1)G^*)) \tag{4.7}$$

Substituting (4.6) and (4.7) in (4.1) leads to

$$i^{coll} = \frac{2(o + (m-1) \cdot G) - 2(o^* + (m-1)G^*)}{L + g + 2(o + (m-1)G)}.$$

In the first case, it is assumed that the gap per byte parameter, G, remains unchanged, and the improvement has been reflected entirely in the overhead, o, which leads to a performance improvement of

$$i^{coll}_{LogGP(o)} = \frac{2o(1-x)}{L + g + 2o + 2G(m-1)}. \tag{4.8}$$

In the second case, it is assumed that the overhead remains unchanged, and the improvement has been reflected entirely in the gap per byte, leading to

$$i^{coll}_{LogGP(G)} = \frac{2G(m-1)(1-x)}{L + g + 2o + 2G(m-1)} \tag{4.9}$$

4.1.2 Deriving the tuned parameter values

In this section equations to determine the improved network parameter l^*, b^* and o^*, G^* are derived. Using the equation (4.1), we can conclude that

$$t_{untuned}(m) = (1-i) \times t_{tuned}(m) \tag{4.10}$$

Using Hockney's communication model

To determine the tuned values for latency l^* and bandwidth b^*, the execution time of a point-to-point operation is modeled, as used by NetPipe. According to Hockney's model, the execution time of a point-to-point communication of message length m is

$$t(m) = l + \frac{m}{b} \tag{4.11}$$

Using equations (4.10) and (4.11) leads to

$$l^* + \frac{m}{b^*} = (1 - i^{p2p}) \cdot \left(l + \frac{m}{b}\right). \tag{4.12}$$

Using the same assumption as above, namely that only one communication parameter is affected by the tuning step, we can derive the formulas for l^* and b^*, respectively. In the first case, we assume that the bandwidth remains unchanged, and all the improvement has been reflected in the latency, resulting in

$$\begin{aligned} l^* + \frac{m}{b} =& (1 - i^{p2p}) \cdot \left(l + \frac{m}{b}\right) \\ l^* =& l - i^{p2p} \cdot \left(l + \frac{m}{b}\right) \end{aligned} \tag{4.13}$$

Similarly, assuming the latency remains unchanged and the improvement has been reflected entirely in the bandwidth, leads to

$$b^* = \frac{m}{\frac{m}{b} - i^{p2p} \cdot (l + \frac{m}{b})} \tag{4.14}$$

Using the LogGP communication model

Similarly, in the LogGP model the execution time of a point to point operation can be estimated by

$$t(m) = L + 2o + (m - 1) \cdot G \tag{4.15}$$

From equations (4.15) and (4.10) we can derive that

$$L + 2o^* + (m-1)G^* = (1 - i^{p2p})(L + 2o + (m-1)G) \tag{4.16}$$

Assuming that overhead remains unchanged, i.e. $o^* = o$, leads to

$$G^* = G - \frac{i^{p2p} \cdot (L + 2o + (m-1)G)}{m-1} \tag{4.17}$$

and assuming in the second case that $G^* = G$

$$o^* = o - \frac{i^{p2p}}{2}(L + 2o + (m-1) \cdot G) \tag{4.18}$$

4.1.3 Performance improvement of collective operations in terms of point-to-point improvement

Using the formulas derived in section 4.1.2, one can directly determine the improvement factor x used in section 4.1.1 as the ratio of the tuned vs. untuned parameter value. Alternatively, one could use the tuned parameter values from section 4.1.2 to calculate the execution time of a collective operation using the formulas presented in Table 4.1, and determine the expected improvement of the collective operation by simply applying the base formula shown in equation (4.1).

In this subsection, we would derive the expected improvement of a collective operation as a function of the improvement in the point-to-point operation, since this provides high-level information on the expected performance improvement.

Using Hockney's communication model

The performance benefit of a binary tree broadcast operation for the condition $b^* = b$ is given in Hockney's model by (4.4). Substituting (4.13) in (4.4) gives

$$i^{coll} = \frac{l - (l - i^{p2p}(l + \frac{m}{b}))}{l + \frac{m}{b}}$$

$$i^{coll}_{Hockney(l)} = i^{p2p} \tag{4.19}$$

Equation (4.19) states that a binary tree broadcast operation should experience the same performance improvement as the improvement observed by tuning a point-to-point operation of the same message length, assuming the performance improvement comes from optimizations to the latency parameter. Slightly rephrased, if the tuning step reduced execution time of a point-to-point of length m by e.g., 10%, equation (4.19) predicts that the performance of a binary tree broadcast operation should also improve by 10% for the same message length, assuming that the performance improvement can be attributed to the latency in Hockney's model.

Assuming that the latency in Hockney's model is unaffected by the tuning, i.e. $l^* = l$, the expected improvement for a binary tree broadcast operations is shown in (4.5). Using the formula for the tuned bandwidth value as shown in eq. (4.14) and substituting it into (4.5) leads to the same equation as (4.19). Thus, according to Hockney's model, the execution time of a binary tree broadcast operation should improve by the same factor as a point-to-point operation of the same message length improved as a result of a tuning process, independent of whether the improvement came from the latency or the bandwidth parameter.

Using LogGP communication model

Using the same approach as outlined above for Hockney's model, the expected performance improvement of a binary tree broadcast operation can be derived from substituting (4.18) in (4.8) assuming that $G^* = G$. This leads to

$$i^{coll}_{LogGP(o)} = i^{p2p}\frac{L + 2o + (m-1)G}{L + g + 2(o + (m-1)G)} \tag{4.20}$$

and for the second scenario, assuming $o^* = o$, by substituting (4.17) in (4.9)

$$i^{coll}_{LogGP(G)} = i^{p2p}\frac{2(L + 2o + (m-1)G)}{L + g + 2(o + (m-1)G)} \tag{4.21}$$

In contrary to Hockney's mode, the LogGP model makes much more nuanced prediction. Assuming that the performance improvement of a point-to-point operation stems from improving the overhead o, the expected performance benefit of a binary tree broadcast operation will be **lower** than the measured improvement of the point-to-point operation, since the denominator in (4.20) is always larger than the numerator. On the other hand, if the improvement comes from the parameter, G, as assumed in eq. (4.21), a collective operation could see a larger, equal or lower improvement than the imrpovement measured by the point-to-point operation, depending on whether $L+2o$ is larger, equal or less than g. Based on LogGP parameter values that we observed using the Netgauge tool [85] on various platform, all three possibilities can occur in real life.

Table 4.2: Improvement for each collective operation and algorithm for both communication models.

Collective	Algorithm	Model	Condition	Performance improvement	
Broadcast	Chain	Hockney	$b^* = b$	$i^{coll} = i^{p2p}$	$=$
			$l^* = l$	$i^{coll} = i^{p2p}$	$=$
		LogGP	$G^* = G$	$i^{coll} = i^{p2p}$	$=$
			$o^* = o$	$i^{coll} = i^{p2p}$	$=$
	Binary	Hockney	$b^* = b$	$i^{coll} = i^{p2p}$	$=$
			$l^* = l$	$i^{coll} = i^{p2p}$	$=$
		LogGP	$G^* = G$	$i^{coll} = i^{p2p} \frac{L+2o+(m-1)G}{L+g+2(o+(m-1)G)}$	$<$
			$o^* = o$	$i^{coll} = i^{p2p} \frac{2(L+2o+(m-1)G)}{L+g+2(o+(m-1)G)}$	
	Binomial	Hockney	$b^* = b$	$i^{coll} = i^{p2p}$	$=$
			$l^* = l$	$i^{coll} = i^{p2p}$	$=$
		LogGP	$G^* = G$	$i^{coll} = i^{p2p}$	$=$
			$o^* = o$	$i^{coll} = i^{p2p}$	$=$

Table 4.2: Improvement for each collective operation and algorithm for both communication models. (Continued)

Allgather	Recursive Doubling	Hockney	$b^* = b$	$i^{coll} = i^{p2p} \frac{log_2(P)(l+\frac{m}{b})}{log_2(P)l+(P-1)\frac{m}{b}}$	$<$
			$l^* = l$	$i^{coll} = i^{p2p} \frac{(P-1)(l+\frac{m}{b})}{log_2(P)l+(P-1)\frac{m}{b}}$	
		LogGP	$G^* = G$	$i^{coll} = i^{p2p} \frac{log_2(P)(L+2o+(m-1)G)}{log_2(P)(L+2o-G)+(P-1)mG}$	
			$o^* = o$	$i^{coll} = i^{p2p} \cdot \frac{(P-1)m-log_2(P)}{log_2(P)(L+2o-G)+(P-1)mG} \cdot \frac{L+2o+(m-1)G}{m-1}$	
	Ring	Hockney	$b^* = b$	$i^{coll} = i^{p2p}$	$=$
			$l^* = l$	$i^{coll} = i^{p2p}$	$=$
		LogGP	$G^* = G$	$i^{coll} = i^{p2p}$	$=$
			$o^* = o$	$i^{coll} = i^{p2p}$	$=$
	Neighbor Exchange	Hockney	$b^* = b$	$i^{coll} = i^{p2p} \frac{P(l+\frac{m}{b})}{2(l+\frac{m}{b}+(\frac{P}{2}-1)(l+\frac{2m}{b}))}$	$<$
			$l^* = l$	$i^{coll} = i^{p2p} \frac{(P-1)(l+\frac{m}{b})}{l+\frac{m}{b}+(\frac{P}{2}-1)(l+\frac{2m}{b})}$	$<$
		LogGP	$G^* = G$	$i^{coll} = i^{p2p} \frac{P(L+2o+(m-1)G)}{P(L+2o+(2m-1)G)-2mG}$	
			$o^* = o$	$i^{coll} = i^{p2p} \cdot \frac{(mP-\frac{P}{2}-m)(L+2o+(m-1)G)}{(\frac{P}{2}(L+2o+(2m-1)G)-mG)(m-1)}$	

Table 4.2: Improvement for each collective operation and algorithm for both communication models. (Continued)

Alltoall	Linear	Hockney	$b^* = b$	$i^{coll} = i^{p2p}$	$=$
			$l^* = l$	$i^{coll} = i^{p2p}$	$=$
		LogGP	$G^* = G$	$i^{coll} = i^{p2p}\frac{L+2o+(m-1)G}{L+2o+(m-1)G+2(P-1)g}$	$<$
			$o^* = o$	$i^{coll} = i^{p2p}\frac{L+2o+(m-1)G}{L+2o+(m-1)G+2(P-1)g}$	$<$
	Pairwise	Hockney	$b^* = b$	$i^{coll} = i^{p2p}$	$=$
			$l^* = l$	$i^{coll} = i^{p2p}$	$=$
		LogGP	$G^* = G$	$i^{coll} = i^{p2p}\frac{(L+2o+(m-1)G)}{2(L+o+(m-1)G+g)}$	$<$
			$o^* = o$	$i^{coll} = i^{p2p}\frac{L+o+(m-1)G}{L+o+(m-1)G+g}$	$<$
	Bruck	Hockney	$b^* = b$	$i^{coll} = i^{p2p}\lceil log_2(P)\rceil(l + \frac{m}{b})/\lceil log_2(P)\rceil \cdot l + \lfloor log_2(P)\rfloor \cdot \frac{m\cdot P}{2\cdot b} + P\cdot m + (P - 2^{\lfloor log_2(P)\rfloor})\cdot\frac{m}{b}$	
			$l^* = l$	$i^{coll} = i^{p2p}(l + \frac{m}{b})(\frac{P}{2}\lfloor log_2(P)\rfloor + (P - 2^{\lfloor log_2(P)\rfloor}))/\lceil log_2(P)\rceil\cdot l + \lfloor log_2(P)\rfloor \cdot \frac{m\cdot P}{2\cdot b} + P\cdot m + (P - 2^{\lfloor log_2(P)\rfloor})\cdot\frac{m}{b}$	

Table 4.2: Improvement for each collective operation and algorithm for both communication models. (Continued)

		LogGP	$G^* = G$	$i^{coll} = i^{p2p} \cdot \frac{L+2o+(m-1)G}{2 \cdot o+(\frac{P \cdot m}{2}-1) \cdot G+\frac{P \cdot m}{2}+L+P \cdot m}$	
			$o^* = o$	$i^{coll} = \frac{i^{p2p}}{(m-1)} \cdot \frac{(\frac{mP}{2}-1)(L+2o+(m-1)G)}{2 \cdot o+(\frac{P \cdot m}{2}-1) \cdot G+\frac{P \cdot m}{2}+L+P \cdot m}$	

Table 4.2 lists the expected improvement for all operations and algorithms. The last column indicates whether the expected improvement of the collective operation is equal (=), larger (>) or lower (<) than that of the point-to-point operation. For some formulas, the column is left blank, which indicates that all three options are possible. For allgather recursive doubling and neighbor exchange we have derived for condition L+o>g, similarly it can be derived for L+o<g. For a few algorithms, namely chain and binomial tree broadcast, and ring allgather, both models agree that the expected improvement in the performance of the collective operation should be equal to the improvement of the point-to-point operation. For the operations analyzed Hockney's model rarely distinguished between scenarios where the latency vs. the bandwidth is affected.

4.1.4 Evaluation of the performance models

The goal of this section is to evaluate how well the performance improvement models derived in previous section match actually observed data. Tests in this section have been executed on the crill cluster at the University of Houston using Open MPI 1.8.3. All tests have been executed at least three times, and the average values are presented subsequently. Tests have been executed for 32 and 64 processes for message length of 128 bytes, 1 KB, 12 KB, 16 KB, 32 KB, and 64 KB. The tuning step involved the tuning of seven parameters of the `openib` btl component, the parameter names and values are shown in Table 3.1. In our tests we enforced that all communication operations are going through the InfiniBand network by disabling shared memory `btl` components.

The evaluation consists of the following steps:

1. Using OTPO and the NetPipe benchmark, tune the parameters listed in Table 3.1 for each message length individually. The result of this step is one or multiple sets of parameters per message length $p_{set}(m)$ which lead to minimal execution time as reported by NetPipe,

2. Calculate the improvement $i^{p2p}(m)$ for each message length individually by comparing the execution time reported by NetPipe with default parameter values vs. the tuned parameter set $p_{set}(m)$. The improvement for each individual message length is shown in Figure 3.1,

3. Measure the execution time for each collective operation and algorithm using the SkaMPI benchmark for each message length m using the default parameter

values as well as the top 5 parameter sets $p_{set}(m)$ determined in step 1. For the subsequent analysis, we choose the best result obtained with any of the top 5 parameter sets,

4. Calculate the expected improvement $i^{coll}(m)$ for each message length using the formulas shown in Table 4.2.

The parameters of the untuned model were determined using the NetPipe benchmark for Hockney's model, using the 0-byte data transfer costs for the message latency, and the asymptotic maximum bandwidth observed for the bandwidth. Those values were determined to be $l = 1.6\ \mu s$ and $b = 1941.5\ MB/s$ for the crill cluster. For the LogGP model, we used the NetGauge toolkit [85] version 2.1. The values used are shown in Table 4.3.

4.1.5 Results of tuning collective operations

Table 4.3: LogGP parameters used for the evaluation

Parameter	Value
L	1.84 μs
o	1.49 μs
g	1.08 μs for $m \leq 32$ KB
	11.9 μs for $m \geq 32$ KB
G	0.00067 μs

Figures 4.1 to 4.12 present the results obtained for the all-gather, broadcast and all-to-all operations. Each graph contains 6 lines, namely:

- *Netpipe*: the improvement observed for point-to-point operations for the corresponding message length using the NetPipe benchmark,
- <*algorithm name*>: the measured improvement of a given algorithm (e.g., binary broadcast, chain broadcast),
- *Hockney(l)* and *Hockney(b)*: the predicted performance improvement of a collective operation using Hockney's model assuming that the improvement can be attributed to the latency only / bandwidth only,
- *LogGP(G)* and *LogGP(o)*: the predicted performance improvement of a collective operation using LogGP model assuming that the improvement can be attributed to the gap per byte parameter or the overhead parameter respectively.

Note that on some graphs (allgather ring, broadcast binomial, and broadcast chain), all models are identical and equal to $i^{p2p}(m)$. Thus, only two lines are visible, namely the measured and the predicted performance.

The main result shown in these graphs is that for the all-gather and the broadcast operations, there is a good correlation between the observed and the predicted improvement of the collective operation, i.e. the trend of the observed performance improvement matches at least one, but typically multiple of the predicted lines. The observed improvement of the all-gather neighbor exchange and all-gather recursive doubling algorithm favor the models that assume the latency related parameters are

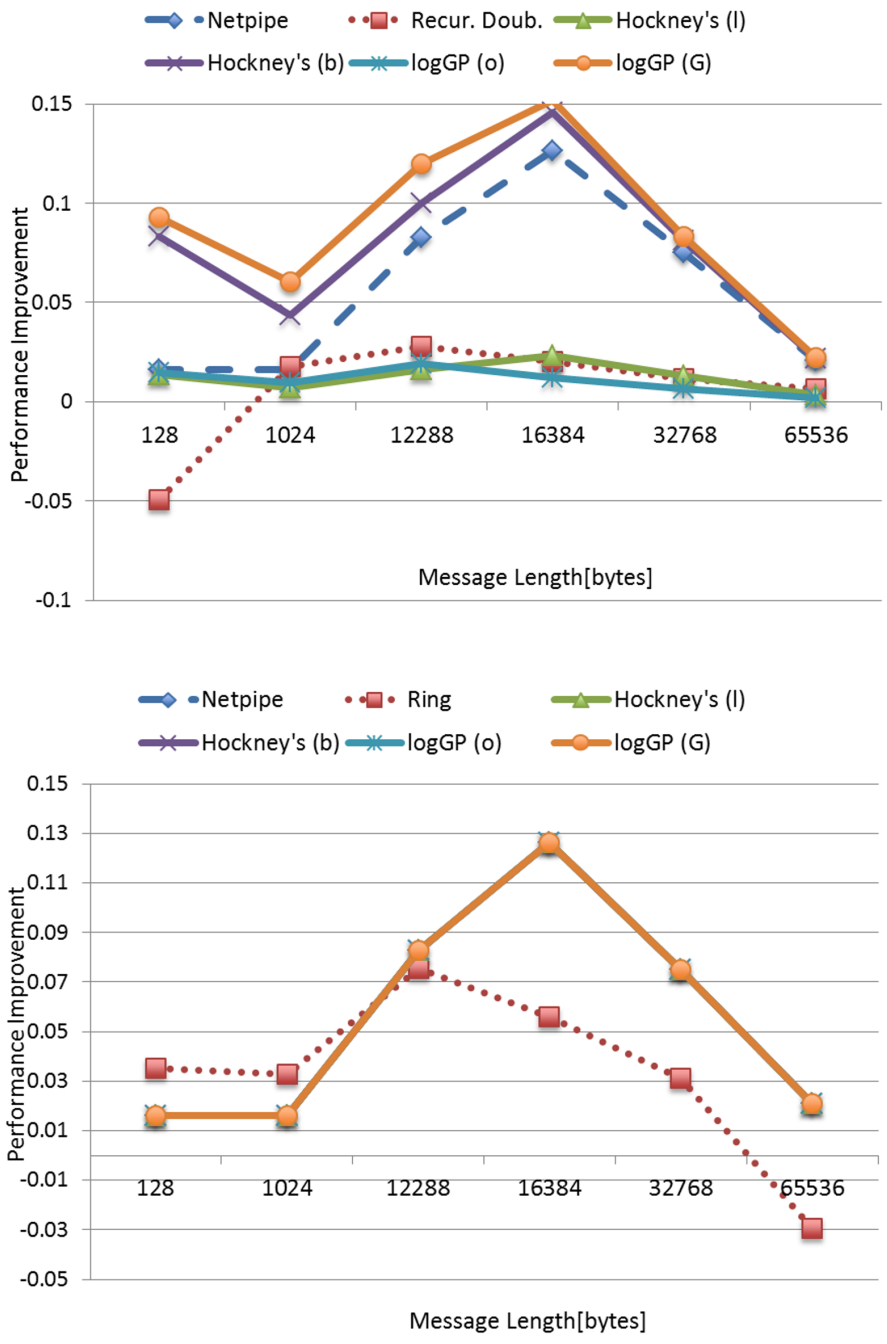

Figure 4.1: Expected and measured performance improvement for 32 processes of recursive doubling (top), and ring (bottom) algorithm for an Allgather operation.

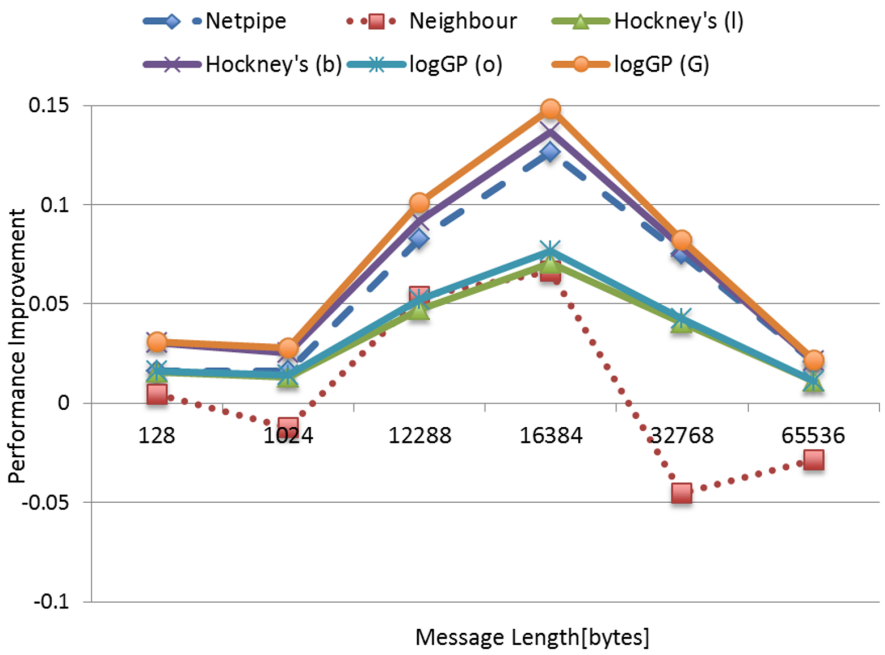

Figure 4.2: Expected and measured performance improvement for 32 processes of neighbor exchange algorithm for an Allgather operation.

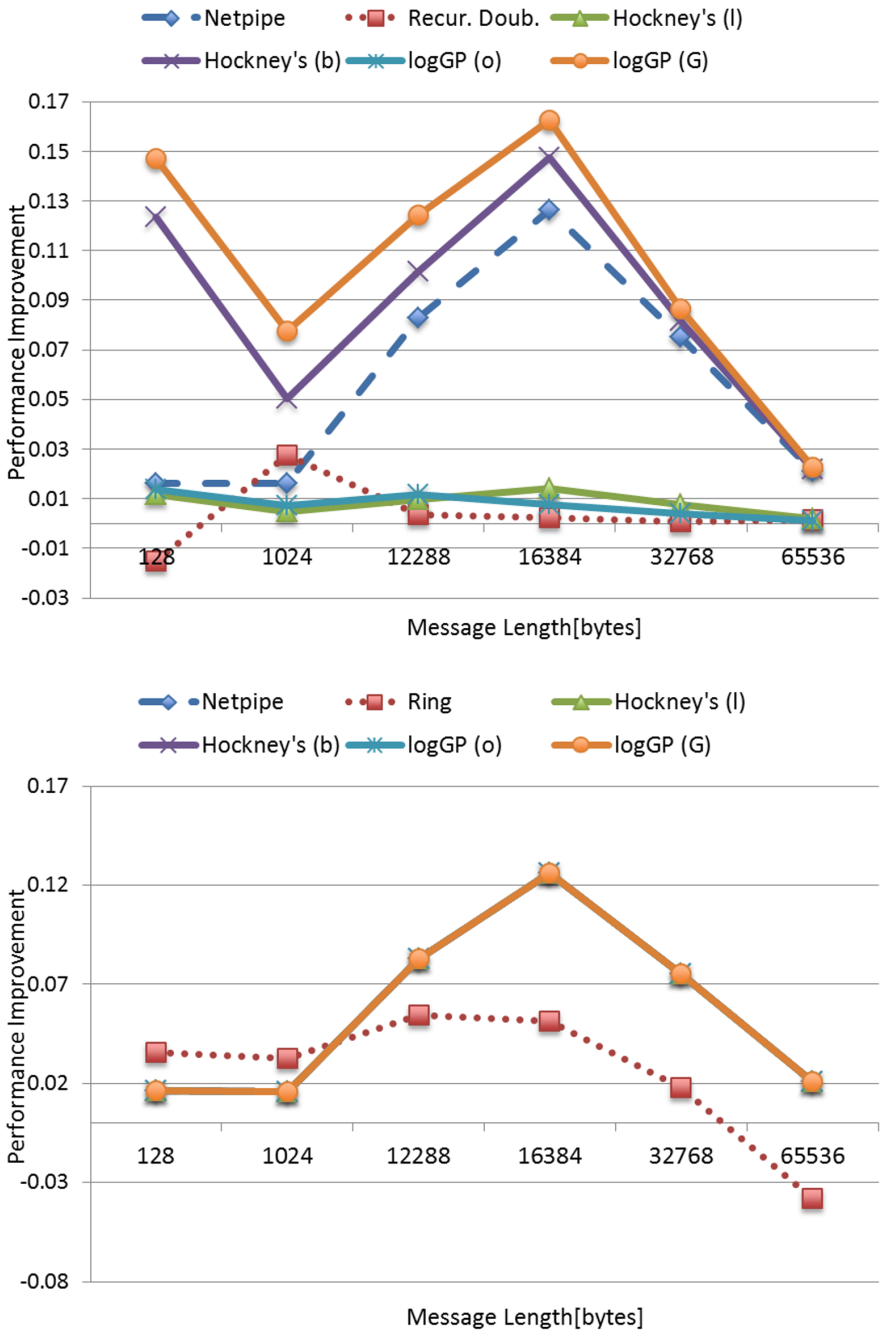

Figure 4.3: Expected and measured performance improvement for 64 processes of recursive doubling (top), and ring (bottom) algorithm for an Allgather operation.

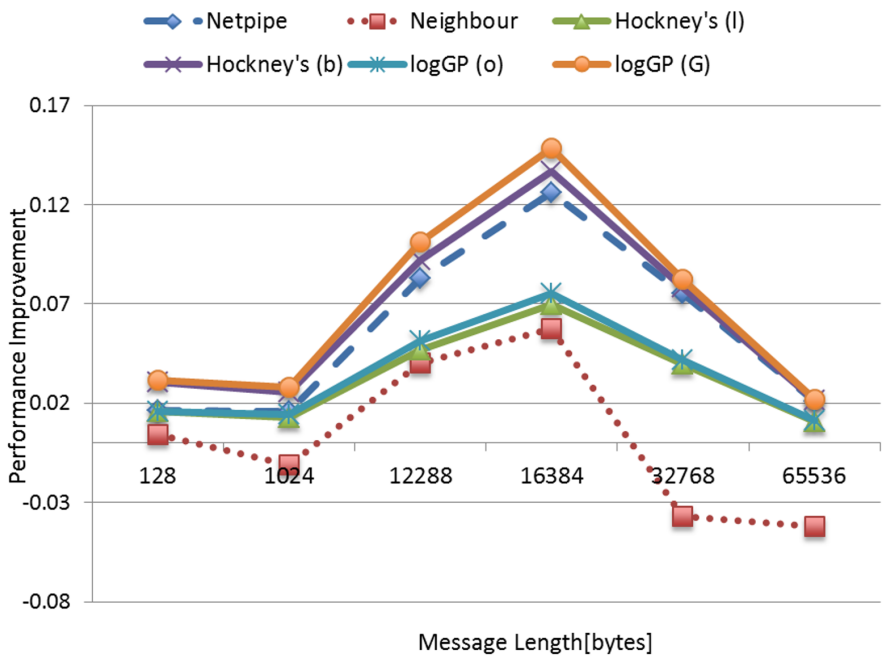

Figure 4.4: Expected and measured performance improvement for 64 processes of neighbor exchange algorithm for an Allgather operation.

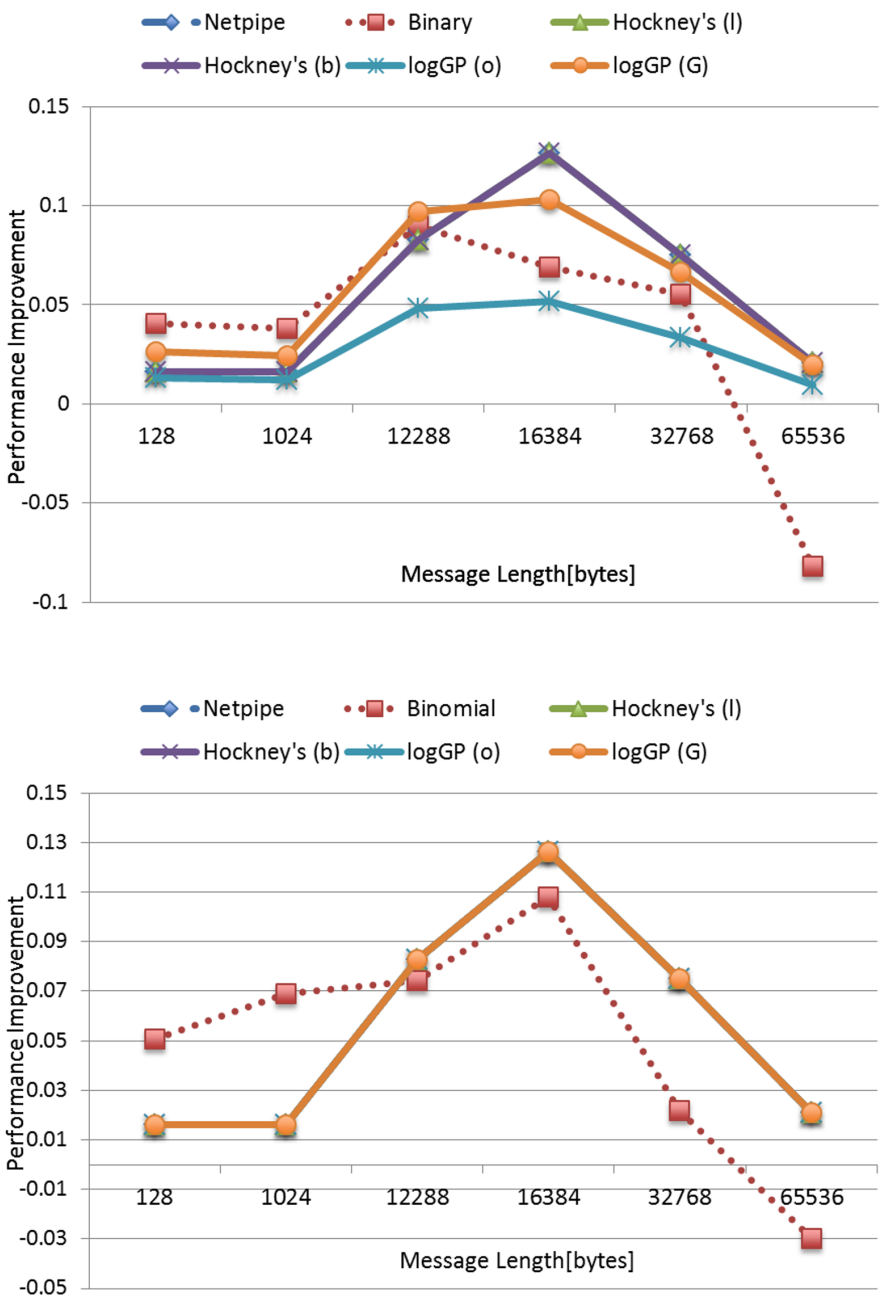

Figure 4.5: Expected and measured performance improvement for 32 processes of binary tree (top), and binomial tree (bottom) algorithm for a broadcast operation.

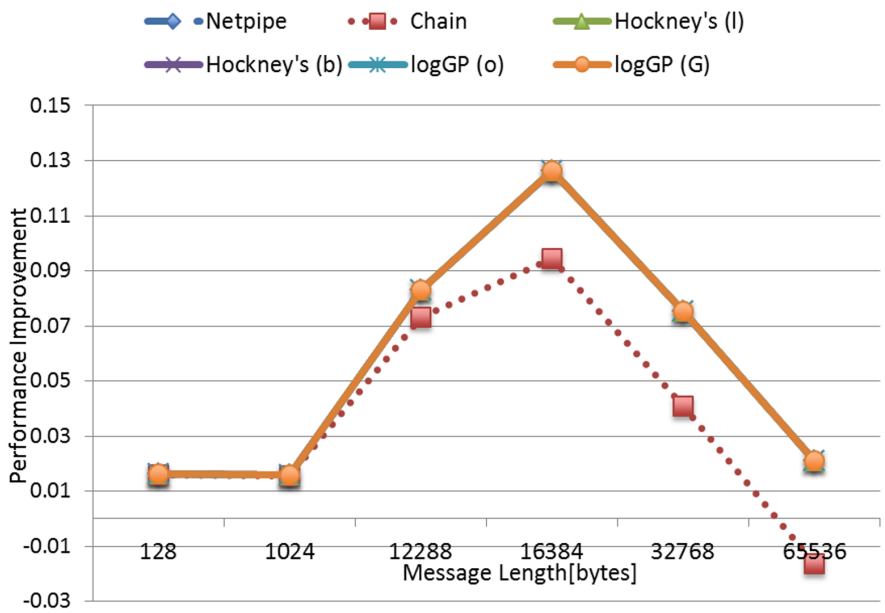

Figure 4.6: Expected and measured performance improvement for 32 processes of chain algorithm for a broadcast operation.

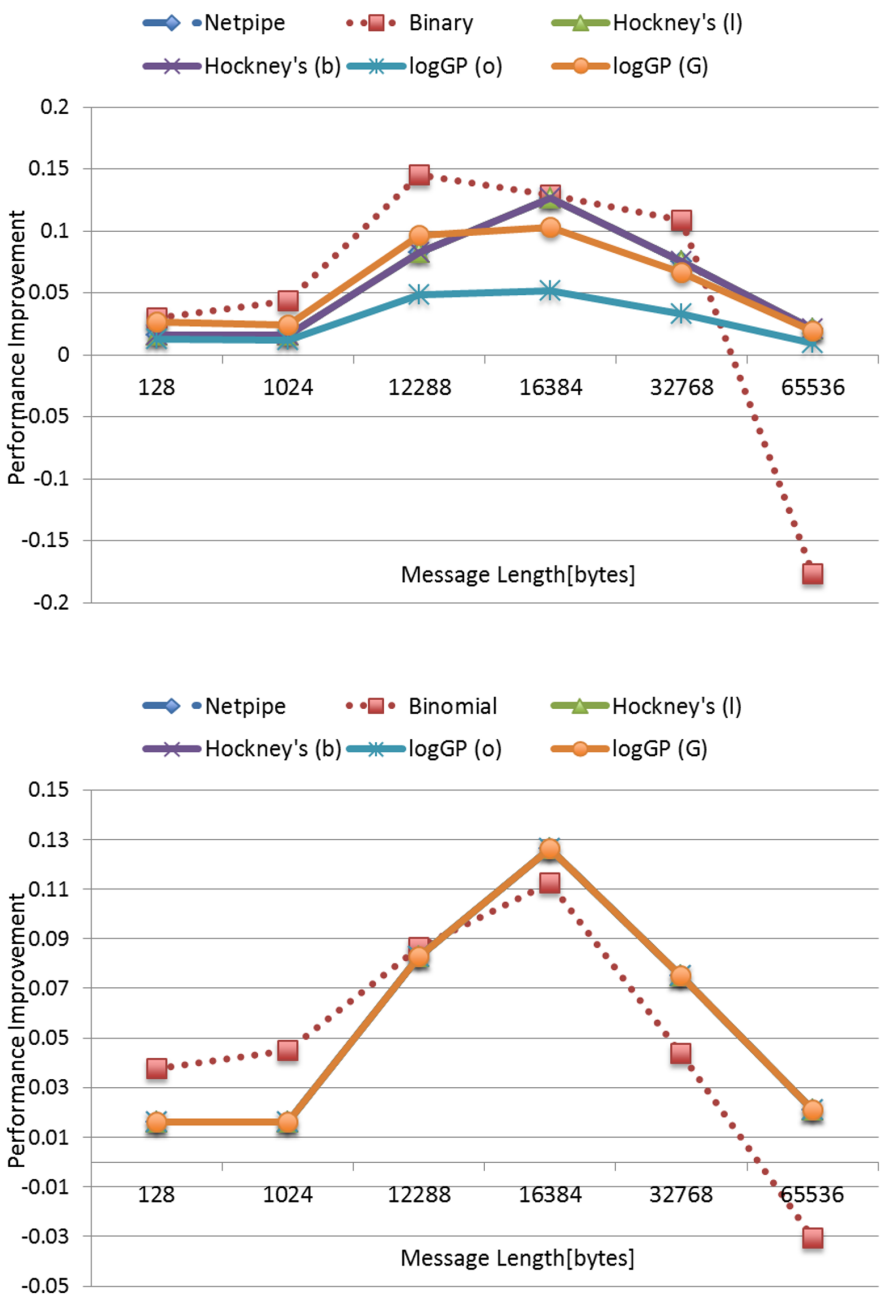

Figure 4.7: Expected and measured performance improvement for 64 processes of binary tree (top), and binomial tree (bottom) algorithm for a broadcast operation.

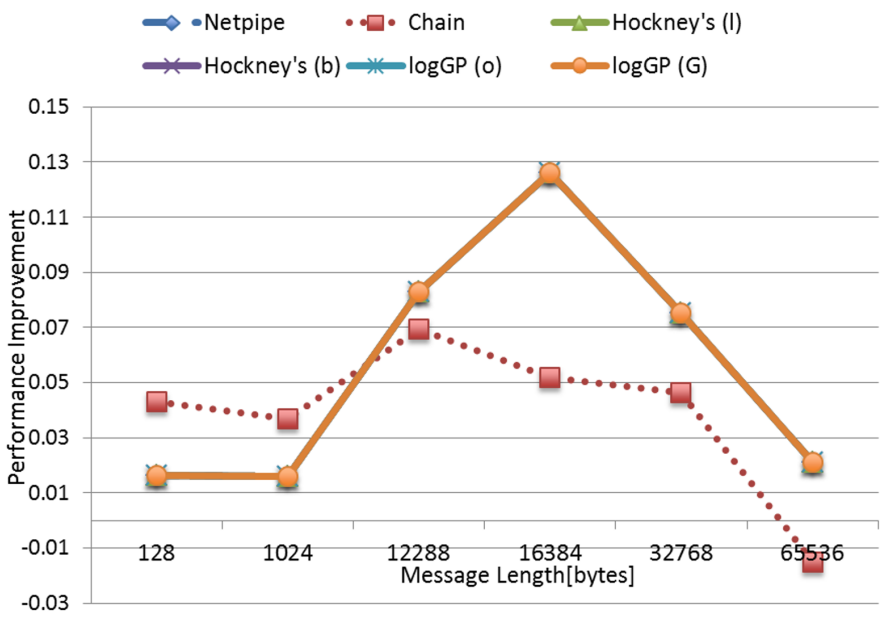

Figure 4.8: Expected and measured performance improvement for 64 processes of chain algorithm for a broadcast operation.

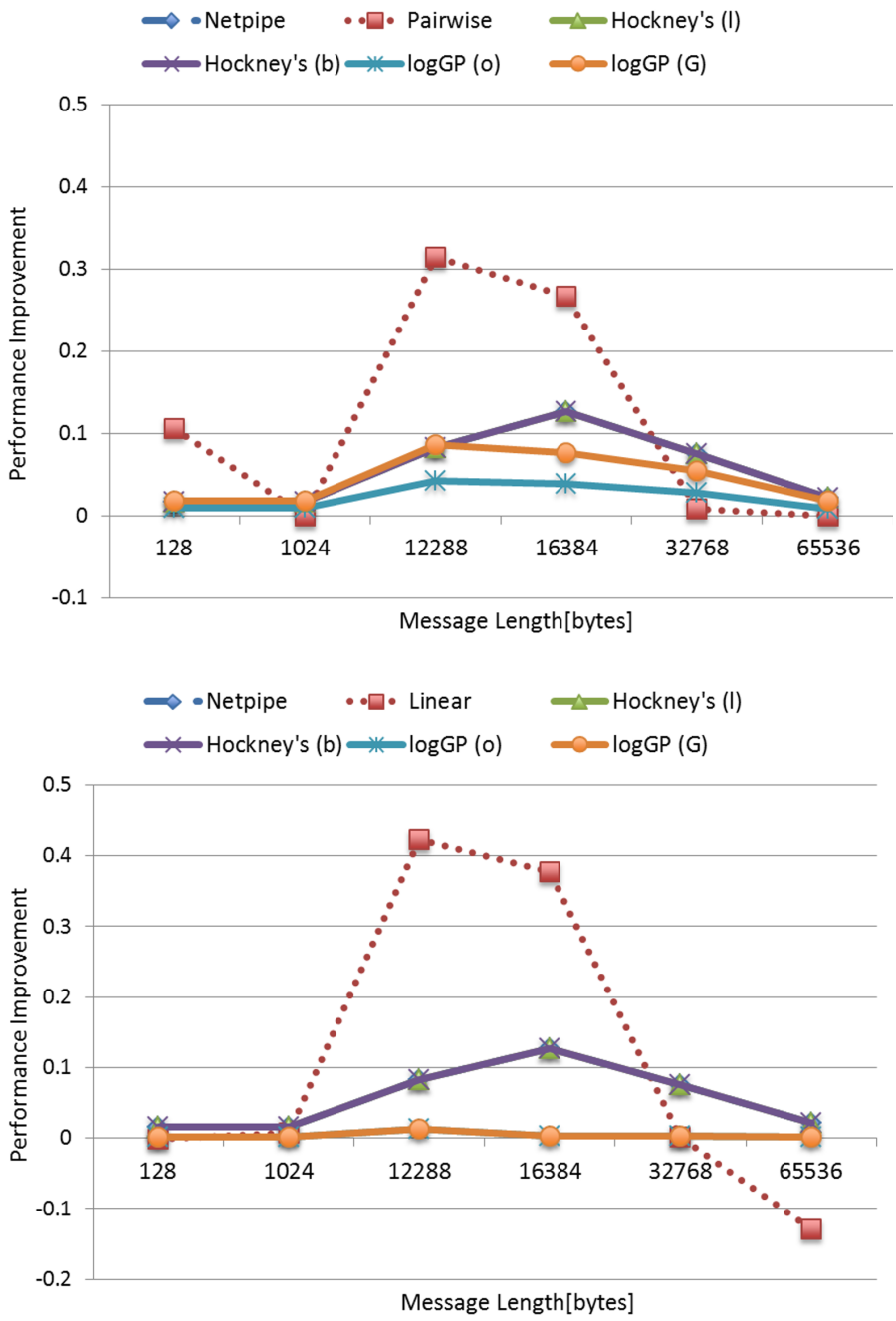

Figure 4.9: Expected and measured performance improvement for 32 processes of pairwise exchange (top), and linear (bottom) algorithm for Alltoall operations.

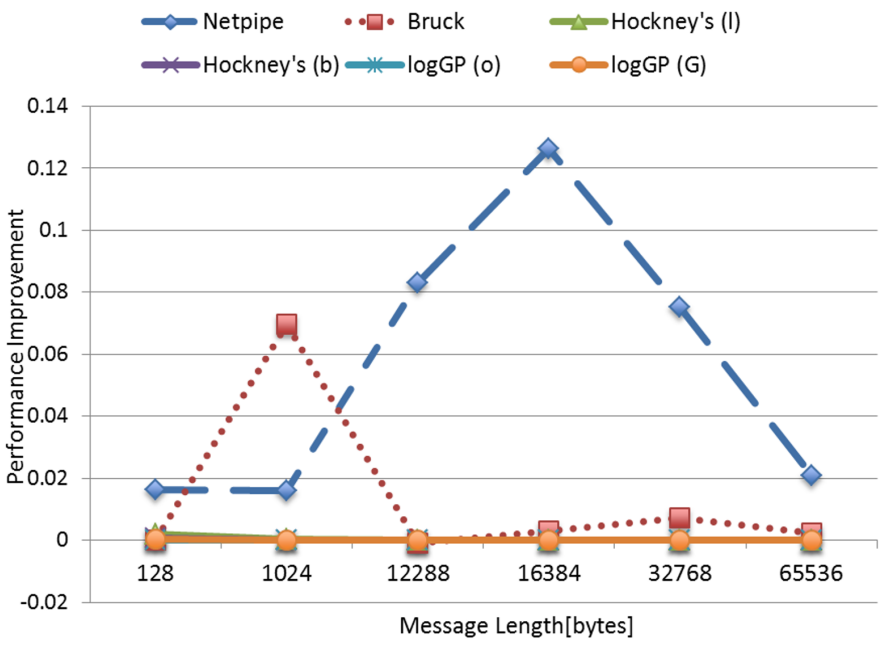

Figure 4.10: Expected and measured performance improvement for 32 processes of brucks algorithm for Alltoall operations.

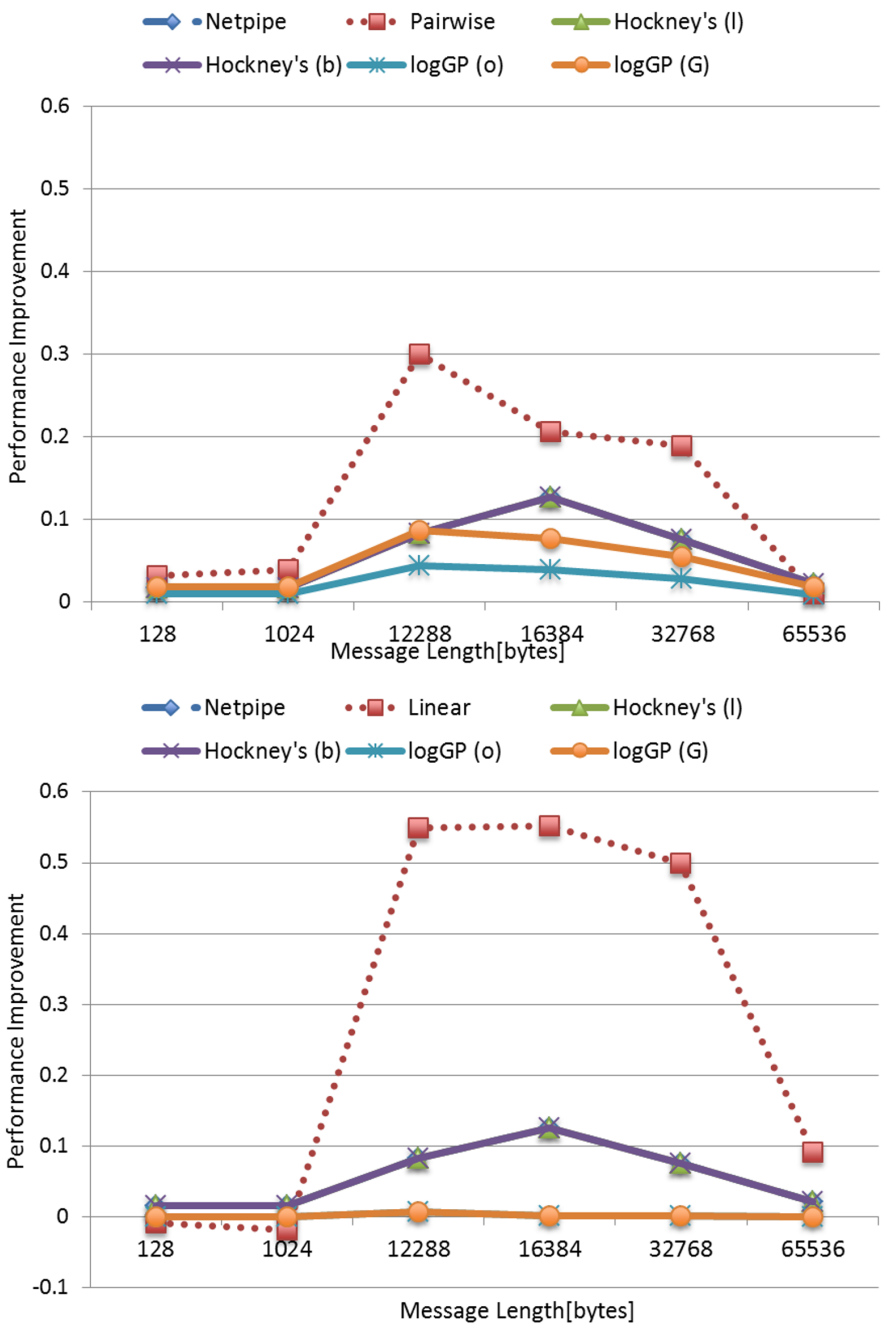

Figure 4.11: Expected and measured performance improvement for 64 processes of pairwise exchange (top), and linear (bottom) algorithm for Alltoall operations.

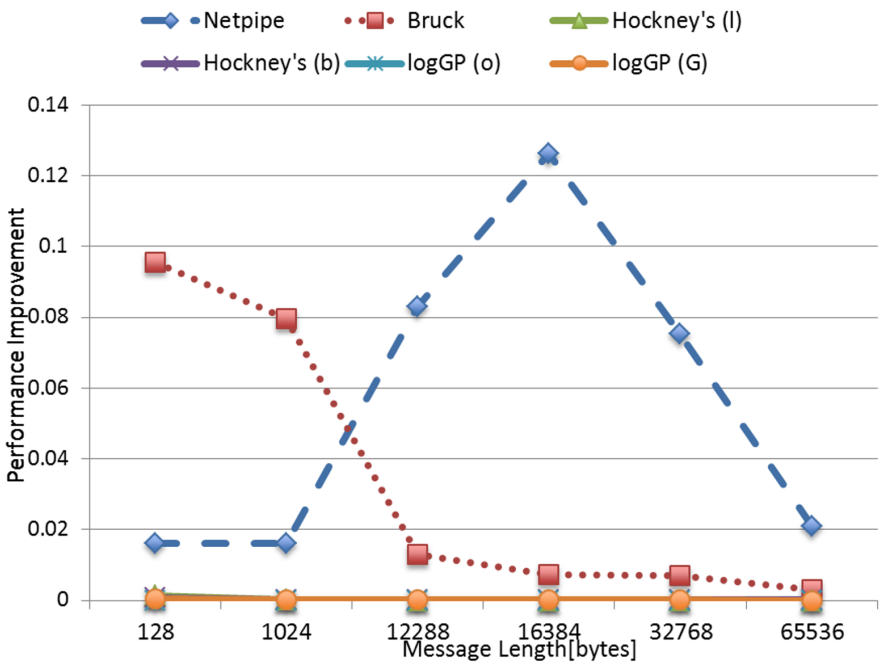

Figure 4.12: Expected and measured performance improvement for 64 processes of brucks algorithm for Alltoall operations.

affected by the tuning, i.e. they follow the predictions made by the Hockney(l) and LogGP(o) models.

The all-to-all results reemphasize the value and the limitations of performance models. While performance models are useful to understand fundamental properties of the algorithms and thus guide the expectation of end-users, they do represent simplifications compared to the real-world. Some data points for the linear and pairwise all-to-all algorithm represent perfect examples for that, since the observed performance benefit is much larger than predicted by any of the models. One has to keep in mind the number of assumptions, simplifications, and potential inaccuracies made in any of the models and the measurements, including:

- Neither Hockney's model nor LogGP handle network congestion or protocol switch from eager to rendezvous protocol (unlike LogGPS),
- In the derived models, we do not distinguishing between sender and receiver side overhead and use for the sake of simplicity the same, untuned LogGP parameter values for both intra - vs. inter node communication.

Considering all of these aspects, the results represent a reasonable good match between the models and observed data with occasional outliers, very similar to many other performance analysis papers using these models. Out of the nine algorithms used in this study, eight had a very small Mean-Squared Error between the most optimal prediction model and the observed improvement (between .0002 to .011) for the 64 process test cases, indicating a good overall match. In the following, we discuss two interesting sub-topics that also contribute towards limitations of the modeling aspect.

4.1.6 Impact of number of processes on optimal parameter values

When tuning parameters of Open MPI using a point-to-point benchmark, one makes fundamentally the assumption that the optimal value for a parameter does not change with the number of processes used. This assumption is not always correct. Consider for example an analysis performed with one of the parameters listed in Table 3.1, namely `btl_openib_eager_limit`. This parameter defines the threshold value starting from which Open MPI will use a rendezvous protocol for the data transfer over an InfiniBand network instead of the eager protocol. Since the range of values for this parameter was set to be between 4 KB and 48 KB, one would expect no impact of this parameter for 64 KB message length, since the message is expected to be transferred in rendezvous mode independent of the eager value used. This is however not necessarily what one observes.

Table 4.4: Impact of eager limit on cost of communication operations (μs) with message length 64 KB.

	no. of procs.	Eager limit 12 KB	Eager limit 48 KB	Relative improv.
NetPipe	2	68.5 μs	68.8 μs	-0.4%
Binary tree bcast	32	941.7 μs	1023.1 μs	-8.6%
Binary tree bcast	64	1106.8 μs	1314.9 μs	-18.8%

Table 4.4 shows the execution time obtained for an eager limit of 12 KB (default value) and 48 KB for a message length of 64 KB using the NetPipe benchmark and for binary tree broadcast operation using SkaMPI for the same message length. As

expected, changing the eager limit to 48 KB did not have an impact on the NetPipe benchmark, which operates one message at a time. In a collective operation however, increasing the eager limit lead to significant performance degradation. It is beyond the scope of this work to detail the reasons for this behavior (hint: the eager limit also has an impact on the memory registration aspect of the `openib` component), the main message is, a parameter value obtained by tuning a ping-pong benchmark might have a very different impact for a multi-process communication pattern. This behavior was identified as the main reason why so many of our measurements have a negative improvement for the 64 KB message length scenario.

4.1.7 Discrepancy between expected and actually used message lengths

Analyzing the lack of performance in Bruck's algorithm revealed another reason why predicting the performance of a collective operation based on point-to-point operations can be challenging. As an example, consider that the NetPipe results shown in Figure 3.1 suggest that the biggest improvement in the performance of point-to-point operation can be obtained for 16 KB message on this platform. While the linear and pairwise exchange all-to-all algorithm do in fact follow this prediction, Bruck's algorithm shows virtually no benefit for this message length. This can be attributed to the fact, that Bruck's algorithm does not send any message of lengths 16 KB between processes in this scenario, but creates larger messages of 256 KB for which Figure 3.1 indicate limited performance improvements. Similarly, a peak for 1 KB message length in the 32 process test cases stems from the fact that the

algorithm uses internally 16 KB messages length, which showed the largest potential for performance improvements.

Chapter 5

Performance Models for Communication in Collective I/O Operations

Many large scale scientific applications operate on large data files and spend a significant amount of time reading and writing input or output files. Collective I/O operations are group I/O operations which allow for optimizations across a group of processes. They reorder the data across process boundaries to match the layout of the data on the file system level which often reduces the time spent in I/O operations significantly. Internally, collective I/O operations are based on a sequence of communication operations, namely the shuffle step and accessing the file system through read and write operations. In most scenarios only a subset of the application processes, often referred to as aggregators, actually touch the file, i.e. perform read

or write operations. For very large amounts of data, collective read and write operations are executed internally in multiple iterations, which allows to limit the amount of temporary memory required to hold data of other processes on the aggregators.

The performance of collective I/O operations depends on numerous factors, including the data decomposition strategy used by the application and file system level characteristics, such as a locking protocol and locking granularity used by the parallel file system. Parameters of collective I/O operations such as the number of aggregators used, or the chunk-size that a collective I/O operation internally operates on, have an enormous influence on the performance of the operation itself [79, 86]. As of today, however, the internal parameters used by libraries providing collective I/O operations are determined using simplistic heuristics or extensive testing of a particular application scenario on a given platform. In this chapter, we develop performance models for collective I/O operations and provide a more systematic approach to determine optimal parameter values for these operations.

As a first step, the communication occurring in collective write operations is modeled. The shuffle step is in the most generic sense an $n \rightarrow m$ communication operation that could be described using a generic all-to-all operation. Yet, this simple model would not take the actual file domain partitioning strategy – i.e. which aggregator is responsible for which portions of the data file – used by the collective I/O algorithm into account, nor the data decomposition used by the parallel application. These two aspects determine however ultimately to which aggregator a particular data item has to be sent as part of the collective read/write operation, and thus determines the underlying communication pattern.

Numerous models for communication operations have been developed over the last two decades. Performance models have to balance simplicity vs. accuracy: simpler models are easier to understand and utilize, but have more severe restrictions in terms of accuracy of their predictions. The performance models developed in the following sections are based on the LogGP [87] model, which represents a reasonable compromise between accuracy and usability.

In this chapter, communication occurring in collective write operations for the two most widely used file domain partitioning strategies in MPI I/O libraries today are modeled. These two file domain partitioning strategy are even [88] and static [89] partitioning. Our models further distinguish between one- and two-dimensional data decomposition used by the parallel applications. Further in the chapter, properties of performance models are discussed, and potential impact on the performance of collective I/O operations using LogGP parameters derived on multiple platforms is demonstrated.

5.1 Concept

Collective I/O operations represent higher level Application Programming Interfaces (APIs) which allow to reorder data across processes to match the layout of the data on the file system level. The most widely used algorithm is the two-phase I/O [88] algorithm, which – as the name implies – consists of two steps. For a write operation, the first phase (often referred to as the shuffle step) redistributes data among the processes to match the layout of the data on the file, while the second phase executes the actual write operation. In addition, the two-phase I/O algorithm

introduces two further optimizations. First, only a subset of the MPI application processes actually touch the file, i.e. perform read or write operations. The processes executing file I/O operations are also referred to as the *aggregators*. Second, for very large collective read and write operations, the two-phase I/O algorithm is split into multiple cycles internally. This keeps memory requirements on the aggregator processes within reasonable limits, and allows for potential overlap of the shuffle step and the write operation of subsequent cycles.

Depending on the internal operations of the file system, the approach taken internally for redistributing data across the processes can vary dramatically. Most notably, the locking protocol used by the file system (client side vs. server side locking), file ranges assigned to a lock as well as the locking granularity have a major impact on how many aggregator processes should be used, and which portion of the data each aggregator should write. In the following we describe the two most popular approaches, namely the even partitioning used in the original two-phase I/O algorithm [88], and the static partitioning [89]. Note that other distributions such as subgrouping of processes as done in the dynamic segmentation algorithm [90] have shown benefits for some scenarios. We omitted this from our work for the sake of clarity.

In the *even* partitioning strategy, the aggregated file regions to be written/read are distributed uniformly as contiguous chunks among the I/O aggregators. The part of the file that one aggregator is responsible for is also called a file partition. Figure 5.1 shows this distribution as part of a collective write operation for four processes, two aggregators and a parallel file system consisting of a two I/O server.

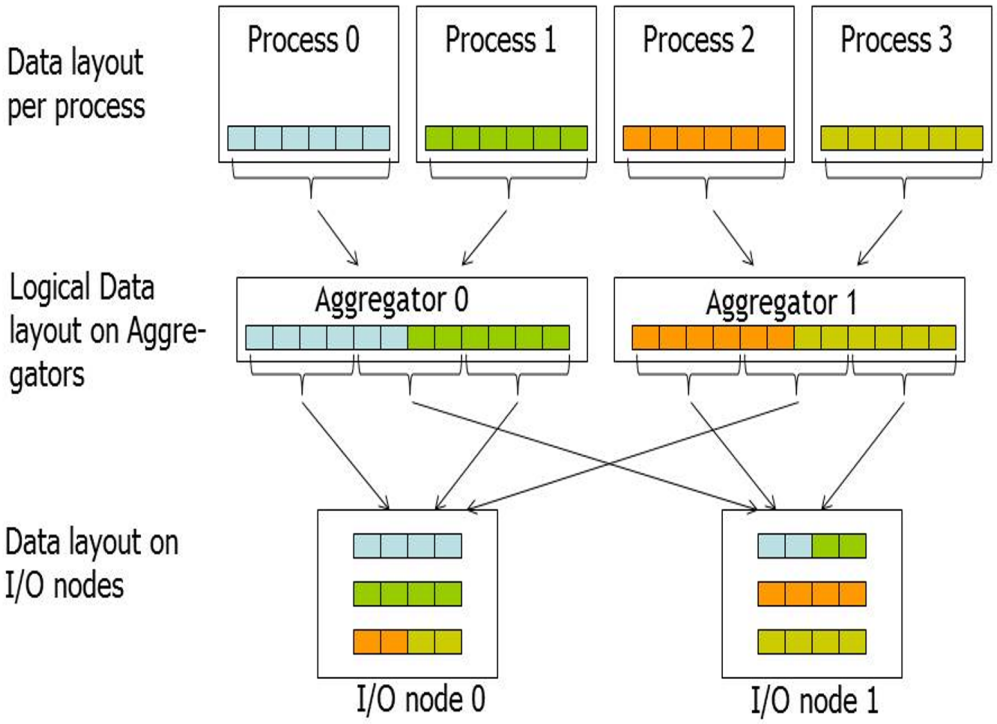

Figure 5.1: An example for the even data redistribution strategy.

Each process holds six elements that are contiguous in the file, leading to a total of 24 elements that have to be written. The even distribution strategy will then lead to each aggregator having to write 12 elements of data in this scenario.

In the *static* distribution, fixed-sized blocks are divided as per the locking granularity, and are distributed to I/O aggregators in round-robin fashion. Each aggregator communicates with same set of I/O servers. Very often, the number of I/O servers and number of aggregators are either identical, or have a common divisor. Figure 5.2 shows the same example as previous and the resulting internal distribution on the aggregator processes. Each aggregator is still responsible to write 12 elements. However, in contrary to the even distribution, the first aggregator only accesses data

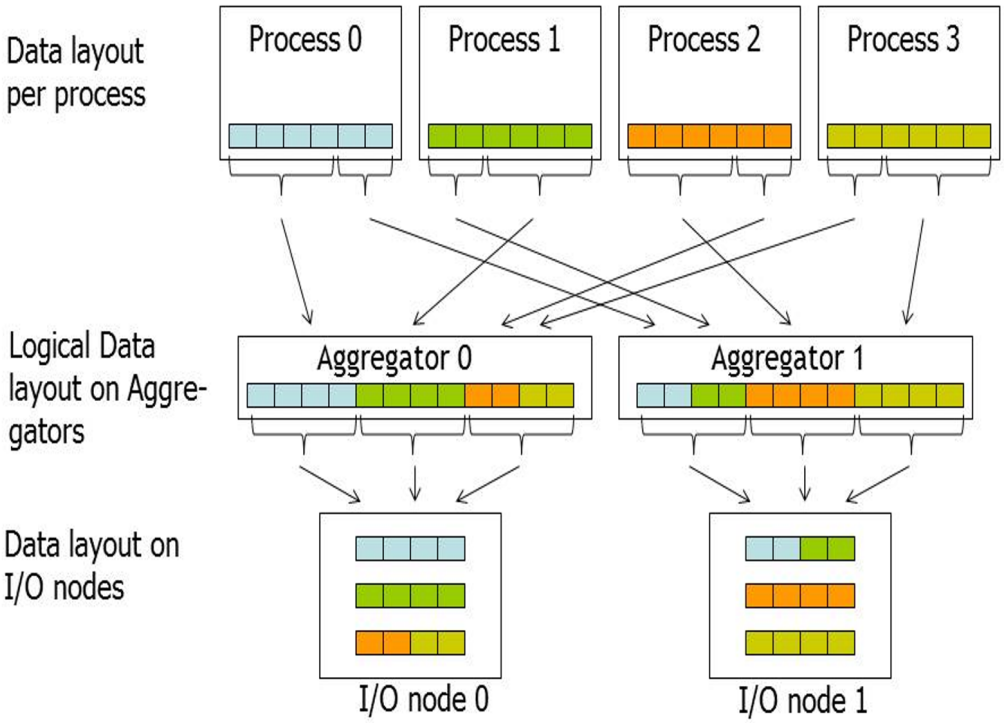

Figure 5.2: An example for the static data redistribution strategy.

blocks from the first I/O server, and the second aggregator only accesses data from the second I/O server.

5.1.1 Generic model

Within the context of this work, we used the LogGP [87], discussed in section 2.5. The generic model makes certain assumptions and simplifications. Given P processes and P_a aggregators (with $P_a \leq P$), we assume for the sake of simplicity that all process provide the same amount of data (d_p bytes) to the collective I/O operation. An aggregator process has an internal collective buffer size of b_c bytes. The generic model assumes that a process is involved in n_s cycles for send operations of m_s

bytes, and communication will have to occur to n_{as} aggregators in each cycle. Thus, according to the LogGP model, the costs for sending data by an MPI process is

$$T^{send} = n_s(L + 2o + (n_{as} - 1)g + (m_s - 1)n_{as}G). \tag{5.1}$$

An aggregator is further involved in n_r cycles requiring receive operations of m_s bytes from n_{ar} processes. Therefore,

$$T^{recv} = n_r(L + 2o + (n_{ar} - 1)g + (m_s - 1)n_{ar}G). \tag{5.2}$$

Since an aggregator process is also a regular MPI process and part of the collective I/O operation, it is involved in both send and receive operations. Thus, it is sufficient to model the time taken by one aggregator process.

$$T = T^{send} + T^{recv} \tag{5.3}$$

Using eq. 5.3 it comes down to determine the parameters n_s, n_{as}, n_r, n_{ar}, and m_s for different partitioning strategies used by the MPI I/O library and data distribution approaches used by the application. The task can be further simplified by considering that the five parameters listed above are not entirely independent of each other. First, each aggregator will have to write in both, the even and the static distribution strategy

$$b_a = \frac{P \cdot d_p}{P_a} \tag{5.4}$$

bytes of data. Since an aggregator is operating on fixed chunks of b_c bytes, the number of receive cycles can generally be expressed as

$$n_r = \lceil \frac{b_a}{b_c} \rceil = \lceil \frac{P \cdot d_p}{P_a \cdot b_c} \rceil. \tag{5.5}$$

Furthermore, eq. 5.1 implies that a process is involved in n_s cycles, in each cycle communicating with n_{as} aggregators by sending m_s bytes each. Assuming that a process writes the entire data of d_p bytes to disk, d_p has to be equal to $n_s \cdot n_{as} \cdot m_s$ or with respect to n_s

$$n_s = \lceil \frac{d_p}{n_{as} \cdot m_s} \rceil. \tag{5.6}$$

Eq. 5.5 and 5.6 help to reduce the actual number of parameter that need to be determined to three, namely n_{as}, n_{ar}, and m_s.

5.1.2 Even partitioning strategy

The two-phase I/O algorithm using the even partitioning strategy requires multiple communication operations for its execution. All-gather and all-reduce operations are used to determine the starting and ending offsets for all data items written in that collective I/O operation, and to determine the file partitions that each aggregator is responsible for. Furthermore, the data gathered in these operations are used to determine in which cycle a process has to provide/send data to which aggregator. We will omit these operations from our model for multiple reasons: first, performance models for various algorithms for all-gather and all-reduce operations are available in the literature [91]. We focus purely on the data exchange between processes and aggregators. Second, these operations remain constant independent of the number of aggregagors or other internal parameters of the two-phase I/O algorithm and are thus irrelevant for developing a model that allows analytical determination of the optimal values for these parameter (e.g., number of aggregators or collective buffer sizes). An underlying goal of this work.

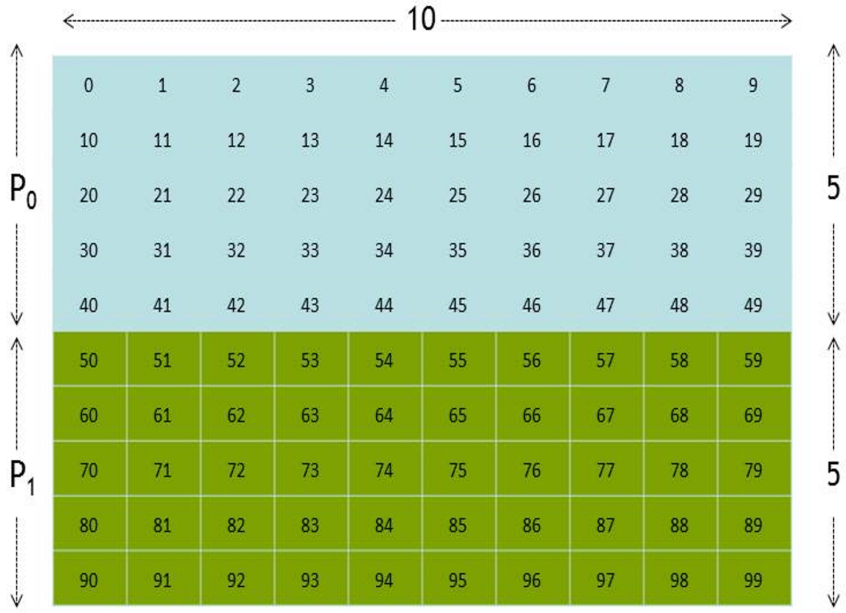

Figure 5.3: An example for a 1-D block-row wise data decomposition of a 2-D matrix.

We analyze two very common data decomposition approaches used by applications and their impact on the communication costs of collective I/O algorithms, namely a 1-D and a 2-D data decomposition.

1-D data decomposition

The first scenario assumes a one-dimensional data distribution of the data. Figure 5.3 shows an example of a 2-D matrix stored using row-major ordering of data (i.e. following the C/C++ convention), a block-row wise data distribution. The numbers in the matrix represent the position of that element in the flat file. As a result of this data distribution in the parallel application, a contiguous chunk of data is assigned to single process.

As discussed in section 5.1.1, the number of iterations required by an aggregator to gather the data assigned to it can be determined using eq. 5.5. We use some special cases to derive the formulas for other parameters, namely, n_{as}, n_{ar}, and m_s.

Case I: if $d_p > b_c$ If the amount of data that a process has to write is (significantly) larger than the collective buffer size, the number of messages in a cycle is predominantly $n_{as}^{1D} = n_{ar}^{1D} = 1$, can be however up to 2 for elements at the boundary of an aggregator partition. Considering that the default collective buffer size in current MPI I/O implementations is in the range of $16 \rightarrow 32$ MB, and the data per process d_p can easily reach hundreds of MB per process, we will consider for our formulas the dominant value, namely 1 message per cycle. The message length in that case equal to the collective buffer size $m_s = b_c$.

Case II: if $d_p \leq b_c$ If the amount of data written by a process is less than the collective buffer size, an aggregator will have to communicate with up to $n_{ar}^{1D} = \lceil \frac{b_c}{d_p} \rceil$ processes. The dominant message length in this scenario (extrapolating from the assumption that d_p is significantly lower than b_c) is $m_s = d_p$ bytes, i.e. an MPI process will send its data in a single message to the corresponding aggregator. Consequently, $n_{as}^{1D} = 1$.

2-D data decomposition

The two-dimensional data decomposition is a logical extension of the one-dimensional case discussed in the previous subsection. For the sake of simplicity, we use process

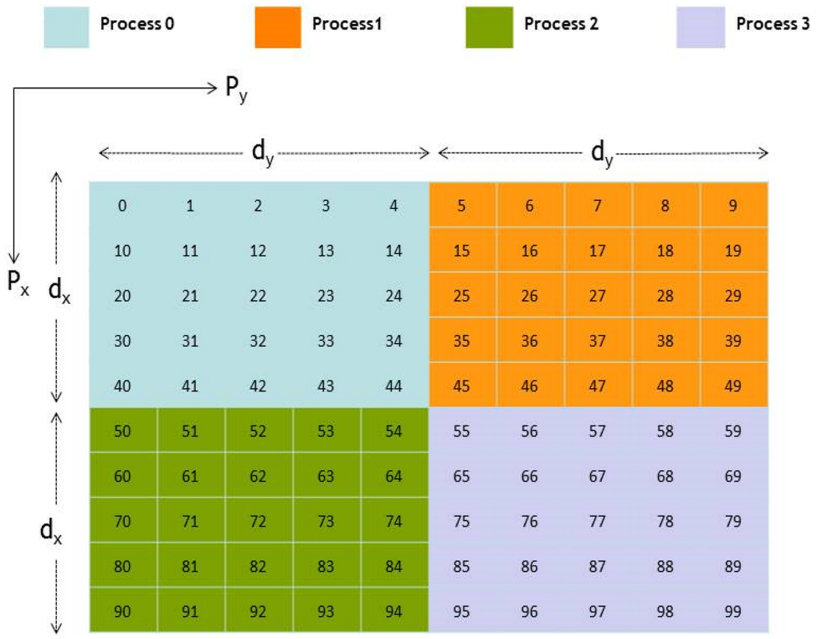

Figure 5.4: An example for a 2-D data decomposition of a 2-D matrix.

counts and matrix sizes that can easily be decomposed in two dimensions. Specifically, we assume that the number of processes can be decomposed in a 2-D cartesian topology, such that

$$P = P_x \cdot P_y \tag{5.7}$$

and the data size per process is

$$d_p = d_x \cdot d_y \tag{5.8}$$

An example is shown in Figure 5.4. In contrary to the 1-D scenario, each process holds multiple, discontigous segments of data in the flat file. The data assigned to a row of processes in the cartesian grid is logically before the data belonging to the processes in the next row of processes. To understand the communication pattern occurring for the even partitioning method in this scenario, several special cases are

analyzed first.

Case I: If $P_a = P_x$ If the number of aggregators is equal to the number of processes in the x-direction of the cartesian process topology, each row of processes has logically a separate aggregator process. Assuming that every process has the same amount of data, each aggregator communicates with $n_{ar} = P_y$ processes, and each process sends data to only one aggregator, i.e. $n_{as} = 1$. We derive the message length used for the communication in the next case.

Case Ia: If $P_a = c \cdot P_x$ An extension to the first case is given, if each row of processes has more than one aggregator assigned. Due to the even partitioning approach, c aggregators will split the partition created by the data of P_y processes. For example having two aggregators per row of processes would lead to the first aggregator having the first half of the domain (consisting of $d_y/2$ rows of data), and the second aggregator the second half. Still, both aggregators would have to communicate with all $n_{ar} = P_y$ processes. A process has to communicate with $n_{as} = c$ aggregators in this scenario.

The message length m_s is either $\frac{b_c}{P_y}$ in case the data of P_y processes is exceeding the collective buffer size b_c of c aggregators, or $\frac{d_p}{c}$ otherwise. It can be formulated the following way:

$$m_s = \begin{cases} \frac{b_c}{P_y} & if d_p \cdot P_y \geq c \cdot b_c \\ \frac{d_p}{c} & else \end{cases} \tag{5.9}$$

Since the else part of the above equation is not self-explanatory, we would like to demonstrate its correctness by calculating the overall amount of data received by an

aggregator. The product of $n_r \cdot n_{ar} \cdot m_s$ should lead to the entire amount of data being received by an aggregator. The first parameter is given by eq. 5.5, which leads to

$$\begin{aligned} n_r \cdot n_{ar} \cdot m_s &= \lceil \frac{P \cdot d_p}{P_a \cdot b_c} \rceil \cdot P_y \cdot \frac{d_p}{c} \\ &= \lceil \frac{P_x \cdot P_y \cdot d_p}{c \cdot P_x \cdot b_c} \rceil \cdot P_y \cdot \frac{d_p}{c} \\ &= \lceil \frac{P_y \cdot d_p}{c \cdot b_c} \rceil \cdot P_y \cdot \frac{d_p}{c} \end{aligned}$$

Since $d_p \cdot P_y < c \cdot b_c$ according to the condition in eq. 5.9, the first fraction will always be less than one, and the ceiling of that fraction will be 1. Thus, the overall amount of data received by an aggregator is $P_y \cdot \frac{d_p}{c}$, i.e. $\frac{1}{c}$ of the overall amount of data held by a row of processes in the 2-D process topology, which is the expected outcome.

Two additional comments on eq. 5.9. First, the if part eq. 5.9 covers the vast majority of real-life usage scenario. Second, there is theoretically a third potential condition that would have to be added to eq. 5.9, in case it takes a single row of the data matrix across less than P_y processes to fill up the collective buffer size b_c, i.e. $k \cdot d_y > b_c$, with $1 <= k <= P_y$. In this case, an aggregator might communicate with fewer than P_y processes per cycle, and the message length would be $m_s = d_y$. A quick estimate shows that assuming a collective buffer size b_c of 32 MB, the overall amount of data would have to exceed 1 PB (PetaByte) in size (i.e. a 2-D matrix larger than 32 MB $\times$ 32 MB), which is as of today still unrealistic. Because of this, we will ignore this scenario for the rest of the paper.

Case Ib: If $P_a > P_x$ This case is a generalization of the previous case, assuming that the number of aggregators is not an integer multiple of P_x. Using the terminology of the Case Ia, $P_a = c \cdot P_x$, with c not necessarily being an integer value in this scenario. In this case, the domain assigned to a particular aggregator might span more than one row of processes. The maximum number of processes an aggregator has to communicate with is however still $n_{ar} = P_y$, with the difference being that the P_y processes might be from different rows of the cartesian process topology. On the sender side, the number of aggregators that a process has to communicate with changes to $n_{as} = \lceil c \rceil$, with c being $\frac{P_x}{P_a}$. All other parameters are otherwise unchanged compared to Case Ia.

Case II: If $P_a = \frac{P_x}{c'}$ In this scenario, an aggregator would be responsible for the partition generated by c' rows of processes. Without going into the details for the sake of simplicity, one can show that the formulas derived in the previous scenarios Ia and Ib also applies for this configuration, by using $c = \frac{1}{c'}$.

To summarize the 2-D case, the parameters for this scenario are: $n_{ar}^{2D} = P_y$, $n_{as}^{2D} = \lceil \frac{P_x}{P_a} \rceil$, and the message lengths m_s is given by eq. 5.9.

5.1.3 Static partitioning strategy

As discussed before, the static partitioning algorithm distributes data to the aggregators such that an aggregator is only required to communicate to a single I/O server. This is beneficial for file systems such as Lustre, which utilizes a locking protocol that assigns a write lock to a process for all segments of a file hosted by that I/O server. If multiple aggregators would write data segments hosted by the same I/O server,

the current lock assigned to one process would have to be revoked and re-assigned to another process, which incurs significant costs.

1-D data decomposition

Recall that in this data decomposition, a contiguous chunk of data is assigned to single process, as shown in Figure 5.3. The first process can for example send the first, b_c, bytes to aggregator one, the second, b_c, bytes to aggregator two, and so on, all within the same cycle. Thus, this file partitioning strategy leads to $n_{as} = P_a$ in most realistic scenarios, and a message length $m_s = b_c$ bytes if $d_p > P_a \cdot b_c$, and $n_{as} = d_p/P_a$ otherwise. From an aggregator's perspective, each aggregator will typically only communicate with one process per cycle, i.e. $n_{ar} = 1$ (ignoring for the moment the scenario where $d_p < b_c$).

2-D data decomposition

Following the previous example, we therefore derive formulas for some special cases first, and then the more generic formulas based on the understanding of the special cases.

Case I: if $d_y \cdot P_y = b_c$ Each row of the global matrix represents data sufficient to fill exactly the collective buffer of an aggregator, and thus each row of the data matrix goes to a different aggregator. Each sender communicates per cycle to all P_a aggregators, i.e. $n_{as} = P_a$, unless the the number of data rows per process d_x is less than the number of aggregators P_a. In the latter case, $n_{as} = d_x$. The message length will be $m = d_y$ in both cases. From the aggregator's perspective, each aggregator

will have to receive data from $n_{ar} = P_y$ processes in each cycle.

Case Ia: if $d_y \cdot P_y = c \cdot b_c$ Each row of the global matrix fills c collective buffers at subsequent aggregator processes, assuming initially that c is an integer value. Imagine for example a scenario based in Figure 5.4 of $P_a = 4$ aggregators, and a collective buffer size of $b_c = 5$ elements. A process in this scenario will only communicate with subset of the aggregator processes, i.e. $n_{as} = \frac{P_a}{c}$. The message length will still be an entire row of data held by a matrix, i.e. $m_s = d_y$. On the receiver side, an aggregator will also only have to communicate with a subset of the processes due to the occurring regular pattern, i.e. $n_{ar} = \frac{P_y}{c}$.

Case Ib: if $d_y \cdot P_y = c \cdot b_c$ This case is a simple extension to case Ia, assuming that c does not have to be an integer value. Without going into the details, most assumptions made in Case Ia still hold for this scenario as well, the main difference being that one has to use the ceiling of the formulas derived for n_{as} and n_{ar}.

Case II: if $c' \cdot d_y \cdot P_y = b_c$ In this scenario c' rows of the global matrix are required to fill a buffer of size b_c. Thus, a process might have to combine the data of multiple rows into a single message to generate enough data per cycle for the aggregator, i.e. $m_s = c' \cdot d_y$. Each process will ultimately communicate with either all P_a aggregators or with as many aggregators as required to send the data of all of its d_x rows. Thus,

$$n_{as} = \begin{cases} P_a & if d_x \geq c' \cdot P_a \\ \frac{d_x}{c'} & else \end{cases} \tag{5.10}$$

On the receiving side, an aggregator has to receive $n_{ar} = P_y$ in each cycle. Note, that the in contrary to the even partitioning case presented in the previous subsection,

the formulas shown for Case Ib and II are not identical, i.e. simply calculating the inverse of c' and using it in Case Ib will lead to the incorrect result. However, for $c = c' = 1$, i.e. the borderline case for both scenario Ib and II, the parameters are identical in both cases.

5.2 Discussion

The goal of this section is to evaluate properties of the models developed and demonstrate the impact of the data distribution and partitioning strategy on the communication costs in a collective write operation. We focus on test cases using 144, 225, and 576 processes. These process counts were chosen to allow for an equal distribution for the 2-D data decomposition cases in both dimensions.

First, to demonstrate the impact of the different data distributions and file partitioning strategies on the parameters of our models derived in the previous section, we compare the values obtained for n_{as}, n_s, n_{ar}, n_r, and m_s for a particular scenario, namely a data size of 1 GB per process, a collective buffer size of $b_c = 32\ MB$ and $P_a = 64$ aggregators. The resulting parameters are shown in Table 5.1.

Table 5.1: Parameters obtained for the different data decomposition and file partitioning strategies.

Parameter	even 1-D			even 2-D			static 1-D			static 2-D		
	144	225	576	144	225	576	144	225	576	144	225	576
n_{as}	1	1	1	6	5	3	32	32	32	64	64	64
n_s	32	32	32	64	96	256	1	1	1	6	8	12
n_{ar}	1	1	1	12	15	24	1	1	1	12	15	24
n_r	72	113	288	72	113	288	72	113	288	72	113	288
m_s	32 MB	32 MB	32 MB	2.66 MB	2.13 MB	1.33 MB	32 MB	32 MB	32 MB	2.66 MB	2.13 MB	1.33 MB

The values shown in Table 5.1 reveal two interesting observations:

- The parameters involved in the sender side (n_{as}, and n_s) are different for all four scenarios. On the receiver side, however, there is difference in the values of n_{ar} and n_r for the 1-D vs. the 2-D cases, but the values for both parameters are identical for the two partitioning strategies, i.e. 1-D even and 1-D static lead to the same parameter values for n_r and n_{ar}, and similarly for 2-D even and 2-D static,

- The message sizes for the 2-D scenarios are shorter, requiring consequently a larger number of messages to transfer the same amount of data compared to the 1-D data distribution scenarios.

Lets analyze the parameter values obtained for 1-D even and the 1-D static scenarios in more details to highlight another properties of our models. The 1-D even model requires 32 cycles, in each cycle a single message (of 32 MB) is being sent by a process to a single aggregator. The 1-D static model on the other hand leads to just 1 cycle, within that cycle there will be however 32 messages (of 32 MB each) sent to 32 different aggregators. In our generic formula as shown in eq. 5.4, the first scenario will lead to a larger number of *network latencies* L (since each cycle only includes the cost of a single network latency), while the second scenario leads to a larger number of *gap per message* g parameters. Thus, the difference in the costs between these two models from the sender perspective will come down to the difference between the costs of a network latency L vs. the gap per message parameter g.

To understand the implications of the difference, we calculated the actual costs of the communication occurring in a collective write operations for LogGP parameters

Table 5.2: LogGP Parameters used

Parameter Name	Opuntia	crill-IB	crill-GE
L	6.04 μs	2.82 μs	32.96 μs
g	7.40 μs	7.4 μs	297 μs
o	25.88 μs	73.38 μs	139.7 μs
G	0.00351 μs	0.00067 μs	0.07991 μs

obtained from three different platforms/networks at the University of Houston: 56 Gbit Ethernet network on the opuntia cluster, the DDR InfiniBand network used by the Crill cluster (referred to as crill-IB), for the same cluster using the administrative Gigabit Ethernet network (referred to as crill-GE). The LogGP parameters were obtained using netgauge v.2.4.6 [92]. For the subsequent analysis, the LogGP values obtained for a message size of 128 KB were used [1]. Table 5.2 lists the LogGP parameters for all three platforms. The results shown in Figure 5.5 indicate that the 2-D data distributions has up to 5% higher communication costs compared to the 1-D data distribution scenario for the DDR InfiniBand network. The difference between the two data distributions however decreases with increase in message size. This can be explained with the fact that with increasing message length the bandwidth component of the formulas are dominating the overall execution time, and thus the number of messages per cycle and the number of cycles become less relevant. Results for the other two networks are similar both from quantitative and qualitative perspectives.

The predicted differences between static and even partitioning are relatively low.

[1] In theory one would have to use the LogGP parameter values obtained for each message length used. For the sake simplicity, we wanted to use a single value, which could be representative for the various message lengths deployed by the actual implementations used in the next section

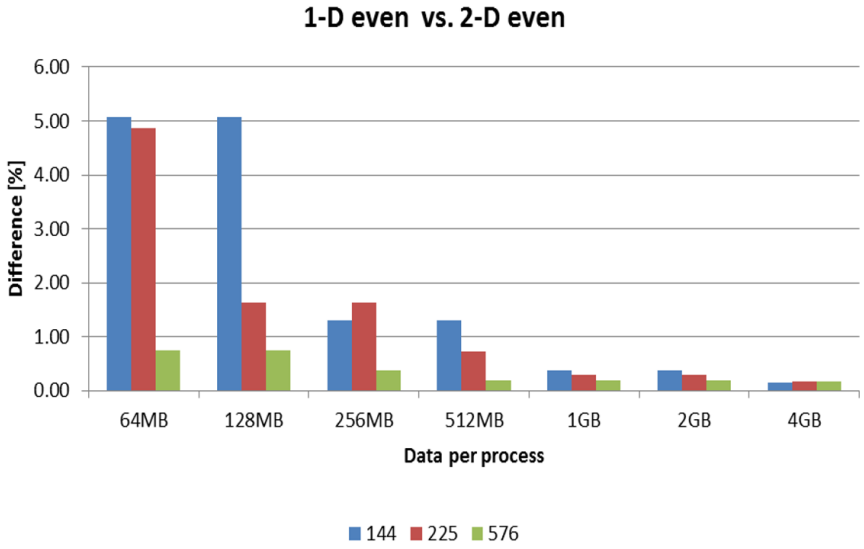

Figure 5.5: Crill DDR IB 1-D vs. 2-D predictions.

In addition to the observation in the previous scenario about the bandwidth becoming the dominant factor, there is one more factor contributing to this. With decreasing ratio of no. of processes vs. no. of aggregators (i.e. $\frac{P}{P_a}$), the receive operations become the dominant component in our performance models. This is highlighted in Figure 5.6, which shows percentage of time spent in the receive portion of the performance models for all four scenarios. As discussed previously (e.g. see Table 5.1), the receive part of the models are however identical between the static and the even partitioning strategy for the same data decomposition. Thus, the difference between static and even partitioning stemming purely from the sender side parameters.

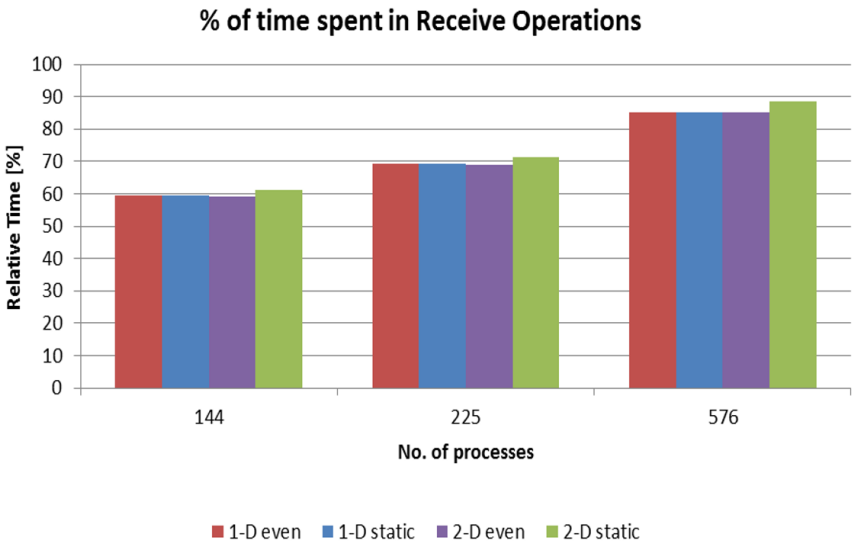

Figure 5.6: Crill-IB relative time spent in receive operations.

5.2.1 Influence of the collective buffer size

One of the advantages of having performance models is the ability to quickly evaluate the impact of various parameters on the overall performance of the algorithm. For example, in Figure 5.7 the impact of various values of the collective buffer size b_c on the communication costs of collective write operations is being evaluated. Specifically, Figure 5.7 shows the predicted communication costs of a collective write operation performed by 576 processes using 100 aggregator processes, in which every process writes 1 GB of data as part of the operation. The LogGP parameters used in this section are based on the Crill-IB platform. The results of this analysis indicate, that there are significant performance improvements when increasing the collective buffer size from lower values up to ∼16 MB, but the performance of the communication operations does not further benefit from increasing b_c beyond that.

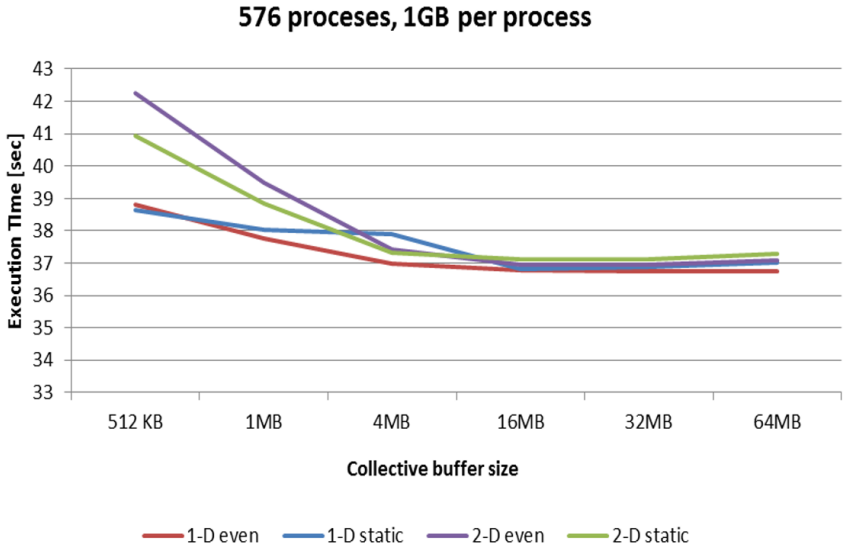

Figure 5.7: Crill-IB - Influence of collective buffer size.

In real life, the even and static partitioning strategies are used with very different collective buffer sizes. Consider an HPC system connected to both, a Lustre storage as well as a GPFS storage system. MPI I/O libraries use the static partitioning for the Lustre file system, with b_c typically being equal to the stripe size of the file system, e.g., very often 1 MB. On the other hand, for a GPFS file system the even partitioning strategy leads to better overall performance, and the collective buffer size for this file system is often in the range of 32 MB. Figure 5.8 shows the difference an application using a 2-D data decomposition strategy would experience in the communication costs of a collective I/O operation, depending on the file system. The results indicate, that purely from the communication costs perspective, using the Lustre file system with a collective buffer size of 1 MB would lead to a significant performance degradation compared to the other file system.

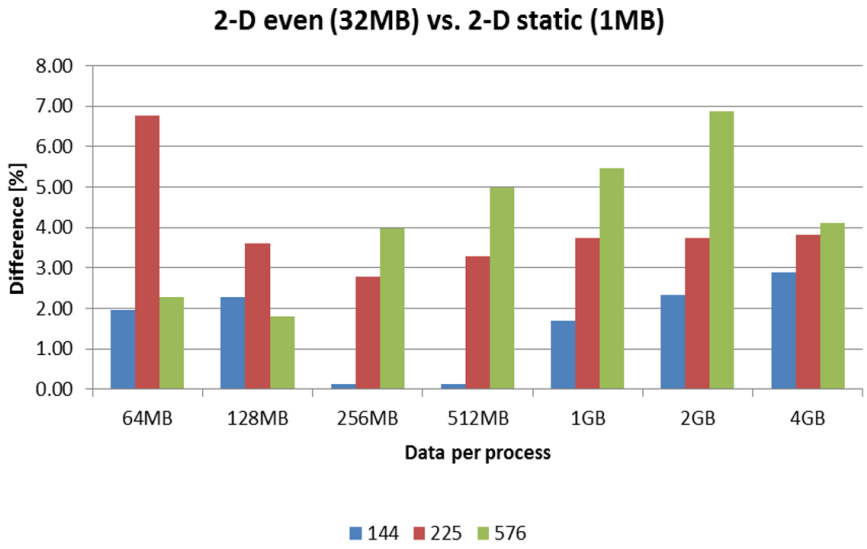

Figure 5.8: Comparison of the communication costs of the even partitioning strategy with a collective buffer size of 32 MB vs. the static partitioning strategy and a collective buffer size of 1 MB.

5.2.2 Projections for large process counts

Performance models such as the one developed in this chapter allow to make predictions for systems and process counts that might not necessarily be available for testing. For example, in Figure 5.9 and Figure 5.10 we show a study which allows us to predict the number of aggregators required for a collective write operation when using 10000 and 250,000 MPI processes. The LogGP parameters used in this study are based on the newest of the three networks available to us, namely the 56 Gbit Ethernet network. Such a study allows system architects to design the overall system characteristics, for e.g., in the static partitioning strategy there is often a direct correlation between the number of aggregator processes and the number of Object Storage Server utilized. The results shown in Figure 5.9 indicates a necessity to have

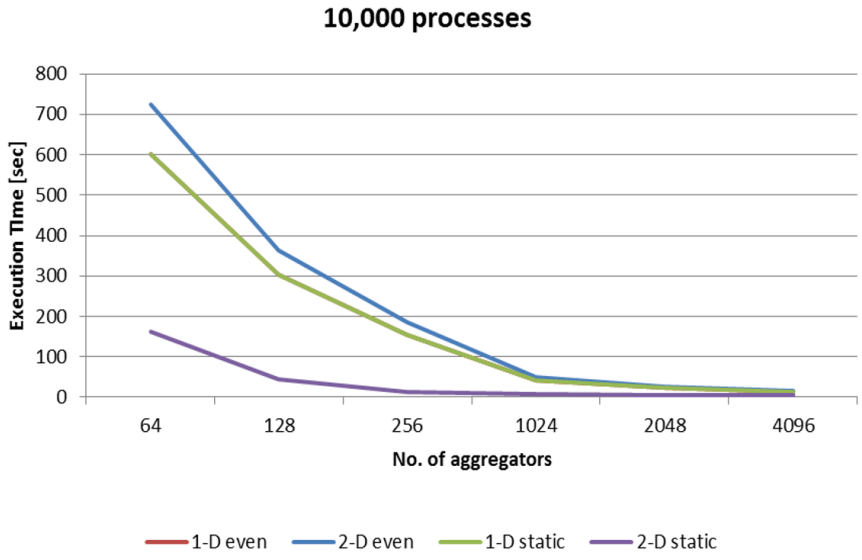

Figure 5.9: Communication costs for 10 k processes for different number of aggregators.

1̃k aggregators and Figure 5.10 necessitates to have somewhere around 8 k - 16 k aggregators to minimize the communication costs of the collective write operation.

5.3 Comparison to actual measurements

In this section, we evaluate how well the performance models derived in section 5.1 match the actual measurements. Tests in this section have been executed on the Crill cluster at the University of Houston using a pre-release version of Open MPI 2.1. The Crill cluster consists of 16 nodes with four 12-core AMD Opteron processor cores each (48 cores per node, 768 cores total) and 64 GB of main memory per node. We used in the subsequent tests the 4X DDR InfiniBand of this cluster.

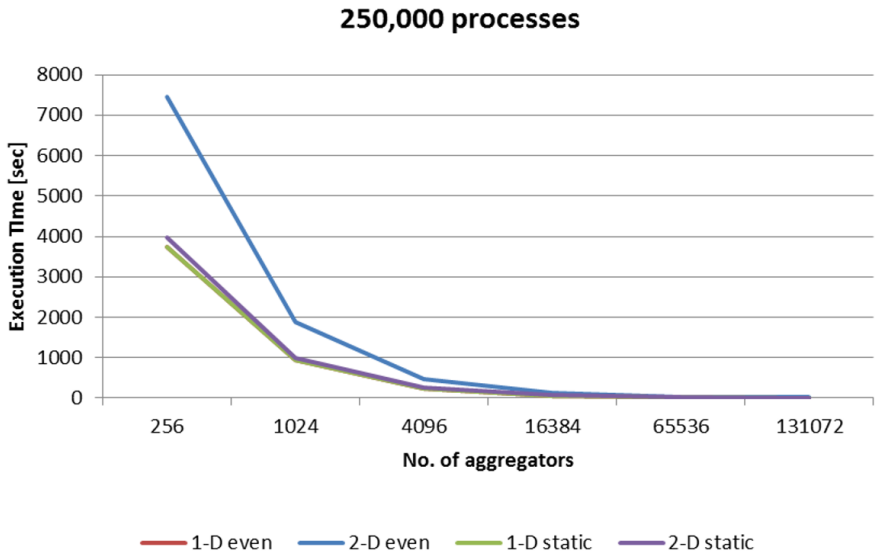

Figure 5.10: Communication costs for 250 k processes for different number of aggregators.

To perform the required measurements, we modified the source code of the *dynamic_gen2* fcoll component in Open MPI to perform the communication in the collective write operations only, i.e. removing the actual file system write operations performed by the aggregators, and measure on a per process basis the time spent in the shuffle step. Furthermore, the *dynamic_gen* component has been modified for our tests such that it is able to perform both, the static and the even partitioning strategy, removing therefore all other potential sources for differences between the two partitioning strategies.

Two benchmarks performing the 1-D and 2-D data decomposition of a two-dimensional array of byte values have been developed, the different data decomposition used reflected in the file-view registered with the MPI file. The benchmarks contain a loop performing a collective write operation a fixed number of times, in our

case 20 iterations. For each collective write operation, the time spent in communication is measured internally in the Open MPI component on each process separately, and the maximum time across all processes is reported as the time spent in communication for each invocation. The benchmark reports the average and the maximum time spent in communication operations across the 20 iterations. Furthermore, each benchmark has been executed three times for each scenario, providing ultimately for each scenario average and maximum values across 60 executions of the collective write operations.

Tests have been executed for 36, 64, 144, 256, and 576 processes for data sizes of $(3,800 \times 3,800)$ bytes ($\sim$13 MB), $(8,192 \times 8,192)$ bytes (64 MB), and $(16,384 \times 16,384)$ bytes (256 MB) per process, leading to overall file sizes between 468 MB and 144 GB. Figure 5.11 and Figure 5.12 present a subset of the results obtained in our tests. The first one depicts the measured and predicted communication times of a collective write operation using the static partitioning strategy performed by 225 processes, each process writing 13 MB data. The second figure provides the same information for the even partitioning strategy using 576 processes and 64 MB per process. The communication times shown are normalized to the communication times obtained using the smallest number of aggregators used in that scenario to simplify the representation of the numbers.

The analysis indicate that the overall tendencies between predicted and measured values are very similar. The performance improvement predicted by our models when increasing the number of aggregators is higher than the actual improvement observed. The difference stems from some limitations of the LogGP model, which

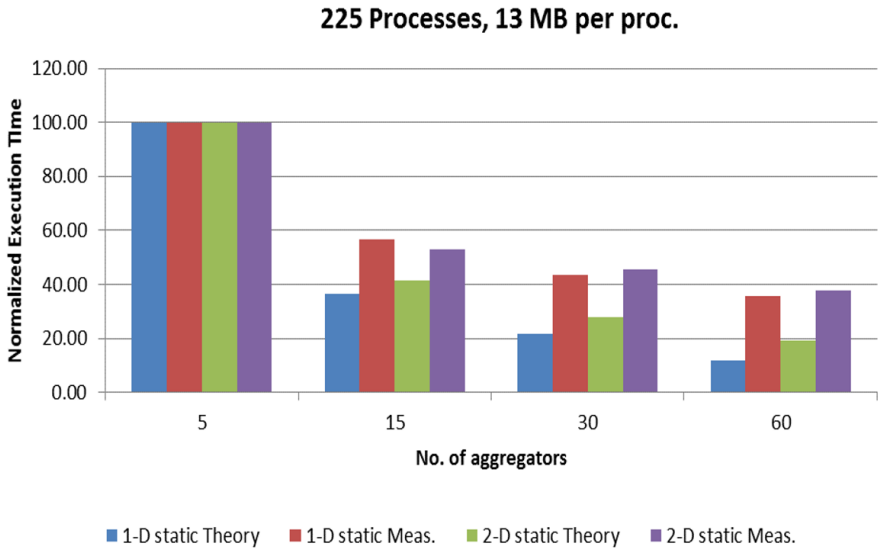

Figure 5.11: Comparison of the measured and predicted normalized communication times of a collective write operation using 225 processes, 13 MB per process using the static partitioning strategy.

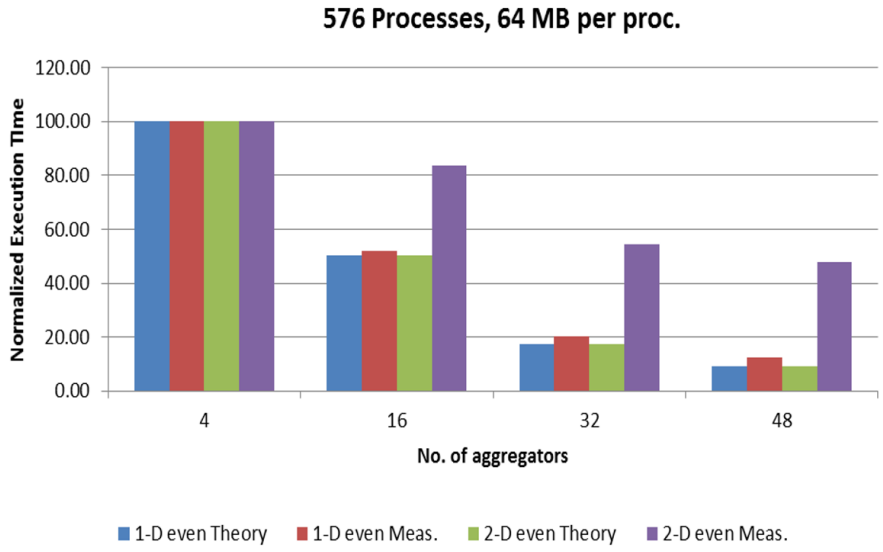

Figure 5.12: Comparison of the measured and predicted normalized communication times of a collective write operation using 576 processes, 64 MB per process using the even partitioning strategy.

does not handle potential congestion occuring on the receiver side if many processes try to communicate with a single aggregator.

To perform a systematic evaluation of the accuracy of our models vs. the measurements over all data points, we analyzed the ratio of two consecutive data points for some well defined scenarios. Specifically, we considered the following three cases:

1. Case 1: varying the datasize per process while maintaining a constant value for the process count and the number of aggregators,

2. Case 2: varying the number of aggregators while maintaining a constant value for the process count and the datasize,

3. Case 3: varying process count while maintaining a constant value for the number of aggregators and the datasize.

For our analysis, we consider a match between the model and the actual measurement if two subsequent data points in a scenario have the same tendency. Using an example from case 2, given a fixed number of processes and a fixed problem size, if increasing the number of aggregators lead to a decrease in the communication time in both the predictions made by our model and the actual measurements, we consider this a match between the model and the data. If the models would predict a decrease in the communication time, while the data shows an increase in communication time, we would consider this a mismatch between the data and the model. Table 5.3, 5.4, and 5.5 summarizes the results of the three cases.

Our experiments show a very good match in most cases between the models and the measurements. There were 0 mismatches for both even and static file partitioning

Table 5.3: Case 1: Evaluation of the models by varying the datasize per process (the process count and the number of aggregators are constant)

	match	mismatch
Even 1D	37	0
Even 2D	36	0
Static 1D	40	0
Static 2D	20	0

Table 5.4: Case 2: Evaluation of the models by varying the number of aggregators (the process count and the datasize are constant)

	match	mismatch
Even 1D	38	4
Even 2D	39	3
Static 1D	37	8
Static 2D	36	7

in case 1 for both 1-D and 2-D data distributions. For case 2, the even 1-D data distribution had 38 matches and 4 mismatches, even 2-D had 39 matches vs. 3 mismatches. The worst compliance between models and measurements were given for the static in case 2, with 37 matches and 8 mismatches for the 1-D data distribution and 36 matches and 7 mismatches for the 2-D scenario. For case 3, the results are in the range of 20 matches and 1 mismatch for the static 1-D scenario and 17 matches with 3 mismatches for the even 1-D scenario. Ultimately, we conclude based on this analysis that the models we derived can be used to predict tendencies for the cost of collective communication in various data decomposition and file domain partitioning strategies.

Table 5.5: Case 3: Evaluation of the models by varying process count (the number of aggregators and the datasize are constant)

	match	mismatch
Even 1D	17	3
Even 2D	17	2
Static 1D	20	1
Static 2D	19	1

5.4 I/O performance modeling

While just looking at the above discussed communication model, some might argue that parameters like n_s, number of send cycles that each process is involved in eq. 5.1 and n_r, number of recieve cycles that each aggregator is involved in eq. 5.2 are not necessary as the LogGP model handles multiple messages efficiently. But these parameters play a vital role while looking at the bigger picture of I/O, i.e. each aggregator is involved in receive function and then in an actual I/O operation to the disk. Thus, each aggregator is involved in n_r cycles of I/O. Therefore, the above discussed model can be extended to collective I/O. However, modeling the whole I/O operation is non-trivial, as:

1. I/O performance depends on the hardware characteristics the specific hardware configuration limits potential network throughput, computation power and available memory bandwidth. The selection of the optimal algorithm depends on the hardware characteristics, the network topology and application behavior. This dependency on component characteristics are mostly non-linear.

2. Even with the similar components, file access time varies according to the I/O path chosen.

In spite of these difficulties, a number of researchers worked on I/O modeling, concentrating more on following two types:

1. To have an analytical model that captures the whole I/O operation. Various projects aim at this modeling approach. In [93], Machine learning techniques such as decision tree for optimal parameter setting that can be used for performance prediction. In [32] the authors developed a performance model for I/O-loads. However the model considers a simple I/O workload. This can however be extended to parallel I/O. Model developed here are used to calculate the mean response time and access time for read and write operation.

2. System that simulates the I/O [94]. This is also called white-box modeling [95]. This provides not only very precise models but also the option of analyzing the performance later using trace files. One of the major limitation of these model is that every system or implementation needs to be treated individually.

   PIOsimHD is one such event driven simulator which can evaluate MPI-IO implementations. It is a hardware model where each component have many implementations. These implementations has characteristics as parameters. Sequential transfer rate, average access time, track-to-track seek time and RPM are used to model the I/O.

5.5 Conclusions

In this chapter, we derived performance models for the communication occurring in collective write operations for the two most widely used file partitioning strategies

used in MPI I/O libraries today. Our models take further the data decomposition used by the parallel applications into account. We discuss properties of our performance models and demonstrate using hardware parameters derived on various platforms the potential impact on the performance of collective I/O operations. We further provided the comparison to actual measurements performed on an InfiniBand cluster. Our results indicate generally a good match between predicted and observed behavior, although some minor discrepancies due to limitations of the LogGP model as well as some simplifications introduced by our models are present.

The work can be extended in multiple directions. The first step involves extending the existing models that were derived for collective write operations for collective read operations as well. The more challenging extension will require including actual I/O operations into our models, which is highly challenging due to the caching effects occurring at various levels. A full evaluation of the predicted performance models with realistic benchmarks and/or applications is also anticipated as a final step.

Chapter 6

Summary

6.1 Contributions

The overall goal of this disseratation is to derive a model to predict the performance of collective communication and data shuffling of I/O in HPC applications. To achieve this, three specific aspects have been targeted

- A personalized MPI library

  Open MPI has a large number of runtime parameters that influence the performance of an application on a particular platform. To minimize the time required to tune these runtime parameters, we introduced the notion of a personalized MPI library by creating a custom set of runtime parameters for a particular application and platform. We have evaluated the effectiveness of the personlized MPI library by comparing the time required to tune multiple parameters for various benchmarks using three different search algorithms,namely

'Brute force search algorithm', 'attribute based search algorithm' and '2 K factorial search algorithm'. The results indicate that the tuning time for the 2 K factorial search algorithm is significantly lesser in all scenarios when compared to the other two search algorithms, while the qualities of the solutions found were either identical or very close to the solution derived from an exhaustive search. We also introduced a framework developed for the purpose of storing application and platform specific parameter sets in a database and retrieving these parameter sets using the mpiexec tool of the MPI library.

- Performance modeling of collective communication

  We developed various theoretical models in order to obtain the expected performance improvement for multiple collective operations and algorithms, given an improvement in the performance of a point-to-point operation of a particular message length. We focussed on collective operations because they are commonly used in parallel applications and have been analyzed extensively from a theoretical and practical perspective. Our key findings include:

  1. a demonstration that based on the models developed, many algorithms and collective operations inherently show a different, often lower performance improvement compared to the performance benefit observed for an individual point-to-point operation,
  2. a good match in the predictions made by our models and the actual observations, in respect of many algorithms and collective operations.

- Performance modeling of collective I/O

  Performance models are derived for the communication occurring in collective write operations for the two most widely used file partitioning strategies used in MPI I/O libraries today. These models take the data decomposition used by the parallel applications into account and demostrate the potential impact of hardware parameters on the performance of collective I/O operations. They also provides a comparison with the actual measurements taken on an InfiniBand cluster. The results obtained showed a significant match between the predicted behavoir and the behavior actually observed, with minor discrepancies observed on account of limitations of the LogGP model, and also on account of a few simplifications introduced by the present models.

6.2 Future work

All the three tasks discussed above can be further developed with more accurate and complete model of HPC application.

- Currently the database managed by the personalized MPI library has limited number of attributes as metadata and to retrieve an optimized parameter (from the database) for a particular scenario, query has to have a match for all of these attributes. However, this can be further developed to allow for more generic query options, and thus broaden the utilization of parameters to more applications and scenarios. This would necessitate the introduction of different

categories of runtime parameters in the database, such as networking parameters, collective parameters etc. Additionally, a number of other algorithms from experimental design theory can also be investigated to extend the search algorithms and make them more robust.

- The performance models of all three collective communications can be extended by including other collective operations and network interconnects. An interesting extension would be the incorporation of multiple message lengths in the model, as applications generally have more that one message length.

- For performance modeling of collective I/O, models are derived for collective write operation, this can be extended to collective read operations. The more challenging extension however would entail the inclusion of actual I/O operations into the already developed models. The challenge however would be to overcome the problem of caching. The performance models could also be used to compare performance with applications or realistic benchmarks.

Bibliography

[1] Richard M. Russell. The cray-1 computer system. *Commun. ACM*, 21(1):63–72, January 1978.

[2] Top500 supercomputer. In *Proceedings of the 2014 ACM/IEEE Conference on Supercomputing*, SC '14, 2014.

[3] I.T. Association. InfiniBand architecture specification. Release 1.2. `http://www.infinibandta.org/specs`, 2004.

[4] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Commun. ACM*, 19(7):395–404, July 1976.

[5] Mark S. Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D. Underwood, and Robert C. Zak. Enabling scalable high-performance systems with the intel omni-path architecture. *IEEE Micro*, 36(4):38–47, July 2016.

[6] The Open Group. POSIX FAQ. Online; accessed 11-nov-2015.

[7] LANL OpenMP tutorial. Online; accessed 11-nov-2015.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[9] V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339, November 1990.

[10] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[11] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22(6):789–828, September 1996.

[12] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[13] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[14] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: Co-array fortran and unified parallel c. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 36–47, New York, NY, USA, 2005. ACM.

[15] UPC consortium. Upc language specification v1.2,. In *Lawrence Berkeley National Lab Tech Report LBNL-59208*, 2005.

[16] J. Nieplocha, M. Krishnan, V. Tipparaju, and B. Palmer. Global arrays user manual.

[17] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug 1998.

[18] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance java dialect. In *ACM*, pages 10–11, Sep 1998.

[19] Aaron Dubrow. Hpcwire accelerator. `http://www.hpcwire.com/2011/08/15`. Online; accessed 11-Nov-2015.

[20] Nvidia launches the world's first graphics processing unit: Geforce 256. `http://www.nvidia.com/object/IO_20020111_5424.html`. Online; accessed 20-Feb-2017.

[21] Cell broadband engine architecture and its first implementation. `https://www.ibm.com/developerworks/library/pa-cellperf/`. Online; accessed 20-Feb-2017.

[22] Field programmable gate array (FPGA). https://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm. Online; accessed 20-Feb-2017.

[23] Intel XEON PHI processors. http://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors.html. Online; accessed 20-Feb-2017.

[24] Nvidia CUDA homepage. http://www.nvidia.com/object/cuda_home_new.html. Online; accessed 11-nov-2015.

[25] OpenACC Homepage. http://www.openacc.org. Online; accessed 11-nov-2015.

[26] 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*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[27] 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*, 1996.

[28] Ralf Reussner, Peter Sanders, Lutz Prechelt, and Matthias Müller. Skampi: A detailed, accurate mpi benchmark. In *Proceedings of the 5th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 52–59, London, UK, UK, 1998. Springer-Verlag.

[29] Thilo Kielmann, Henri E. Bal, and Kees Verstoep. Fast measurement of logp parameters for message passing platforms. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, IPDPS '00, pages 1176–1183, London, UK, UK, 2000. Springer-Verlag.

[30] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, 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.

[31] 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.

[32] Elizabeth Varki, Xiaozhou Qiu, and Arif Merchant. An analytical performance model of disk arrays under synchronous I/O workloads. , University of New Hampshire, Jan 2003.

[33] S. D. Hammond, G. R. Mudalige, J.A. Smith, S.A. Jarvis, J.A. Herdman, and A. Vadgama. Warpp - a toolkit for simulating high-performance parallel scientific codes. In *International Conference on Simulation Tools and Techniques (SIMUTOOLS 2009)*. ICST, 2009.

[34] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. Pace-a toolset for the performance prediction of parallel and distributed systems. *Int. J. High Perform. Comput. Appl.*, 14(3):228–251, August 2000.

[35] Julian M. Kunkel. Simulating parallel programs on application and system level. *Comput. Sci.*, 28(2-3):167–174, May 2013.

[36] Ewa Deelman, Aditya Dube, Adolfy Hoisie, Yong Luo, Richard L. Oliver, David Sundaram-Stukel, Harvey Wasserman, Vikram S. Adve, Rajive Bagrodia, James C. Browne, Elias Houstis, Olaf Lubeck, John Rice, Patricia J. Teller, and Mary K. Vernon. Poems: End-to-end performance design of large parallel adaptive computational systems. In *In Proceedings of First International Workshop on Software and Performance (WOSP*, pages 18–30, 1998.

[37] Sabri Pllana and Thomas Fahringer. Performance prophet: A performance modeling and prediction tool for parallel and distributed programs. In *In Proceedings of The 2005 International Conference on Parallel Processing (ICPP-05.*

[38] S. Jha, E. Gabriel, and S. Feki. A Personalized MPI library for Exascale Applications and Environments (Hot Topic paper). In *Workshop on Exascale MPI 2014, SC 2014*, New Orleans, LA, 2014.

[39] S. Jha and E. Gabriel. Impact and limitations of point-to-point performance on collective algorithms. *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 00:261–266, 2016.

[40] S. Jha and E. Gabriel. Performance Models for Communication in Collective I/O Operations . In *International Workshop on Theoretical Approaches to Performance Evaluation, Modeling and Simulation, (TAPEMS), held in conjunction with CCGRID*, Madrid, Spain, 2017.

[41] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[42] Matteo Frigo, Steven, and G. Johnson. The design and implementation of fftw3. In *Proceedings of the IEEE*, pages 216–231, 2005.

[43] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 303–316, New York, NY, USA, 2014. ACM.

[44] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Towards an accurate model for collective communications. In *Proceedings of the International Conference on Computational Sciences-Part I*, ICCS '01, pages 41–50, London, UK, UK, 2001. Springer-Verlag.

[45] Michael J. Voss and Rudolf Eigenmann. Adapt: Automated de-coupled adaptive program transformation. In *In Proc. ICPP*, pages 163–170, 2000.

[46] Protonu Basu, Mary Hall, Malik Khan, Suchit Maindola, Saurav Muralidharan, Shreyas Ramalingam, Axel Rivera, Manu Shantharam, and Anand Venkat. Towards making autotuning mainstream. *Int. J. High Perform. Comput. Appl.*, 27(4):379–393, November 2013.

[47] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: Towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[48] Intel Software Autotuning Tool (ISAT). `https://software.intel.com/en-us/articles/intel-software-autotuning-tool`. Online; accessed 23-nov-2015.

[49] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[50] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, Washington, 1998.

[51] Victor Eijkhout, Erika Fuentes, Thomas Eidson, and Jack Dongarra. The component structure of a self-adapting numerical software system. *International Journal of Parallel Programming*, 33(2-3):137–143, 2005.

[52] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, R. Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. In *Proceedings of the IEEE*, page 2005, 2005.

[53] Anirudh Jayakumar, Prakash Murali, and Sathish Vadhiyar. Matching application signatures for performance predictions using a single execution. In *IPDPS*, pages 1161–1170. IEEE Computer Society, 2015.

[54] Sanath Jayasena, Milinda Fernando, Tharindu Rusira, Chalitha Perera, and Chamara Philips. Auto-tuning the Java Virtual Machine. *10th IEEE International Workshop on Automatic Performance Tuning*, pages 581–600, May 2015.

[55] Simone Pellegrini, Radu Prodan, and Thomas Fahringer. Tuning mpi runtime parameter setting for high performance computing. In *CLUSTER Workshops*, pages 213–221. IEEE, 2012.

[56] 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*, ICS '06, pages 199–208, New York, NY, USA, 2006. ACM.

[57] E. Gallardo, J. Vienne, L. Fialho, P. Teller, and J. Browne. MPI Advisor: a Minimal Overhead MPI Performance Tuning Tool. *EuroMPI*, September 2015.

[58] Edgar Gabriel and Shuo Huang. Runtime optimization of application level communication patterns. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pages 1–8, 2007.

[59] Edgar Gabriel, Saber Feki, Katharina Benkert, and Michael M. Resch. Towards performance portability through runtime adaptation for high-performance computing applications. *Concurr. Comput. : Pract. Exper.*, 22(16):2230–2246, November 2010.

[60] E. Gabriel, S. Feki, K. Benkert, and M. Chaarawi. The abstract data and communication library . *Journal of Algorithms and Computational Technology*, pages 581–600, Dec 2008.

[61] Saber Feki and Edgar Gabriel. Incorporating historic knowledge into a communication library for self-optimizing high performance computing applications. In Sven A. Brueckner, Paul Robertson, and Umesh Bellur, editors, *SASO*, pages 265–274. IEEE Computer Society, 2008.

[62] Jelena Pješivac-Grbović, George Bosilca, Graham E. Fagg, Thara Angskun, and Jack J. Dongarra. Mpi collective algorithm selection and quadtree encoding. *Parallel Comput.*, 33(9):613–623, September 2007.

[63] Richard Vuduc, James W. Demmel, and Jeff A. Bilmes. Statistical models for empirical search-based performance tuning. *Int. J. High Perform. Comput. Appl.*, 18(1):65–94, February 2004.

[64] Simone Pellegrini, Jie Wang, Thomas Fahringer, and Hans Moritsch. Optimizing mpi runtime parameter settings by using machine learning. In Matti Ropo, Jan Westerholm, and Jack Dongarra, editors, *PVM/MPI*, volume 5759 of *Lecture Notes in Computer Science*, pages 196–206. Springer, 2009.

[65] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Comput. Oper. Res.*, 13(5):533–549, May 1986.

[66] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[67] Frank Winkler. vampir. `https://www.alcf.anl.gov/files/Vampir.pdf`. Online; accessed 11-Nov-2015.

[68] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, 22(6):702–719, April 2010.

[69] David Böhme, Markus Geimer, Felix Wolf, and Lukas Arnold. Identifying the root causes of wait states in large-scale parallel applications. In *Proc. of the 39th International Conference on Parallel Processing (ICPP), San Diego, CA, USA*, pages 90–100. IEEE Computer Society, September 2010. Best Paper Award.

[70] PMPI. `https://www.open-mpi.org/faq/?category=perftools`. Online; accessed 11-Nov-2015.

[71] MPIT. `https://computation.llnl.gov/project/mpi_t`. Online; accessed 11-Nov-2015.

[72] Roger W. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Comput.*, 20(3):389–398, March 1994.

[73] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. Technical report, Berkeley, CA, USA, 1992.

[74] Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara. Loggps: A parallel computational model for synchronization analysis. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 133–142, New York, NY, USA, 2001. ACM.

[75] 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.

[76] Huaiming Song, Yanlong Yin, Yong Chen, and Xian-He Sun. A cost-intelligent application-specific data layout scheme for parallel file systems. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 37–48, New York, NY, USA, 2011. ACM.

[77] Kwangho Cha and Seungryoul Maeng. Reducing communication costs in collective I/O in multi-core cluster systems with non-exclusive scheduling. *The Journal of Supercomputing*, 61(3):966–996, Sep 2012.

[78] Jialin Liu, Yong Chen, and Yi Zhuang. Hierarchical I/O scheduling for collective I/O. In *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013, Delft, Netherlands, May 13-16, 2013*, pages 211–218. IEEE Computer Society, 2013.

[79] Yin Lu, Yong Chen, Yu Zhuang, and Rajeev Thakur. Memory-conscious collective I/O for extreme scale HPC systems. In Torsten Hoefler and Kamil Iskra, editors, *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers, ROSS 2013, Eugene, Oregon, USA, June 10, 2013*, pages 5:1–5:8. ACM, 2013.

[80] F. Isaila, P. Balaprakash, S. M. Wild, D. Kimpe, R. Latham, R. Ross, and P. D. Hovland. Collective I/O tuning using analytical and machine learning models. In *2015 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2015. Acceptance rate 24

[81] J. Pjesivac-Grbovic. *Towards Automatic and Adaptive Optimizations of MPI Collective Operations*. PhD thesis, University of Tennessee - Knoxville, 2007.

[82] Rajeev Thakur and Rolf Rabenseifner. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 19:49–66, Feb 2005.

[83] E. W. Chan, M. F. Heimlich, Avi Purkayastha, and Robert A. van de Geijn. On optimizing collective communication. In *2004 IEEE International Conference on Cluster Computing (CLUSTER 2004), September 20-23 2004, San Diego, California, USA*, pages 145–155, Sep 2004.

[84] Jing Chen, Yunquan Zhang, Linbo Zhang, and Wei Yuan. Performance evaluation of allgather algorithms on terascale linux cluster with fast ethernet. *High Performance Computing and Grid in Asia Pacific Region, International Conference on*, 0:437–442, Dec 2005.

[85] T. Hoefler, A. Lichei, and W. Rehm. Low-Overhead LogGP Parameter Assessment for Modern Interconnect Networks. In *Proceedings of the IPDPS*. IEEE, March 2007.

[86] Mohamad Chaarawi and Edgar Gabriel. Automatically Selecting the Number of Aggregators for Collective I/O Operations. In *Workshop on Interfaces and Abstractions for Scientific Data Storage, IEEE Cluster 2011 conference*, page t.b.d, Austin, Texas, USA, 2011.

[87] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105. ACM Press, 1995.

[88] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *FRONTIERS 99: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, page 182. IEEE Computer Society, 1999.

[89] Wei keng Liao and Alok Choudhary. Dynamically adapting file domain partitioning methods for collective i/o based on underlying parallel file system locking protocols. In *Proceedings of the 2008 IEEE/ACM Supercomputing Conference*, pages 1–12. IEEE Computer Society, 2008.

[90] Mohamad Chaarawi, Suneet Chandok, and Edgar Gabriel. Performance Evaluation of Collective Write Algorithms in MPI I/O. In *Proceedings of the International Conference on Computational Science (ICCS)*, volume 5544, pages 185–194, Baton Rouge, USA, 2009.

[91] Jelena Pjesivac-Grbovic, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack J. Dongarra. Performance Analysis of MPI Collective Operations. *Cluster Computing*, 10(2):127–143, Mar 2007.

[92] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm. Netgauge: A Network Performance Measurement Framework. In *Proceedings of High Performance Computing and Communications, HPCC'07*, volume 4782, pages 659–671. Springer, Sep. 2007.

[93] Julian M. Kunkel, Michaela Zimmer, and Eugen Betke. Predicting performance of non-contiguous I/O with machine learning. In Julian M. Kunkel and Thomas Ludwig, editors, *High Performance Computing - 30th International Conference, ISC, High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings*, volume 9137 of *Lecture Notes in Computer Science*, pages 257–273. Springer, 2015.

[94] Julian Martin Kunkel. Using simulation to validate performance of MPI(-IO) implementations. In Julian M. Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, *Supercomputing - 28th International Supercomputing Conference, ISC, 2013, Leipzig, Germany, June 16-20, 2013. Proceedings*, volume 7905 of *Lecture Notes in Computer Science*, pages 181–195. Springer, 2013.

[95] Adam Crume, Carlos Maltzahn, Lee Ward, Thomas Kroeger, Matthew Curry, and Ron Oldfield. Fourier-assisted machine learning of hard disk drive access time models. In *Proceedings of the 8th Parallel Data Storage Workshop*, PDSW '13, pages 45–51, New York, NY, USA, 2013. ACM.