University College Dublin

Hierarchical Approach to Optimization of MPI Collective Communication Algorithms

Khalid Hasanov

This thesis is submitted to University College Dublin

in fulfilment of the requirements for the degree of

Doctor of Philosophy in Computer Science

School of Computer Science

Head of School: Pádraig Cunningham

Research Supervisor: Alexey Lastovetsky

October 2015

Acknowledgements

There are a number of people behind this thesis who deserve to be both acknowledged and thanked here.

Words can never be enough to express my utmost gratitude to my supervisor, Dr. Alexey Lastovetsky for giving me the opportunity to do my PhD in the Heterogeneous Computing Laboratory; for his patience, support, faith, and for always being there whenever I needed encouragement; for going beyond his duties to fight my worries and frustrations. Put simply, I cannot imagine a better supervisor than him.

Thank you to all my colleagues from the Heterogeneous Computing Laboratory, Jean-Noël Quintin, Vladimir Rychkov, Kiril Dichev, Ashley DeFlumere, Zhong Ziming, Ken O'Brien, David Clarke, Tania Malik, Oleg Girko, Amani Al Onazi, Jun Zhu.

This thesis is the result of research conducted with the financial support by IRCSET(Irish Research Council for Science, Engineering and Technology) and IBM, grant number EPSPG/2011/188. The experiments were carried out on Grid'5000 developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies. Another part of the experiments were carried out using the resources of the Supercomputing Laboratory at King Abdullah University of Science&Technology (KAUST) in Thuwal, Saudi Arabia.

I am forever indebted to my parents, Azer and Ofelya for everything I am and everything I will ever be. I thank my brother Vasif for always encouraging me throughout my study.

I thank my wife Aynur for her understanding and continues support. Last but by no means least, I am thankful to my little angel Fidan for bringing so much happiness into our life.

DEDICATION

To

My Parents, My Brother, My Wife and My Daughter

Abstract

A significant proportion of scientific applications developed for execution on high-end computing systems are based on parallel algorithms proposed between the 1970s and 1990s. These algorithms were designed with relatively small computing systems in mind and tested on such systems. Indeed, in 1995, the number of cores in the top 10 supercomputers ranged from 42 to 3680 [1]. Nowadays, in mid 2015, this number ranges from 147,456 to 3,120,000. Thus, over last two decades the number of processors in HPC systems has increased by three orders of magnitude. This drastic increase in scale significantly increases the cost of coordination and interaction of processes in traditional message-passing data-parallel applications. In other words, it increases their communication cost. In these applications, all processes are peers and the number of directly interacting processes grows quickly with the increase of their total number.

We address the problem of reduction of the communication cost of such traditional message-passing data-parallel applications on large-scale distributed-memory computing systems. The approach we propose is a traditional methodology widely used for dealing with the complexity of coordination and management of a large number of actors, namely, the hierarchical approach. According to this technique, thousands or millions of actors are structured, and instead of interacting with a large number of peers,

they coordinate their activities with one superior and a small number of peers and inferiors. This way the overhead of interaction is significantly reduced.

We present in this thesis how the hierarchical approach can be applied in a topology-oblivious way to optimize the communication cost of widely used MPI collective communication operations. The thesis demonstrates theoretical and experimental study of hierarchical transformations of the state-of-the-art algorithms used to implement MPI broadcast, reduce, allreduce, scatter and gather operations. The experimental validation of the approach and an application study in the context of parallel matrix-matrix multiplication on a medium-scale cluster and on a large-scale IBM BlueGene/P machine are demonstrated. We have implemented the hierarchical algorithms in a software library called Hierarchical MPI (HiMPI). The HiMPI library is layered on top of the existing MPI implementations, provides automatic estimation and selections of the optimal parameters in the hierarchical algorithms and is portable to any parallel platforms that supports MPI.

Contents

List of Figures

List of Tables

Statement of Original Authorship

I hereby certify that the submitted work is my own work, was completed while registered as a candidate for the degree stated on the Title Page, and I have not obtained a degree elsewhere on the basis of the research presented in this submitted work.

Chapter 1

Introduction

1.1 Motivation and Contributions

1.1.1 Motivation

A significant proportion of scientific applications developed for execution on high-end computing systems are based on parallel algorithms proposed between the 1970s and 1990s. These algorithms were designed with relatively small computing systems in mind and tested on such systems. Indeed, in 1995, the number of cores in the top 10 supercomputers ranged from 42 to 3680 [1]. Nowadays, in mid 2015, this number ranges from 115,984 to 3,120,000. Thus, over last two decades the number of processors in HPC systems has increased by three orders of magnitude. This drastic increase in scale significantly increases the cost of coordination and interaction of processes in traditional message-passing data-parallel applications. In other words, it increases their communication cost. In these applications, all processes are peers and the number of directly interacting processes grows quickly with the increase of their total number.

The existing works on optimization of the communication cost in messages-passing data-parallel applications can be classified into two main categories:

- Topology aware optimization of communication cost.

- Topology oblivious optimization of communication cost.

1.1.1.1 Topology Aware Optimization of Communication Cost

The key idea in the topology aware optimization methodology is using the topology as input to build optimal communication trees. This approach is used both at the MPI level and the application level. The MPI level optimizations mainly focus on the MPI collective communication operations as part of the MPI libraries where low-level optimizations, e.g. using hardware specific data access mechanism and tuning underlying network protocol parameters, are hidden from the application developer.

The efficient implementation of topology aware collective communication operations has received much attention. Such implementations mainly focus on heterogenous platforms where the communication times between process pairs are not equal. One of the earliest works in this area was done in the MPI-StarT [2], an MPI implementation for a cluster of SMPs interconnected by a high-performance interconnect. The authors demonstrate a topology aware optimization that treats inter- and intra-cluster communications differently and minimizes the inter-cluster communication cost of collective operations. The similar approach was used in the MagPIe [3] library, which assumes a two-level communication network and distinguishes between local and wide area communication. Later introduced MPICH-G [4][5], a grid-enabled implementation of MPI, applies the similar optimization technique to networks having more than two levels of hierarchy. More recent research work [6] uses similar approach and discusses hierarchical implementation of MPI collective communication operations by taking hardware specific data access mechanisms into consideration. That work is implemented in a framework called Cheetah and incorporated into Open MPI. The main idea behind all these topology and platform aware techniques is to take the existing generic algorithms and optimize them to specific topology and platforms.

In addition to MPI collective operations, a significant amount of research has been done in topology aware optimization of communication cost in

scientific applications. The authors in [7] discuss improving communication cost in parallel matrix multiplication and LU factorization by topology aware mapping of processes and as a result using effective collective communication operations designed for IBM Blue Gene/P supercomputers. A more recent work in [8] demonstrates a heuristic topology-aware approach to optimization of communication cost in parallel matrix multiplication on hierarchical heterogeneous platforms. Unlike existing methods, the authors take into account not only the topology but also the heterogeneity of the processors and the entire communication flow of the application.

1.1.1.2 Topology Oblivious Optimization of Communication Cost

Topology oblivious optimizations of communication cost in parallel algorithms on distributed-memory platforms usually depend on a lesser number of parameters in comparison to the topology or platform aware optimizations. As an example, optimal data partitioning, which is a widely used technique to minimize communication cost of message-passing data parallel applications, takes the number of processes and their speeds as the only input parameters, while ignoring the underlying topology, the underlying platform, and the communication network parameters. Despite being quite simple, such a model can still take into account the communication cost of a parallel algorithm. Two widely used techniques of this kind are the constant and functional performance models [9]. In the first case the model includes numeric parameters, the number of processors and the speed of each processor. While in the case of functional performance models the speed of each processor is defined as a continuous function of the task size. Using these performance models the fundamental optimization problem of data partitioning can be reduced to the problem of partitioning some mathematical objects such as sets, matrices, graphs and so forth.

The distinguished idea of the data partitioning approach is that it is very application specific. Therefore it is necessary to design specific optimal data partitioning algorithms for different applications. Moreover, even a single application, for example, a parallel matrix multiplication, might need different

data partitioning algorithms depending on the target platform, matrix shapes, and problem size.

In addition to topology aware optimizations, the state-of-the-art MPI implementations, such as MPICH and Open MPI, provide a few generic collective communication algorithms that are not specific to any kind of topology or platform. For instance, the broadcast operation is implemented using flat, linear, pipeline, binary, split-binary, binomial tree, scatter-ring-allgather, and scatter-recursive-doubling-algather algorithms. All these algorithms are designed without taking the topology into account and they all are widely used depending on the message size and the number of processes in the communication operation. The same applies to other collective communication operations as well.

1.1.2 Contributions

The existing approaches do not address the issue of scale, as they try to optimize the traditional general-purpose algorithms for specific network topologies and architectures. In contrast to the existing optimization techniques, we focus on more general use cases and try to be effective on a variety of platforms, be applicable to way more applications than the previous techniques. We mainly focus on the scale of the platform while ignoring its complexity. Therefore our main parameter is the number of processors, giving us the reason to employ the simplest communication model, namely the Hockney model [10] in our optimization techniques. The approach we propose is a traditional methodology widely used for dealing with the complexity of coordination and management of a large number of actors, namely, the hierarchical approach. According to this technique, thousands or millions of actors are structured, and instead of interacting with a large number of peers, they coordinate their activities with one superior and a small number of peers and inferiors. This way the overhead of interaction is significantly reduced.

The main contributions of this thesis are topology-oblivious hierarchical optimizations of MPI broadcast, reduce, allreduce, scatter and gather

operations. We also provide an application study in the context of parallel matrix multiplication. Finally, we discuss the design of Hierarchical MPI software library.

The hierarchical optimizations of the MPI broadcast and reduce operations have been published in [11] [12] [13], and the hierarchical optimization of parallel matrix multiplication published in [14] [15].

The content of the thesis is organized as follows. The chapter 2 surveys communication performance models, MPI collective communication operations, algorithms and existing optimization techniques. Then, the chapter 3 introduces our hierarchical approach to optimization of MPI broadcast, reduce, allreduce, scatter and gather operations. Chapter 4 presents our hierarchical parallel matrix multiplication algorithm, chapter 5 discusses hierarchical MPI (HiMPI) software design and finally the chapter 6 concludes the thesis and discusses the future work.

Chapter 2

Background and Related Work

This chapter first reviews the history of the Message Passing Interface (MPI) [16] and outlines alternative parallel programming systems. Then we provide a comprehensive discussion of MPI collective communication operations and the state-of-the-art algorithms used to implement them.

2.1 Message Passing Interface

MPI is a library specification for the message-passing programming paradigm on distributed-memory high-performance computing systems. The specification is defined and maintained by the MPI Forum, which is an open group consisting of many organizations. Communication in MPI applications is performed by message passing between processes. The communication can either be a point-to-point between any two processes, or a collective operation involving a specific group of processes.

The first version (1.0) of the MPI Standard was published in 1994, and since then it is periodically being updated. In 1995 the version 1.1 and in 1997 the version 1.2 were published. In the same year, MPI version 2.0 was published with many new features, most prominently, dynamic process creation and management, parallel I/O and one-sided communication routines. The latest release of MPI-1 series was the version 1.3 and approved by the MPI Forum in 2008. Minor edits and corrected errata of MPI-2 series

with version numbers of 2.1 and 2.2 were published in 2008 and 2009 accordingly.

In 2012, another major release of the MPI Standard, MPI-3 was released and introduced new one-sided communication operations, Fortran bindings, and particularly nonblocking collective operations. Finally, MPI-3.1 is the latest release of the MPI standard by the time of writing this thesis and is mainly focused on errata corrections of the previous release. This version of the MPI standard was approved by the MPI Forum in June 2015.

2.1.1 Alternative Parallel Programming Systems

2.1.1.1 PVM

While MPI is the de facto standard communication library for running parallel programs on distributed memory systems, PVM (Parallel Virtual Machine) [17] existed before MPI and had been used for designing portable parallel programs. The portability of the library was centered around the idea of "virtual machine", which abstracts the underlying connected nodes as a single entity. The high level abstraction of the library comes with easy to use application programming interfaces (API). The PVM API can be used to join or leave the virtual machine, start new processes potentially using a different selection criteria, kill a process, send a signal to a process, testing if a process is responding, and notify any process if some other process disconnects from the system. Having easy process management design, PVM also provides fault tolerance, as processes can easily join or leave the system dynamically.

2.1.1.2 Linda

Linda [18] [19] is a programming language model used for parallel and distributed programming where the main concept is centered around associative shared virtual memory system, or tuple space. The tuple space is used as the technique for process communications. Each tuple of the space consists of an arbitrary number of ordered and typed elements. Tuples can

be created by using either *out()* or *eval()* operations. The first one basically evaluates the expressions that results in tuples in the tuple space. The *eval()* operation is similar to the *out()*, except it is asynchronous in the sense that each field making the tuple is evaluated in parallel. There are two more operations defined in Linda, *rd()* operation reads the tuple and *in()* reads the tuple and removes it from the space.

In contrast to MPI, in Linda the compiler can check the types of data being sent through tuple space. Another difference between the two systems is that in Linda communicating processes are loosely coupled, whereas in MPI each sender process has to know its destination process. Moreover, the MPI Standard provides a large number of APIs, whereas as we have seen Linda consists of only four simple commands.

2.1.1.3 mpC

mpC [20] [21] is a parallel language that was designed for programming high-performance parallel computations on heterogeneous networks of computers. It allows the programmer to define at runtime all the main features of the parallel algorithm, which affects the application performance, namely, the total number of participating parallel processes, the total volume of computations to be performed by each of the processes, the total volume of data to be transferred between each pair of the processes, and how exactly the processes interact during the execution of the algorithm. This information is used by the mpC compiler to find the optimal configuration of the applications minmizing its execution time. The disadvantage of mpC is that it is not easy to learn.

2.1.1.4 HeteroMPI

HeteroMPI [22] [23] is an extension of MPI for high-performance heterogeneous systems. It is based on the approach used in the mpC language and shares the same design principles. The MPI standard provides communicator and group constructs, which allow the programmer to create a group of processes that execute together some parallel computation or

communication operations of a parallel algorithm. However, the grouping of the processes is done without taking into account the heterogeneity of the target platform. Despite it is possible for the programmer to create optimal groups of processes, it is not a trivial task and would require a lot of complex code in order to measure the speeds of the processors and the performance of the communication links.

The HeteroMPI automates the selection of the optimal group of processes that would execute the heterogeneous algorithm faster than any other group.

2.1.1.5 Charm++

Charm++ [24] is a parallel object-oriented system based on the C++ programming language [25] and developed at the University of Illinois at Urbana-Champaign. Charm++ provides message-driven migratable objects which are supported by adaptive runtime system. Programs written in Charm++ are broken down into tasks called *chares*. The adaptive runtime system assigns chares to processors and the programmer can override the automatic mapping of chares. In addition, the runtime system provides some load balancing features. For example, if the runtime system detects that the distribution of the chares is not balanced among the processors, it can migrate the processes during the execution time to better balance the computation load. Charm++ has been used to implement Adaptive MPI (AMPI) [26], which came with some features that did not exist in the MPI-2 standard, such as asynchronous collectives. In addition, AMPI supports automatic load balancing and adaptive overlapping of computation and communication. MPI level malleability, the ability to shrink and grow the number of processes at runtime, and checkpointing can be achieved with additional code.

2.1.1.6 Partitioned Global Address Space

Partitioned Global Adress Space (PGAS) [27] is a parallel programming model where it is assumed that the global memory space is logically partitioned between processes. The abstraction of the shared address space

provides data locality and at the same time simplifies parallel programming. The PGAS model also offers affinity of the shared memory space to a particular process. A number of implementations of the PGAS model exist.

High Performance Fortran (HPF) [28] appeared in 1993 and is one of the earliest PGAS languages. It is based on Fortran 90 and designed for writing data parallel programs for shared and distributed memory platforms. HPF provides high level constructs/directives such as FORALL statements, data mapping directives, intrinsic and library functions, and extrinsic procedures. These constructs make it easy for the programmer to express the operations that supposed to be run in parallel. The most difficult part is done by the compiler, which analyzes the code and the constructs to generate the appropriate parallel code.

Co-array Fortran [29] is another PGAS language which is a parallel extension of Fortran 95 and introduces co-array as a new construct to the language. It provides a set of participating processes similar to MPI and the participating processes have access to local memory, while access to remote memory is supported by explicit language constructs. Later, the extensions were included in the Fortran 2008 [30] standard.

Unified Parallel C (UPC) [31] is a PGAS-like language, which is based on the ISO C 99 language standard and provides a number of extensions such as an explicit parallel execution model, communication and synchronization primitives, shared address space support. The last one is achieved by extending C arrays and pointers with their shared variants that maps into global memory. The Cascade High Productivity Language (CHAPEL) [32] is a parallel programming language developed by Cray. Chapel is a high-level language abstracting away many low-level parallel programming primitives and facilitates productivity. The main features of the language are multithreading, locality-awareness, object-orientation, and generic programming.

X10 [33] is another PGAS language being developed by IBM and supports both task and data parallelism. It is an object-oriented, strongly typed, and garbage-collected high-level language. The Java programming language has been used as a basis for the serial subset of X10.

2.2 Communication Performance Models

Having an analytical model for parallel communication plays an important role in predicting and tuning the performance of parallel applications. At the same time, there are two main issues associated with building such analytical models. First of all, it is difficult to design a parameterized analytical model itself. Second, accurate and efficient estimation of these parameters in a general way for different targeted platforms is quite complex. The overall process can be even more complicated if different factors, such as the heterogeneity of platforms, being taken into account. In this section we discuss the most commonly used communication models for homogeneous and heterogeneous platforms.

2.2.1 Homogeneous Communication Performance Models

The most common and comprehensive model for point-to-point communication is the Hockney model [10]. If α is the latency and β is the transfer time per byte, the inverse of which is called bandwidth, then the time T to transfer a message of size m between any two processor is given as follows:

$$T(m) = \alpha + m \times \beta \tag{2.1}$$

The parameters of the Hockney model can be estimated from point-to-point tests for different message sizes with the help of linear regression. There are also several benchmarking libraries and tools such as MPIBlib [34], NetPIPE [35] and others which can be used to estimate the parameters of the model.

The LogP model [36] is a more advanced model and consists of four parameters; L - an upper bound on the latency, o - the overhead which is defined as the length of time that a processor is involved in the transmission or reception of each message, g - the gap between messages and P - the number of processors. By using these parameters the LogP model estimates the time spent on a point-to-point communication as $L + 2 \times o$. The model assumes that the maximum number of messages that can be transmitted by

the network is $\left\lfloor \frac{L}{g} \right\rfloor$ and there is no contention in the network. An experimental study shows that this model can be accurate only for short messages [36] [37].

Later introduced LogGP model [38] extends the LogP model for large messages where the point-to-point time is estimated as $L + 2{\times}o + (m-1){\times}G$. The meaning of the parameters is the same as in the LogP model and the new G parameter represents the bandwidth for long messages. The authors use this model to analyse a number of algorithms including a single node scatter. The most important advantage of the LogGP model is that it can capture the overlap of communication and computation. A more recent extension of the LogP model is the PLogP [39] model, which differentiates the send and the receive overheads. Unlike the previous models this model is not linear and takes some parameters as a piecewise linear functions of message size. The gap is defined as $g(m)$ which is the reciprocal value of the end-to-end bandwidth between two processors for messages of a given size m. If $o_s(m)$ and $o_r(m)$ denote the send and receive overheads respectively then it is assumed that the gap covers the overheads: $g(m) \geq os(m)$ and $g(m) \geq or(m)$. The logp_mpi library [39] can be used to estimate the parameters of the LogP family models including PLogP.

Despite the LogGP model handles long messages it does not take the synchronization needed when sending long messages into account. This issue was addressed in the LogGPS [40] model. However, this model assumes a constant synchronization overhead independent of the message size. The LogGOPS model [41] overcomes this limitation.

2.2.2 Heterogeneous Communication Performance Models

The mentioned point-to-point performance models do not take heterogeneity of the target platforms into account. Therefore, prediction of collective communication operations using those models is not accurate enough on such platforms. On heterogeneous clusters we may have different parameters for each point-to-point communication. As a matter of fact, more accurate performance models have been designed with heterogeneity in

mind.

In [42] a parametrized model for clusters of clusters is given and called HiHCoHP. The model assumes that the processors are heterogeneous, the network interconnecting the processors inside a cluster is hierarchical, the clusters itself are hierarchical consisting of different size of clusters. The HiHCoHP model is based on several existing models, such as the LogP, BSP [43], the postal model [44] and few other models [45] [46] [47] [48] [49]. The authors use their model to predict broadcast and reduce operations.

The research work in [50] proposes a heterogeneous model for single-switched clusters. The model represents the communication time of sending a message of size m from a source processor i to a destination processor j as $C_i+m\times t_i+C_j+m\times t_j+\frac{m}{\beta_{ij}}$. Here, C_i and C_j are the fixed processing delays, t_i, t_j denote the delays of processing a byte, and β_{ij} is the transmission rate. This model reflects the heterogeneity of processors by introducing different fixed and variable delays. Later, the authors proposes a thechnique to accurately estimate the parameters of the model [51] [52].

2.2.3 Contention-Aware Communication Performance Models

Heterogeneity is not the only difficulty in building analytical communication models. There are more challenging issues if we take resource sharing on hierarchical networks into account. The mentioned models consider neither network resource sharing nor hierarchy in the communication.

The LoGPC [53] is yet another extension of the LogP and the LogGP models and tries to model the impact of network contention by analyzing k-ary n-cubes.

In [54] a performance model for many-to-one collective communications has been proposed. The model demonstrates that for medium size messages there is non-deterministic relationship between the execution time and the message size. According to the authors the reason for this behaviour is the resource contention on the root processor. The research work in [55] proposes contention-aware communication model for multi-core clusters

interconnected with an InfiniBand network. That work describes the bandwidth distribution at the NIC level and analyses dynamic network contention on flat star networks connected with a single switch. In [56] the authors introduce a contention-aware communication model for resource-constrained hierarchical Ethernet networks. This model employs asymmetric network property [57] [58] on TCP layer for concurrent bidirectional communications and the authors show that the symmetric network property (i.e. when the available bandwidth is shared equally between all concurrent communications) does not hold for complicated hierarchical networks.

2.2.3.1 Conclusion

Despite the Hockney model is the simplest among the outlined models, it is widely used to design algorithms in a general way on a variety of platforms. As a matter of fact, the model has been used to design and optimize collective communication [59], numerical linear algebra algorithms [60] [9] and many other scientific applications on distributed memory platforms. This model does not take any specific design and topology of the underlying system into account. Therefore, by using the Hockney model it is possible to design algorithms which are portable between different platforms. Moreover, it is believed that the algorithms designed with this model can also be useful in designing algorithms for specific topologies [61]. Thus, in this work we use the Hockney model to design and analyze topology-oblivious optimization of MPI collective operations.

2.3 Overview of MPI Collective Communication Operations

The Message Passing Interface provides two kinds of communication operations, point-to-point and collective operations. The point-to-point operations are expressed as a set of send and receive functions that allow transmitting a message of the specified size and type between two processes. On the contrary, the collective communication functions usually involve more than two processes and provide more abstraction of parallel processing than point-to-point operations. Some collective communication operations offer collective computation and synchronization features besides transmitting data among the processes. The high level of abstraction of the collectives makes it possible to express an appropriate problem in an elegant declarative way. This in turn improves their portability across different platforms while hiding the implementation details. Therefore it is surely preferable to use collectives rather than point-to-point operations where applicable. The collective operations in MPI can be classified as follows:

- Data movement functions - broadcast, scatter, scatterv, gather, gatherv, allgather, allgatherv, alltoall, and alltoallv.
- Collective computation functions - reduce, allreduce, reduce-scatter, scan and exscan.
- Synchronization functions - barrier.

The definitions of these functions in the MPI standard are given below:

- Broadcast:

```
int MPI_Bcast(void *buffer,int count, MPI_Datatype datatype,
    int root, MPI_Comm comm)
```

  This function broadcasts a message from a specified root process to all processes of a given group. In the rest of the text by process we mean MPI process which can be different from the operating system process.

- Scatter and Scatterv:

```
int MPI_Scatter(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, int root, MPI_Comm comm)
```

The MPI_Scatter function as the name suggests scatters the data specified in the buffer of the root process to all processes of a given group. During the scatter operation, i^{th} block of the data is transmitted to the member having rank i in the group.

The scatterv, also called irregular scatter operation, is similar to the scatter and is used to send a varying count of data to each process.

```
int MPI_Scatterv(const void *sendbuf, const int sendcounts[],
    const int displs[], MPI_Datatype sendtype, void *recvbuf,
    int recvcount, MPI_Datatype recvtype, int root, MPI_Comm
    comm)
```

The $sendcounts$ and $displs$ arrays control the varying counts of elements sent to each process and the relative displacements from $sendbuf$.

- Gather and Gatherv:

```
int MPI_Gather(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, int root, MPI_Comm comm)
```

This routine can be seen as a reverse scatter operation where data is gathered to the root process by the order of the ranks of all processes.

The irregular gather operation is defined as below:

```
int MPI_Gatherv(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, const int
    recvcounts[], const int displs[], MPI_Datatype recvtype,
    int root, MPI_Comm comm)
```

The $recvcounts$ array contains the counts of elements that are received from each process, $displs$ shows the displacements relative to the receive buffer where the received data should be placed into.

- Reduce:

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

In the reduce operation each process i owns a vector x_i of n elements. After completion of the operation all the vectors are reduced element-wise to a single n-element vector which is owned by a specified root process.

- Reduce-Scatter:

```
int MPI_Reduce_scatter(const void *sendbuf, void *recvbuf,
    const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
    MPI_Comm comm)
```

This operation can be seen as a reduce operation followed by scatterv operation.

- Allreduce:

```
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int
    count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Allreduce is very similar to the reduce operation except that all processes receive the same copy of the result vector.

- Scan:

```
int MPI_Scan(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Scan is used to perform an inclusive prefix reduction on data distributed across the calling processes.

- Exscan:

```
int MPI_Exscan(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

  Exscan operation is used to perform an exclusive prefix reduction on data distributed across the calling processes.

- Alltoall and Alltoallv:

```
int MPI_Alltoall(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm comm)
```

  Alltoall operation is used to transmit the same amount of data among all processes of the given group.

  The irregular alltoall operation, which allows all processes to send and receive different amount of data to and from all processes, is defined as follows:

```
int MPI_Allgatherv(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, const int
    recvcounts[], const int displs[], MPI_Datatype recvtype,
    MPI_Comm comm)
```

- Allgather and Allgatherv:

```
int MPI_Allgather(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm comm)
```

  This routine gathers specifed data by the order of process ranks to all processes.

  The irregular allgather operation is defined as follows:

```
int MPI_Allgatherv(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, const int
```

```
recvcounts[], const int displs[], MPI_Datatype recvtype,
MPI_Comm comm)
```

Allgatherv operation gathers a different count of data from each process to the root.

- Barrier:

```
int MPI_Barrier(MPI_Comm comm)
```

This routine blocks its caller until all processes in a given group have called it.

Recently, MPI-3 standard introduced nonblocking versions of all collective communication operations that gives the opportunity to prevent deadlocks, defer synchronization and overlap communication with computation. This work does not focus on the nonblocking collectives.

2.3.1 MPI Broadcast Operation

MPI broadcast is used in a variety of basic scientific applications and benchmarking libraries such as parallel matrix-vector and matrix-matrix multiplications, LU factorization, High-Performance Linpack [62] along with others. During a broadcast operation, the root process sends a message to all other processes in the specified group of processes.

In the rest of this paper, the amount of data to be broadcast and the number of MPI processes will be denoted by m and p respectively. It is assumed that the network is fully connected, bidirectional and homogeneous. The cost of sending a message of size m between any two processes is modeled by the Hockney's model [10] as $\alpha + m \times \beta$. Here α is the startup cost or latency, while β is the reciprocal bandwidth. The lower bound of the latency cost in the broadcast operation is $\lceil \alpha \times log_2 p \rceil$ as at each step of the broadcast the number of processes communicating can double at most. A message of size m should be transmitted at the end of the operation, which means that the lower bound of the bandwidth cost is equal to $\beta \times m$.

2.3.1.1 MPI Broadcast Implementation and Algorithms

The research by Johnson et al. [63] proposes n edge-disjoint spanning binomial trees (EDST), a balanced spanning tree and a graph consisting of n rotated spanning binomial trees for designing collective communication operations. Namely, the authors discuss how to design optimal one-to-all broadcasting, one-to-all personalized communication, all-to-all broadcasting, and all-to-all personalized communication operations on Boolean n cube architectures using the mentioned communication graphs. However, the main limitation of the proposed algorithms is that they can only be used for a power-of-two number of processes. Moreover, since these optimizations are designed for n-cube networks they result in high edge contention on meshes and other lower bandwidth networks [64].

The InterComm library [65] was one of the earliest libraries providing high-performance implementation of commonly used collective communication algorithms. The library was implemented for a

two-dimensional physical mesh of processes and the Hockney model was employed in the design of the algorithms. The implementation of the broadcast operation was presented as a minimum spanning tree scatter followed by an allgather operation. In addition, a comparison of the scatter/allgather algorithm with the theoretically optimal EDST algorithm was discussed. According to the authors, the EDST algorithm outperforms the scatter/allgather algorithm on hypercubes by a factor of two for large messages. However, it is also noted that the EDST algorithm is difficult to implement and lags behind the scatter/allgather algorithm when implemented on real systems.

The CCL library [61] designed for IBM SP1 machines is another early library implementing collective operations including broadcast. The library employs a slightly modified version of the Hockney model taking the effect of the congestion on the network into account. The authors discuss binomial tree, recursive doubling and the EDST based implementations of the broadcast operation. The MagPIe [3] library provides optimized collective algorithms for wide area cluster systems. The optimizations in the library are done by employing PLogP based analytical models for point-to-point communications. The main idea is to perform more communication over fast local-area links in order to minimize the communication over slower wide-area links. The PACX-MPI [66] library, which is an MPI implementation for heterogeneous metacomputing platforms, proposes optimized collective algorithms for clusters of massively parallel processing systems (MPP). The library is built on top of native MPI libraries, in the sense that the communication operations inside each MPP use the communication routines from the underlying MPI library, while the communication operations between MPPs are optimized by employing a specialized daemon nodes on each MPP. The daemon nodes acts like a gateway between MPPs. The library implements the broadcast operation in a hierarchical way, where the data is first broadcast between MPPs and then each MPP broadcasts locally. The algorithm is not optimal as it is assumed that only one message is sent to each machine. A library called ECO [67] offers optimized collective operations including broadcast for heterogeneous networks. Vadhiyar et

al. [68] [69] provide a methodology for modeling and automatic tuning of collectives on a given system by conducting a series of experiments. The authors compare their model with the MagPIe library and show that their approach provides better predictions for collective communication operations on unbalanced tree topologies.

The research work in [70] introduces another theoretically optimal broadcast algorithm based on fractional trees. The main idea of this algorithm is being able to transform from a pipelined binary tree to a linear pipeline algorithm when the message size increases. The work in [71] introduces a pipelined broadcast algorithm similar to the Johnson's algorithm when the number of processes is a power of two and extends it to an arbitrary number of processes.

The research work in [72] demonstrates a new broadcast algorithm for grid environments equipped with multi-lane network interface cards (NICs). Unlike previously proposed ones this algorithm can be used effectively both for small and large message sizes and can leverage multi-lane NICs without switching between different algorithms. The algorithm was validated on a simulator and compared with chain, binary and binomial algorithms.

The implementations of the broadcast operation in MPICH [73] and Open MPI [74] are typically based on linear, binary, binomial and pipelined algorithms [75]. The linear algorithms are not good for larger numbers of processes, the binary and binomial algorithms are not efficient for large data sizes. On the other hand, pipelined algorithms are more efficient for larger number of processes and data sizes. Other widely used broadcast algorithms are scatter-ring-allgather and scatter-recursive-doubling-allgather [65], which have been implemented in MPICH. In [76] the authors provide an experimental study of several broadcast algorithms, such as round-robin, chain, binomial, scatter-ring-allgather and pipeline algorithms. In addition, comparision with native MPICH broadcast implementation and possible improvements are discussed.

Another research direction in this area focuses on optimizing MPI broadcast for some specific platforms and topologies. The TMD-MPI [77] library provides an MPI implementation for clusters of embedded processors,

such as field-programmable gate arrays (FPGAs). The library implements MPI collective operations including broadcast. In [78] the researchers extend this work by adding hardware support for broadcast and reduce operations on Multiprocessor System-on-Chip platforms. The research works in [79] and [80] present efficient implementations of MPI broadcast, which use native Infiniband [81] multicast. It is further optimized for SMP clusters in [82]. In [83] optimization of broadcast over multidimensional process grids on multi-core platforms by means of optimal process to core mapping is presented. Teng Ma et al. [84] propose an adaptive collective communication framework, which allows one to build an optimal communication topology by using runtime process distance and underlying hardware architecture. The authors employ the Portable Hardware Locality (hwloc) [85] tool to build the framework and gather the required runtime information. MPI broadcast and allgather operations are used as example collectives. In [86] the implementation of topology-aware MPI collectives for symmetric multiprocessor systems in MPICH2 [87] is discussed. The authors use a two-level hierarchical design taking the topology of the system into account.

Cheetah [6] offers a hierarchical collective communication framework that takes advantage of the hardware-specific data-access mechanisms. IBM BlueGene comes with its own platform specific optimizations of MPI collectives [88]. A recent research work in [89] presents a generic framework for optimization of the performance of MPI collectives on Intel MIC clusters. Subramoni et al. [90] introduce topology and speed aware broadcast algorithms for InfiniBand Clusters. A comprehensive overview of the optimization techniques for collectives on heterogeneous HPC platforms using broadcast as a use case can be found in [91].

In [92] implementation issues of MPI collectives in the Cloud environments are discussed. Latency and bandwidth based network performance metrics are used to classify the closeness of virtual machines. Later this information is used to develop optimal MPI collective operations. A recent research [93] work presents a collective communication layer for data-intensive iterative MapReduce [94] applications. The authors introduce collective operations similar to the MPI collectives such as broadcast for

iterative MapReduce [95] applications.

2.3.1.2 Theoretical Analysis of MPI Broadcast Algorithms Implemented in MPICH and/or Open MPI

This section overviews eight MPI broadcast algorithms implemented in MPICH and Open MPI, namely flat/linear, chain, pipeline/pipelined linear algorithm, binary, split-binary, binomial tree, scatter-ring-allgather, and scatter-recursive-doubling-allgather broadcast algorithms. The first six algorithms are implemented in Open MPI and the last three algorithms are implemented in MPICH.

- Flat tree broadcast algorithm.
  Flat tree is the simplest MPI broadcast algorithm, in which the root process sequentially sends the same message to all the processes participating in the broadcast operation. This algorithm does not scale well for large communicators. Its cost is estimated as below:

$$(p-1)\times(\alpha+m\times\beta) \tag{2.2}$$

- Chain broadcast algorithm.
  In this algorithm each process sends or receives at most one message. The root does not receive any message and the last node (process p) does not send any message. Theoretically, its cost is the same as that of the flat tree algorithm:

$$(p-1)\times(\alpha+m\times\beta) \tag{2.3}$$

- Pipelined linear tree broadcast algorithm.
  The performance of the chain algorithm can be improved by splitting and pipelining the message. The run time of the algorithm is the following:

$$(X+p-2)\times(\alpha+\frac{m}{X}\times\beta) \tag{2.4}$$

Here, it is assumed that a broadcast message of size m is split into X segments so that segments of size $\frac{m}{X}$ are communicated between processes.

- Binary tree broadcast algorithm.
  Consider a full and complete binary tree. Let $T(h)$ denote the run time of the broadcast at the height h in this tree. Then, the run time of the broadcast on a single node is zero, $T(0) = 0$. The last node receiving the message at height h will send two messages to its children at height $h + 1$, therefore $T(h + 1) = T(h) + 2(\alpha + m\beta)$. It can easily be shown that $T(h) = 2\times h\times(\alpha + m\beta)$ and the number of nodes of the binary tree is $2^{h+1} - 1$, therefore the overall run time will be equal to the following function:
  $$2\times(\log_2(p + 1) - 1)\times(\alpha + m\times\beta) \tag{2.5}$$

- Split-binary tree broadcast algorithm.
  The split-binary tree algorithm [96] consists of forwarding and exchange phases. In the forwarding phase, the root process splits the original message of size m in half, then each of the halves is broadcast down the left and right subtrees respectively using the binary tree algorithm. In the exchange phase, each process in both subtrees will exchange their message of size $\frac{m}{2}$ with the corresponding pairing process from the other subtree. The time to complete a broadcast operation with this algorithm is equal to the sum of the times spent in the forwarding and exchange phases. Thus, if we assume a full-duplex network, then its cost on a full and complete binary tree, where the number of processes is one less than an exact power of two, will be as follows:
  $$2\times(\log_2(p + 1) - 2)\times(\alpha + \frac{m}{2}\times\beta) + \alpha + \frac{m}{2}\times\beta \tag{2.6}$$

- Binomial tree broadcast algorithm.
  Consider a binomial tree of height h with the number of nodes equal to 2^h. The number of nodes sending and receiving will double with each increment of the height in the algorithm. Thus, the run time $T(h)$ will

change as follows: $T(0) = 0$, $T(1) = \alpha + m\beta$, $T(2) = 2(\alpha + m\beta)$, ... $T(h) = h(\alpha + m\beta)$. For $p = 2^h$ the run time of the algorithm will be

$$\log_2 p \times (\alpha + m \times \beta) \tag{2.7}$$

- Scatter-ring-allgather broadcast algorithm.
  The algorithm consists of scatter and allgather phases. The message is scattered by a binomial tree algorithm in the first phase, and in the next phase a ring algorithm for allgather is used to collect all segments from all processes. The time taken by the scatter phase is $\log_2 p \times \alpha + \frac{p-1}{p} \times m \times \beta$ and the ring phase takes $(p-1) \times \alpha + + \frac{p-1}{p} \times m \times \beta$. Thus, the total cost of this algorithm will be as below:

$$(\log_2 p + p - 1) \times \alpha + 2 \times \frac{p-1}{p} \times m \times \beta \tag{2.8}$$

  The algorithm is used in MPICH for large messages.

- Scatter-recursive-doubling-allgather broadcast algorithm.
  This algorithm is very similar to the scatter-ring-allgather algorithm except the allgather phase uses a recursive doubling algorithm. In each step of the recursive doubling algorithm the distance between the communicating processes is doubled. Figure 2.1 illustrates the concept in a more clear way. For a power-of-two number of processes, p, the algorithm consists of $\log_2 p$ steps. The amount of data exchanged by each process doubles in each step, so that in the first step it will be $\frac{n}{p}$, in the second step it will be $2 \times \frac{n}{p}$, and it continues this way until in the last step the amount of data exchanged becomes $\frac{2^{\log_2 p - 1} \times n}{p}$. If the number of processes is not a power-of-two then at each step of the algorithm additional communication is performed for each set of non-power-of-two number of processes with their power-of-two number of peer set. This additional step takes logarithmic time which means that in the case of non-power-of-two processes the algorithm is bounded by $2 \times \lfloor \log p \rfloor$. This algorithm is used in MPICH for medium-size messages. A research work shows [59] that the ring algorithm is more efficient than

recursive doubling for large messages because of its nearest-neighbor communication pattern. The time taken by this algorithm will be as follows:

$$2\times \log_2 p\times\alpha + 2\times\frac{p-1}{p}\times m\times\beta \tag{2.9}$$

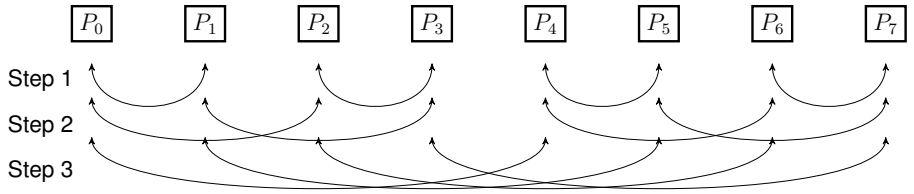


Figure 2.1: Communication Pattern in Recursive Doubling Allgather

2.3.2 MPI Reduction Operations

Reduce is an important and commonly used collective operation in the Message Passing Interface (MPI) [16]. According to a research study [97], MPI reduction operations, particularly MPI reduce and allreduce, are the most used collective operations in scientific applications. We have already briefly outlined MPI reduction operations in Section 2.3. In this section, we focus on MPI reduce and allreduce operations and discuss the state-of-the-art algorithms and optimizations for those two operations. In the reduce operation, each node i owns a vector x_i of n elements. After completion of the operation, all the vectors are reduced element-wise to a single n-element vector, which will be owned by a specified root process. In the allreduce operation, the result vector will be accumulated on all the processes in the same way as it happens in the reduce. Another widely used reduction operation is reduce-scatter, which can be seen as a reduce operation followed by a scatter operation.

The reduce operation was implemented as an inverse broadcast in the CCL [61] library and was not optimized for different message sizes. The InterComm [65] library proposes a more advanced reduce implementation as a combination of reduce-scatter and gather, and tries to be efficient for small and large message sizes. The allreduce operation was implemented in a similar way as a reduce-scatter followed by an allgather operation. The MagPIe [98] library provides optimized collective algorithms, including algorithms for reduction operations for wide area systems. The PACX-MPI [66] library provides an MPI implementation for computational grids, which may consist of interconnected clusters of high-performance computers such as massively parallel processing systems (MPP). The library implements the reduce operation similar to the broadcast operation as discussed in the previous section. In the first phase, the operation is performed locally inside each MPP and then the locally reduced data is further reduced globally among MPPs. The research work in [99] proposes two algorithms for the reduce-scatter operation, designed in the LogGP model. The first algorithm is called RS_1 and designed for associative and

commutative reduction operators. The author shows that this algorithm is optimal within a small constant factor. The second algorithm, RS_2, is not optimal but simpler and can be used with a non-commutative operations. Moreover, the second algorithm can be very efficient in practice for large messages. Later, in [100] the authors discuss implementation issues of the RS_1 algorithm. Patarasuk et al. [101] propose an efficient allreduce algorithm for large messages on tree topologies.

Design and high-performance implementation of collective communication operations and commonly used algorithms, such as minimum-spanning tree reduce algorithm, are discussed in [102]. The authors also discuss the lower bounds of the reduction operations. The lower bounds on the latency and computation costs are the same in both the reduce and allreduce operations, and equal to $\lceil log_2(p) \rceil$ and $\frac{p-1}{p} \times n \times \gamma$ respectively. On the other hand, the lower bound on the bandwidth cost is $n \times \beta$ for the reduce, while it is $2 \times \frac{p-1}{p} \times n \times \beta$ for the allreduce operation.

Automatic tuning of collectives for a given system by conducting a series of experiments on the system was discussed in [69]. Rabenseifner [103] proposes five reduction algorithms optimized for different message sizes and numbers of processes. Implementations of MPI collectives in MPICH, including reduction, are discussed in [59].

Algorithms for MPI broadcast, reduce and scatter, where the communication happens concurrently over two binary trees, are presented in [64]. Cheetah framework [6] implements MPI reduction operations in a hierarchical way on multicore systems, which supports multiple communication mechanisms. Unlike that work, our optimization is topology-oblivious, and for MPI reduce optimizations in our work we do not design new algorithms from scratch, employing the existing reduce algorithms underneath. Therefore, our hierarchical design can be built on top of the algorithms from the Cheetah framework as well.

In the next two sections, we consider reduce and allreduce algorithms implemented in MPICH and Open MPI into our analysis.

2.3.2.1 Theoretical Analysis of MPI Reduce Algorithms Implemented in MPICH and/or Open MPI

- Flat tree reduce algorithm.
  In this algorithm, the root process sequentially receives and reduces a message of size m from all the processes participating in the reduce operation in $p-1$ steps:

$$(p-1) \times (\alpha + m \times \beta + m \times \gamma) \tag{2.10}$$

  In a segmented variation of the flat tree algorithm, a message of size m is split into X segments, in which case the number of steps becomes $X \times (p-1)$. Thus, the total execution time will be as follows:

$$X \times (p-1) \times \left(\alpha + \frac{m}{X} \times \beta + \frac{m}{X} \times \gamma\right) \tag{2.11}$$

  The flat tree reduce algorithm causes a bottleneck on the root process.

- Linear tree reduce algorithm.
  Unlike the flat tree algorithm, in the linear tree each process receives or sends at most one message. Theoretically, its cost is the same as for the flat tree algorithm:

$$(p-1) \times (\alpha + m \times \beta + m \times \gamma) \tag{2.12}$$

- Pipeline reduce algorithm.
  In the pipeline reduce algorithm, a message of size m is split into X segments and in one step of the algorithm segments of size $\frac{m}{X}$ are communicated between p processes. If we assume a logically reversed linear array, in the first step of the algorithm the first segment of the message is sent to the next process in the array. Next, while the second process sends the first segment to the third process, the first process sends the second segment to the second process, and the algorithm continues this way. The first segment takes $p-1$ steps and the remaining segments take $X-1$ steps to reach the end of the array. If

we also consider the computation cost in each step, then the overall execution time of the algorithm will be as follows:

$$(p + X - 2) \times \left(\alpha + \frac{m}{X} \times \beta + \frac{m}{X} \times \gamma\right) \tag{2.13}$$

- Binary tree reduce algorithms.
  By employing a divide and conquer strategy in tree based algorithms, one can improve the performance of the simple linear algorithms. As a matter of fact, a binary tree reduce algorithm eliminates the bottleneck on the root process. If we take a full and complete binary tree of height h, its number of nodes will be $2^{h+1} - 1$. In the reduce operation, a node at the hight h will receive two messages from its children at the height $h + 1$. In addition, if we divide a message of size m into X segments, the overall run time will be as follows:

$$2 \times (\log_2 (p + 1) + X - 2) \times \left(\alpha + \frac{m}{X} \times \beta + \frac{m}{X} \times \gamma\right) \tag{2.14}$$

  Open MPI uses an in-order binary tree algorithm for non-commutative operations. It works similarly to the binary tree algorithm but enforces order in the operations. Binary tree algorithms are efficient for small messages where the overall communication time is dominated by the latency term. However, these algorithms still suffer from load balancing issues because of contention on the internal nodes. This issue can be avoided by using a binomial tree data structure.

- Binomial tree reduce algorithm.
  The binomial tree algorithm takes $\log_2 p$ steps, communicating messages of size m at each step. If each message is divided into X segments, then the number of steps and the message size at each step will be $X \times \log_2 p$ and $\frac{m}{X}$ respectively. Therefore, the overall run time will be as follows:

$$\log_2 p \times (\alpha + m \times \beta + m \times \gamma) \tag{2.15}$$

- Rabenseifner's reduce algorithm.

The Rabenseifner's algorithm [103] is designed for large messages. The algorithm consists of reduce-scatter and gather phases. It has been implemented in MPICH [59] and used for message sizes greater than 2KB. The reduce-scatter phase is implemented with recursive-halving, and the gather phase is implemented with binomial tree. Therefore, the cost of the algorithm is the sum of the costs of these two phases:

$$2\times \log_2 p\times\alpha + 2\times\frac{p-1}{p}\times m\times\beta + \frac{p-1}{p}\times m\times\gamma \qquad (2.16)$$

The algorithm can be further optimized by recursive halving and distance doubling. Despite the recursive algorithms naturally fit to a power-of-two number of processes, they can be modified to support a non-power-of-two number of processes as well. One such technique is to reduce the number of processes to the nearest lower power-of-two ($p_{new} = 2^{\lceil \log_2 p \rceil}$) number. It can be achieved by removing the remaining $r = p - p_{new}$ processes. After that, all the even ranks in the first $2\times r$ processes send their second half of the data to their right neighbour, while all the odd ranks send the first half of the data to their left neighbour. Then, the even and odd ranks perform reduction on the first and second half of the data respectively. In the next step, the odd ranks send their reduced results to their left neighbours and do not participate in the rest of the algorithm. All the remaining processes form a power-of-two number and continue the reduction operation altogether and finally send their results back to the odd ranks. However, the recursive doubling and halving can result in load imbalance for a non-power-of-two number of processes. Two approaches are used to deal with that case: binary block and ring algorithms. Before discussing those optimizations, we put together the terminology used by the author:

- *Recursive vector halving*: The input buffer is split into two halves, then one half is reduced by the process itself, and the other half is sent to a neighbour process to be reduced. The process continues in the same way recursively.

- *Recursive vector doubling*: Scattered parts of the buffer recursiveley gathered into a larger result vector.
- *Recursive distance doubling*: The distance between the communicating processes is doubled in each step (Figure 2.1).
- *Recursive distance halving*: The distance between the communicating processes is halved in each step.

In the binary block algorithm, the number of processes is transformed to a sum of power-of-two numbers which are called blocks. Then each block performs their own reduction operation. The ring algorithm for reduce uses $p-1$ ring exchanges in the reduce-scatter phase and after that each process sends its result to the root. The allreduce implementation is similar to the reduce except the gather phase uses a ring exchange as well.

2.3.2.2 Theoretical Analysis of MPI Allreduce Algorithms Implemented in MPICH and/or Open MPI

The MPICH implementation employs two algorithms for the allreduce operation, a recursive-doubling used for short messages and long messages with user-defined reduction operations, and Rabensifner's algorithm used for long messages and native MPI reduction operations. On the other hand, the Open MPI uses a recursive-doubling algorithm for short messages. In the case of long messages and commutative operations, it uses linear and segmented-ring allreduce algorithms. For non-commutative reduction operations and long messages, a simple reduce followed by broadcast algorithm is used.

- Linear allreduce algorithm
  Open MPI implements the linear allreduce algorithm as a linear reduce to a specified root followed by a linear broadcast from the same root. Despite the root process faces the communication and computation overhead, the linear algorithm can be a preferred algorithm for small messages on a small number of processes. The time for the allreduce

will be the sum of the linear reduce (formula 2.17) and linear broadcast (formula 2.3) times:

$$2\times(p-1)\times(\alpha+m\times\beta)+(p-1)\times m\times\gamma \tag{2.17}$$

- Recursive doubling allreduce algorithm
  The recursive doubling allreduce algorithm is similar to the allgather phase of the scatter-allgather broadcast algorithm (see Section 2.3.1.2). In each step of the algorithm, the distance between the communicating processes is doubled. For a power-of-two number of processes, the algorithm consists of $\log_2 p$ steps. The amount of data exchanged by each process doubles in each step. In the first step it is $\frac{m}{p}$, in the second step it is $2\times\frac{m}{p}$, and it continues this way until in the last step the amount of data exchanged becomes $\frac{2^{\log_2 p-1}\times m}{p}$. Therefore, the total execution time of the algorithm will be as follows:

$$\log_2 p\times(\alpha+m\times\beta+m\times\gamma) \tag{2.18}$$

- Rabensifner's allreduce algorithm
  Rabensifner's allreduce algorithm consists of a reduce-scatter followed by an allgather. The cost of the algorithm will be equal to the sum of the costs of the reduce-scatter ($\log_2 p\times\alpha+\frac{p-1}{p}\times m\times\beta+m\times\log_2 p\times\gamma$) and the allgather part ($\log_2 p\times\alpha+\frac{p-1}{p}\times m\times\beta$), which will result in the following total cost:

$$2\times\log_2 p\times\alpha+2\times\frac{p-1}{p}\times m\times\beta+\frac{p-1}{p}\times m\times\gamma \tag{2.19}$$

  The main limitation of this algorithm is that it can not be applied to a user defined operations.

- Ring allreduce algorithm
  The ring algorithm for allreduce uses a nearest-neighbour communication pattern and is used for commutative operations. The algorithm has computation and distribution phases. The send buffer is

divided into p blocks of size $\frac{send\_count}{p}$. The algorithm can be quite easily modified to support a use case where p does not divide the send count [96]. In each iteration i of the computation phase, rank $(r-1+p)\%p$ sends block $(r-i+p)\%p$ to rank r, which in turn receives the data using a non-blocking receive and performs the reduction operation on the block before sending the result to rank $(r+1)\%p$. In the data distribution phase, rank r receives the reduced data from its left neighbour and sends it to its right neighbour. The algorithm continues this way in $2\times p-1$ iterations. Its total cost will be the following:

$$2\times(p-1)\times(\alpha+\lceil\frac{m}{p}\rceil\times\beta)+(p-1)\frac{m}{p}\times\gamma \tag{2.20}$$

The algorithm assumes that $send\_count > p$.

- Segmented ring allreduce algorithm
  In the segmented ring allreduce algorithm, all blocks is divided into segments of X size. Then the computation phase is performed in a block-cyclic way for each of the segment groups. The distribution phase is executed similarly to that of the non-segmented ring algorithm. The main limitation of the algorithm is that it can be applied only if the send count is greater than $p\times\frac{block\_size}{X}$. The cost of the algorithm is given below:

$$(p+X-2)\times(\alpha+\frac{m}{X}\times\beta+\frac{m}{X}\times\gamma)+(p-1)\times(\alpha+\lceil\frac{m}{p}\rceil\times\beta) \tag{2.21}$$

2.3.3 MPI Scatter and Gather Operations

At the beginning of the scatter operation, the root process owns a vector $x = (x_0, x_1, ..., x_n)$ of length n, where each element has the same type and the same length. After successful execution of the operation, process with rank i receives element x_i.

The gather can be seen as a reverse scatter operation, initially each element x_i owned by process i and after successful execution of the operation the root process owns the entire x vector. In the same way as in scatter, each element either can be a single element or a subvector of the original vector with all elements being of the same type and the same length. These two operations are also called regular scatter and gather in the Message Passing Interface(MPI) [16]. The MPI specification defines irregular scatter and gather operations as well. The main difference between the regular and irregular operations is that in the latter case the size of the data to be scattered/gathered from/to the root does not have to be the same.

It has been shown that (Chan et al. [102]) the lower bounds of the scatter and gather operations are $\lceil \log_2 p \rceil \times \alpha$ and $\frac{p-1}{p} \times n \times \beta$ for latency and bandwidth respectively. The authors present a minium spanning tree (MST) algorithm for the scatter and gather operations. In [104] three heuristic scheduling algorithms for a gather operation on heterogeneous platforms are introduced. According to the authors, binomial gather algorithms can be worse than sequential (based on flat trees) gather algorithms on distributed heterogeneous systems and their heuristic algorithms show better results than the sequential algorithms. The performance of the algorithms validated using simulations. The research work in [105] presents topology-aware design of MPI collectives, employing scatter and gather as a use case on large scale InfiniBand clusters. Träff proposes [106] binomial-tree based scatter/gather algorithms and discusses optimization of these algorithms on hierarchical SMP-clusters. The algorithms require a fixed small amount of intermediate buffer where the messages are scattered/gathered in a tree-like way. If the message size exceeds the intermediate buffer the algorithms continue in a serial way in which the root process scatters/gathers data

to/from one process in a sequence. The author extends these algorithms to irregular scatter and gather operations as well. Later in [107] two model-based irregular scatter and gather algorithms proposed for heterogeneous platforms. The algorithms modify the binomial and Träff's algorithms by using heterogeneous communication performance models.

2.3.3.1 Theoretical Analysis of MPI Scatter and Gather Algorithms Implemented in MPICH and/or Open MPI

The state-of-the-art MPI implementations of the scatter and gather operations are based on linear and binomial tree algorithms. In the linear scatter/gather algorithm, the root process sends/receives corresponding messages to/from the other $p - 1$ processes using non-blocking send operations with a communicator of size p. In case of linear gather operation, the root process may get overloaded when the message or the communicator size is large. To overcome this potential issue, a variation of the linear algorithm with synchronization implemented in Open MPI. The algorithm can be described in the following basic steps [96]:

- All non-root processes:
  - Receive a zero byte message from the root process.
  - Divide the message into two segments.
  - Send the first and the second segments of the message synchronously.
- The root process:
  - Posts an asynchronous receive for the first segment of the message.
  - Then sends a zero byte message to all the non-root processes.
  - After that, posts an asynchronize receive for the second segment of the message and waits for the first segment to complete.
  - Finally, copies local data if necessary and waits for the second segment to complete.

The binomial algorithm in Open MPI uses an in-order binomial tree. The total bandwidth cost of the operation increases if the binomial tree is used, but the latency cost reduces as there will be a smaller number of steps in comparison with the linear algorithm. Therefore, the binomial algorithm is used for small messages. On the other hand, the linear algorithm is better suited for large messages.

The scatter and gather operations in MPICH use a binomial tree algorithm for both short and long messages when used with intra-communicators. In the case of inter-communicators, a minimum spanning tree algorithm is used for short messages, and a linear algorithm is used for long messages. On the other hand, Open MPI employs a binomial tree algorithm for communicators of size larger than 10 and message sizes less than 300B. In all other cases it uses a linear scatter algorithm. The selection of the gather algorithms in Open MPI is more complex. Namely, if the message size is greater than 92KB then the linear with synchronization gather algorithm with 32KB segment size is used, else if the block size is greater than 6KB then the same algorithm with 1KB segment size is used. Otherwise, when the communicator size is greater than 60 or for communicator sizes greater than 10 and messages less than 1KB, a binomial gather algorithm will be used. In all other cases, a basic linear algorithm is used. Theoretical costs of all these algorithms within the Hockney model are given below:

- Linear Scatter and Gather Algorithms
  The root process in the linear algorithm for scatter/gather sends/receives equal amount of data (sub-vectors) to/from all processes in each step. Assuming that the total amount of data in the scatter/gather operation is $p \times m$ the cost of the scatter/gather operation can be derived as below:

$$(p-1)\times(\alpha + m\times\beta) \tag{2.22}$$

- Linear with Syncronization Gather Algorithm
  The root process can send a zero byte message in α time. Let us assume that after dividing a message of size m into two parts the size of the first part will be X. Then the cost of this part will be $\alpha + X\times\beta$,

while the cost of the second part will be equal to $\alpha + (m - X)\times\beta$. In the case of full-duplex network, the overall cost will be as follows:

$$\alpha + (p-1)\times(2\times\alpha + m\times\beta) \tag{2.23}$$

- Binomial Scatter and Gather Algorithms
  We only discuss the binomial tree gather algorithm as the algorithm for the scatter operation has the similar design. In the binomial gather, the leaf nodes send their data to their parent processes. Intermediate nodes forwards data up the tree after they receive data from all their children. Upon receipt of all data, the root node might need to perform local shift operation to put the data in correct order. The implementation of the algorithm in Open MPI uses in-order binomial tree topology [96]. Thus, the cost will be
  $$\log_2 p\times\alpha + (p-1)\times m\times\beta \tag{2.24}$$

- Minium-Spanning Tree Scatter and Gather Algorithms
  In the minimum-spanning tree scatter algorithm, the processes are divided into two groups. The root process reside in the first group and it communicates with a process from the second group. During this communication, the root process transmits the part of the data that is ultimately should reside in the second group. Upon the receipt of the data, the process from the second group behaves as the root of its own group and the algorithm continues recursively and in parallel in the two separate groups. If we assume that the number of processes is a power-of-two and all process own equal amounts of data, then the final cost of the MST scatter will be as follows:
  $$\log_2 p\times\alpha + \frac{p-1}{p}\times m\times\beta \tag{2.25}$$

We have also mentioned that the MST algorithm achieves the lower bounds on both the latency and the bandwidth costs of the scatter operation [102].

The MST gather algorithm can be seen as MST scatter algorithm in reverse order. Therefore their costs are equal.

It is worth mentioning that these theoretical analysis is not the original contribution of this thesis and an interested reader is referred to the corresponding references [102], [96]. The main goal of this part is establishing the fundamental background for our introduction to the hierarchical optimizations of the MPI collective operations in the next chapter.

2.3.4 Conclusion

Despite there has been a lot of research in the optimization of MPI collective operations, most of the recent works are not general purpose and very often focus on some specific toplogy and platforms. At the same time, the state-of-the-art MPI implementations provide a bunch of algorithms where a decision function switches between the algorithms depending on the number of processes, the message size and the reduce operation (for reduction operations). However, the switching points are based mainly on measurements which obtained on specific platforms. This in turn means that, the decision function is not general and the selected algorithms can be far from optimal.

With these issues in mind, we propose topology and platform oblivious optimization of MPI collective operations. The main idea is not introducing new algorithms from scratch but rather building a hierarchical optimization tool on top of the existing algorithms. The approach lets us use the existing algorithms underneath and improve their performance further. At the same time, our hierarchical algorithms can fall back to the original algorithms if their performance can not be improved. The next section presents the hierarchical optimization of MPI collective algorithms.

Chapter 3

Hierarchical Optimization of MPI Collective Operations

3.1 Hierarchical Transformation of MPI Broadcast Algorithms

This section introduces a simple but general optimization of the MPI broadcast algorithms. The proposed optimization is based on the hierarchical arrangement of the processes, participating in the broadcast, into logical groups. Let us denote by p the total number of MPI processes. For simplicity we assume that the number of groups divides the number of MPI processes and can change between one and p. Let G be the number of groups. Then there will be $\frac{p}{G}$ MPI processes per group. Figure 3.1 shows an arrangement of 12 processes in a non-hierarchical way and a hierarchical grouping of 12 processes into 3 groups of 4 processes. The hierarchical optimization has two steps: in the first step a group leader is selected for each group and the broadcast is performed between the group leaders (see Figure 3.2), and in the next step the leaders start broadcasting inside their own group (in this example between 4 processes). The grouping can be done by taking the topology into account as well. However, this work focuses on the case where the grouping is topology-oblivious and the first process in each group is selected as the group leader. The broadcasts inside different groups are

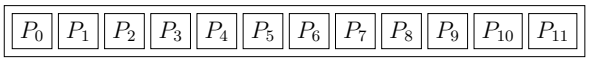


Figure 3.1: Arrangement of processes in MPI broadcast.

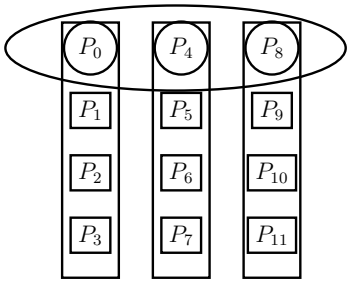


Figure 3.2: Arrangement of processes in the hierarchical broadcast. Processes in the ellipses are the group leaders. The rectangles show the processes inside groups. In the first step the broadcast is performed between the group leaders and in the next step it is performed among the processes inside each group.

executed in parallel. While in general different broadcast algorithms can be used inside and between groups, this work focuses on the case where the same broadcast algorithm is employed in both levels. Algorithm 1 shows the pseudocode of the hierarchical broadcast; Line 4 calculates the root for the broadcast inside the groups. Then line 5 creates a sub-communicator of G processes among the groups and line 6 creates a sub-communicator of $\frac{p}{G}$ processes inside the groups. Our implementation uses MPI_Comm_split MPI routine to create new sub-communicators.

3.1.1 Theoretical Analysis

3.1.1.1 Hierarchical Flat and Linear Tree Broadcast

If we group the processes in the hierarchical way and apply the flat or linear tree broadcast algorithm among G groups and inside the groups among $\frac{p}{G}$ processes then the overall broadcast cost will be equal to their sum:

$$F(G) = (G-1)\times(\alpha+m\times\beta)+(\frac{p}{G}-1)\times(\alpha+m\times\beta) = (G+\frac{p}{G}-2)\times(\alpha+m\times\beta) \tag{3.1}$$

Data: p - Number of processes
Data: G - Number of groups
Data: buf - Message buffer
Data: $count$ - Number of entries in buffer (integer)
Data: $datatype$ - Data type of buffer
Data: $root$ - Rank of broadcast root
Data: $comm$ - MPI Communicator
Result: All the processes have the message of size m
begin

```
1  MPI_Comm comm_outer        /* communicator among the groups */
2  MPI_Comm comm_inner       /* communicator inside the groups */
3  int root_inner          /* root of broadcast inside the groups */
4  root_inner = Calculate_Root_Inner(G, p, root, comm)
5  comm_outer = Create_Comm_Between_Groups(G, p, root, comm)
6  comm_inner = Create_Comm_Inside_Groups(G, p, root_inner, comm)
7  MPI_Bcast(buf, count, datatype, root, comm_outer)
8  MPI_Bcast(buf, count, datatype, root_inner, comm_inner)
```

end

Algorithm 1: Hierarchical transformation of an MPI broadcast algorithm.

Here $F(G)$ is a function of G for a fixed p. Its derivative is equal to $(1 - \frac{p}{G^2}) \times (\alpha + m \times \beta)$. It can be shown that $G = \sqrt{p}$ is the minimum of the function $F(G)$ as in the interval $(1, \sqrt{p})$ the function decreases, and in the interval $(\sqrt{p}, p)$ it increases. If we take $G = \sqrt{p}$ in the $F(G)$ function the optimal value of the broadcast cost will be as follows:

$$F(\sqrt{p}) = 2 \times (\sqrt{p} - 1) \times (\alpha + m \times \beta) \tag{3.2}$$

3.1.1.2 Hierarchical Pipelined Linear Tree Broadcast

In the same way, if we add two pipelined linear tree broadcast costs among G groups and inside the groups among $\frac{p}{G}$ processes then the overall communication cost for the hierarchical pipelined linear tree will be as follows:

$$F(G) = (2 \times X + G + \frac{p}{G} - 4) \times (\alpha + \frac{m}{X} \times \beta) \tag{3.3}$$

It can be shown that $G = \sqrt{p}$ is the minimum point again and at this point the cost will be as follows:

$$F(\sqrt{p}) = (2{\times}X + 2{\times}\sqrt{p} - 4){\times}(\alpha + \frac{m}{X}{\times}\beta) \tag{3.4}$$

3.1.1.3 Hierarchical Binary and Binomial Tree Broadcast

Let us apply the binary tree algorithm among G groups and inside the groups among $\frac{p}{G}$ processes. The broadcast cost among G groups and inside the groups will be $2{\times}\log_2 G{\times}(\alpha + m{\times}\beta)$ and $2{\times}\log_2 \frac{p}{G}{\times}(\alpha + m{\times}\beta)$ respectively. If we calculate the sum of these two costs then the overall cost of the hierarchical binary broadcast algorithm will be the same as the corresponding non-hierarchical broadcast algorithm.

Because of the same reason the hierarchical modification of the binomial tree algorithm does not improve the non-hierarchical binomial tree algorithm. It is worth mentioning that while we claim the cost of the hierarchical and the corresponding original binary and binomial algorithms are the same we assume that the overhead to create the sub-communicators is negligible or it can be avoided by the implementation where the creation of the sub-communicators can be done during MPI initialisation time. We experimentally show that the first assumption holds for medium and large message sizes. Otherwise, the implementation can fall back to the original algorithm by using the number of groups equal to one.

3.1.1.4 Hierarchical Scatter-Ring-Allgather Broadcast

The sum of the costs of two scatter-ring-allgather algorithms inside and outside the groups will give us the following formula:

$$F(G) = \left(\log_2 p + G + \frac{p}{G} - 2\right){\times}\alpha + 2{\times}m{\times}\left(2 - \frac{1}{G} - \frac{G}{p}\right){\times}\beta \tag{3.5}$$

Let us find the optimal value of the $F(G)$ function: $F'(G) = \frac{g^2-p}{G^2} \times \left(\alpha - \frac{2 \times m \times \beta}{p}\right)$. It is clear that if

$$\frac{\alpha}{\beta} > \frac{2 \times m}{p} \tag{3.6}$$

then $G = \sqrt{p}$ is the minimum point of the $F(G)$ function in the interval $(1, p)$. The value of the function at this point will be as follows:

$$F(\sqrt{p}) = (\log_2 p + 2 \times \sqrt{p} - 2) \times \alpha + 2 \times m \times \left(2 - \frac{2}{\sqrt{p}}\right) \times \beta \tag{3.7}$$

3.1.1.5 Hierarchical Scatter-Recursive-Doubling-Allgather Broadcast

The hierarchical modification of this algorithm has a higher theoretical cost compared to the cost of the original algorithm. The latency term is increased two times and the bandwidth term is increased as well:

$$F(G) = 2 \times \log_2 p \times \alpha + 2 \times m \times \left(2 - \frac{1}{G} - \frac{G}{p}\right) \times \beta \tag{3.8}$$

It can be shown that $G = \sqrt{p}$ is the extremum point and the function achieves its maximum at this point. Therefore for this algorithm a hierarchical implementation of broadcast should use $G = 1$ in which case the algorithm will fall back to the original algorithm.

3.1.1.6 Hierarchical Split-Binary Tree Broadcast

The cost of the hierarchical transformation of the split-binary broadcast can be derived by summing up the broadcast cost inside each group among $\frac{p}{G}$ processes and that of outside the groups amoung G processes. Thus, according to formula 2.6 the cost function will be as follows:

$$2 \times (\log_2 (p + G) - 4) \times (\alpha + \beta \times \frac{m}{2}) + 2 \times \left(\alpha + \frac{m}{2} \times \beta\right) \tag{3.9}$$

3.1.1.7 Summary of Theoretical Analysis

We can summarise this section by saying that the hierarchical transformations of the flat, chain, pipeline and scatter-ring-allgather algorithms theoretically reduce the communication cost of the corresponding underlying algorithms. The communication costs of the binary, binomial, scatter-recursive-doubling-allgather and split-binary tree algorithms get their best performance when the number of groups is one or equal to the number of processes.

3.1.2 Experimental Study

3.1.2.1 Experiments on BlueGene/P

Some of our experiments were carried out on the Shaheen BlueGene/P at the Supercomputing Laboratory at King Abdullah University of Science&Technology (KAUST) in Thuwal, Saudi Arabia. Shaheen has 16 racks with a total of 16384 nodes. Each node is equipped with four 32-bit, 850 Mhz PowerPC 450 cores and 4GB DDR memory. The BlueGene/P (BG/P) architecture provides a three-dimensional point-to-point BlueGene/P torus network which interconnects all compute nodes and global networks for collective and interrupt operations. The use of this network is integrated into the BG/P MPI implementation. BlueGene/P MPI is based on MPICH which uses three different broadcast algorithms depending on the message size and the number of processes in a broadcast operation [59]:

- binomial tree algorithm - when the message size is less than 12kB or when the number of processes is less than eight.
- scatter-recursive-doubling-allgather algortihm - when the message size is less than 512kB and the number of processes is a power-of-two.
- scatter-ring-allgather (SRGA) algorithm - otherwise, for long messages greater than or equal to 512kB or with non power-of-two number of processes.

Despite the referenced paper [59] was published more than a decade ago it still reflects the current version of MPI broadcast operation implemented in MPICH according to its source code.

In addition to the algorithms implemented in MPICH, the broadcast operation on BG/P comes with different optimizations and algorithms specifically for the BG/P itself. Namely, if the communicator is MPI_COMM_WORLD it uses the BG/P collective tree network which supports hardware accelerated collective operations such as broadcast and all-reduce, and otherwise depending on the communicator shape either a rectangular broadcast algorithm or the broadcast algorithms from MPICH are used [88]. However, algorithms for some fundamental scientific applications such as parallel matrix multiplication, LU factorization does not use MPI_COMM_WORLD in their main communication steps, for example, it is more typical to use sub-communicators for rows and columns in a two-dimensional arrangement of processes. On the other hand, the rectangular broadcast is used only for rectangular shaped sub-communicators which strongly depends on the mapping of the processes into the physical topology and depending on the allocated BG/P partition can be arbitrary. Furthermore, the optimal mapping of processes to network hardware is not a trivial task and is a separate research area itself. The proposed optimization in this work is more general and topology-oblivious.

We present experiments with the corresponding hierarchical modifications of the scatter-ring-allgather algorithm and the native MPI broadcast operation. Experiments with the binomial and scatter-recursive-doubling-allgather algorithms demonstrated only slight fluctuations as expected theoretically.

While performance modeling and analysis of the BG/P-specific broadcast algorithms and optimizations are beyond the scope of this text, we present some experiments with the native BG/P broadcast operation as well. The experiments have been done with different configurations, message sizes from 1kB up to 16MB and the number of MPI processes from 8 up to 6142. The number of the allocated BG/P nodes was 6144, however we deliberately excluded two of them and used 6142 nodes by creating sub-communicators to avoid the case with MPI_COMM_WORLD. We present results mainly for

2048 and 6142 processes and message sizes of 512kB and 2MB. Figure 3.3 demonstrates experiments with the scatter-ring-allgather broadcast using message sizes of 512kB (left) and 2MB (right). The improvement with a message size of 512kB on 2048 nodes is 1.87 times, however with a message size of 2MB there is a performance drop. This is an expected behaviour according to the formula 3.6. On the other hand, because of the same formula it is expected that if we fix the message size, and keep increasing the number of processes the hierarchical transformation gradually should improve the performance. This is validated with the experiments: Figure 3.4 shows that for a message size of 512kB the speedup increases up to 3.09 times on 6142 nodes and unlike on 2048 nodes, the hierarchical algorithm outperforms the original algorithm with a message size of 2MB as well. In addition, if we put the platform and algorithm parameters in formula 3.5, we will see that the theoretically expected plots of the hierarchical algorithm will be parabola-like as well (Figure 3.4). Experiments with the native BG/P MPI broadcast operation are given in Figure 3.5.

It is already mentioned that during these experiments a sub-communicator of size 6142 was deliberately created from an MPI_COMM_WORLD of size 6144 to disable BG/P optimizations for MPI_COMM_WORLD. As a result the native BG/P MPI broadcast operation performed worse than the scatter-ring-allgather broadcast with a message size of 2MB.

It is obvious that the time between the groups will increase if the number of groups increases, however it is the opposite inside the groups. Figure 3.3 confirms that by showing the broadcast times separately spent inside and between groups.

Our theoretical models do not take sub-communicator creation overheads into account. However, MPI_Comm_split is also a collective operation and depending on the number of groups it makes different contributions into the overall time of the hierarchical broadcast. For example, in Figure 3.3 we do not have the expected upside-down parabola-like shape for a message size of 2MB because of the additional overhead from MPI_Comm_split operations. The reason is that when the number of groups is 2 or 1024 the total cost of creation of the two sub-communicators exceeds the gains due to

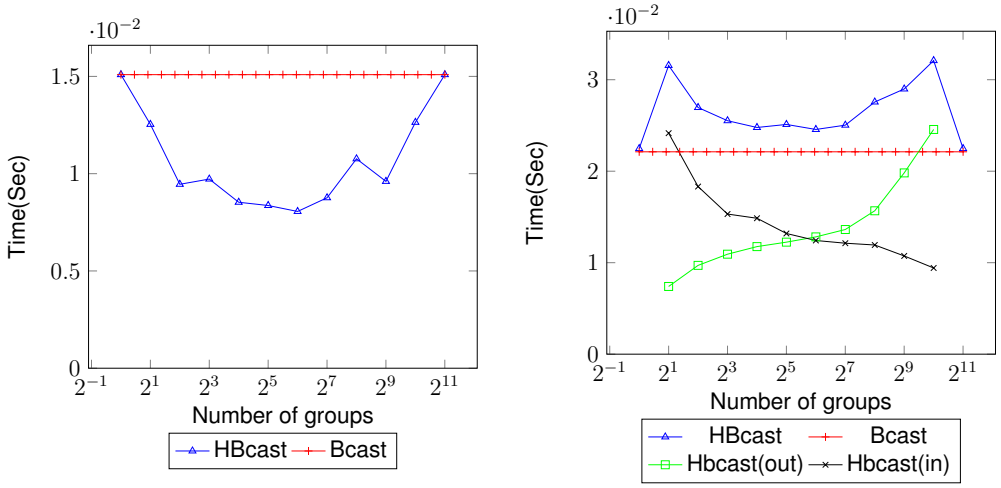


Figure 3.3: Hierarchical SRGA bcast on 2048 cores of BG/P with messages sizes 512kB (left) and 2MB (right). The experiments with 2MB also present the broadcast times spent inside each group and outside between groups.

the hierarchical optimization. Figure 3.3 (right) demonstrates these results. If the reduction of the execution time due to the optimization is greater than the overhead itself then the sub-communicator creation times will be well compensated by the reduction. It is the case in the experiments with 512kB (Figure 3.3).

Figure 3.6 presents the results of experiments with scatter-ring-allgather and native MPI broadcast operations. It shows the speedup due to the hierarchical optimization of these operations. Here the number of processes is fixed to be equal to 6142, and the message sizes change from 1kB up to 16MB. The experiments with 2048 processes have more data points than that of 6142 processes. The reason for that is we take only the factors of the number of processes as group numbers.

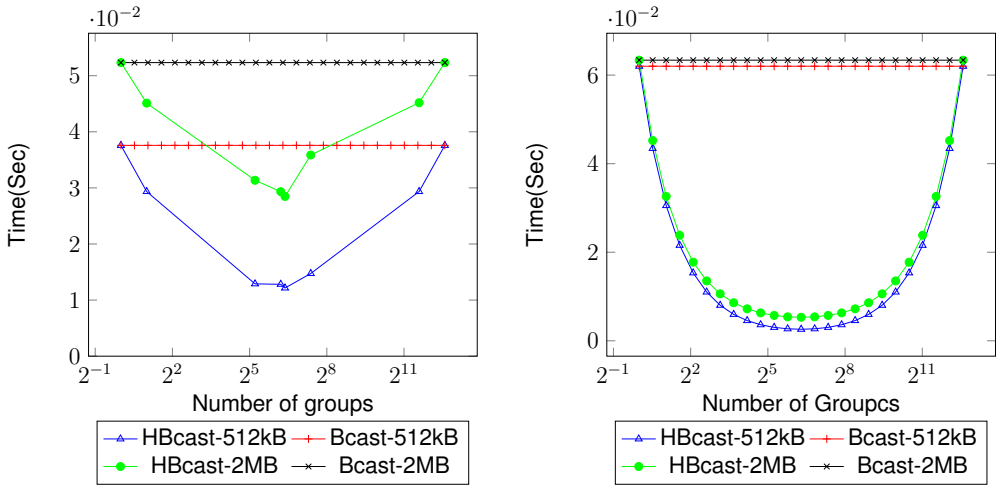


Figure 3.4: Hierarchical SRGA bcast on 6142 cores of BG/P with message sizes of 512kB and 2MB (left), and theoretical prediction using the same broadcast algorithm (right)

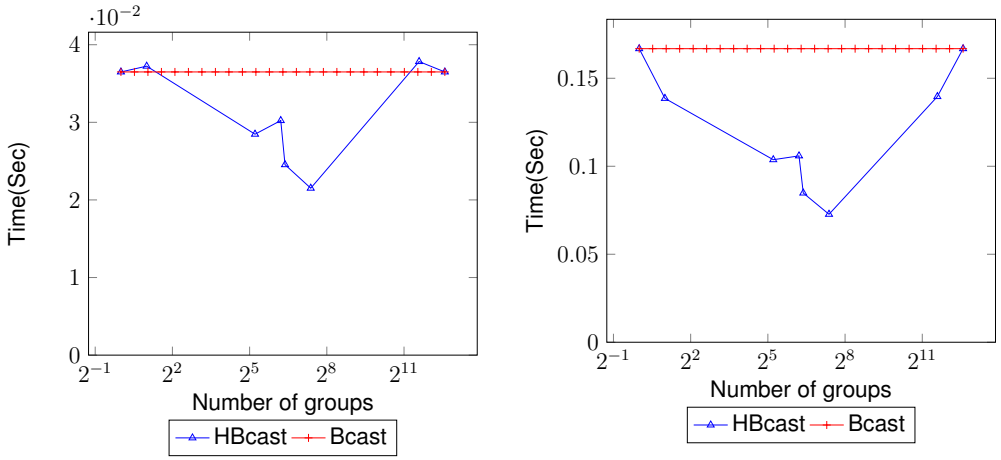


Figure 3.5: Hierarchical bcast on 6142 cores with message sizes 512kB (left) and 2MB (right)

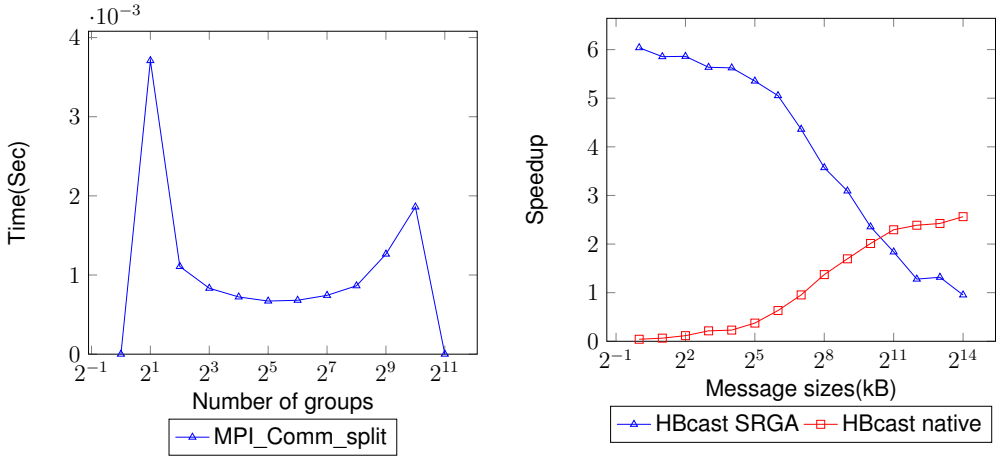


Figure 3.6: Time spent on MPI_comm_split time on 2048 cores (left) and Speedup of hbcast on 6142 cores (right)

3.1.2.2 Experiments on Grid'5000

The next part of the experiments was carried out on the Graphene cluster of the Nancy site of the Grid'5000 [108] infrastructure in France. The platform consists of 20 clusters distributed over nine sites in France and one in Luxembourg. The Grid'5000 web site (`http://www.grid5000.fr`) provides more comprehensive information about the platform.

The experiments on Grid'5000 have been done with Open MPI 1.4.5 which provides a few broadcast implementations, such as flat, chain (linear), pipelined, binary, binomial, split-binary tree. We present experimental study with Open MPI native broadcast operation, the chain and pipeline broadcast algorithms. During the experiments the hierarchical transformations of the binary and binomial tree algorithms had the same performance as the original algorithms. The same technique as described in MPIBlib [34] has been used to benchmark the performance.

3.1.2.3 Experiments on Grid'5000: One Process per Node

An experimental study with the Open MPI native broadcast operation is given in Figure 3.7. The first measurement was performed with a message size of 16kB where there is more than 3 times improvement. The experiment with

16MB showed 2.6 times reduction of the broadcast time. In the experiments with smaller message sizes up to 1kB and 128 processes the overhead from the two MPI_Comm_split operations was higher than the broadcast itself. However, with message sizes larger than 1kB the overhead from the split operations was negligible, for example, Figure 3.7 also shows the split time on 128 nodes. We had the same trend with larger number of processes. Figure 3.8 shows experimental results with the chain broadcast algorithm and its hierarchical transformation for message sizes of 16kB and 16MB. The speedup with the first setting is more than 8 times and with 16MB there is about 3 times improvement. In such situations an implementation of the algorithm could check the message size beforehand and fall back to use the regular MPI_Bcast for short messages to reduce the overhead even further.

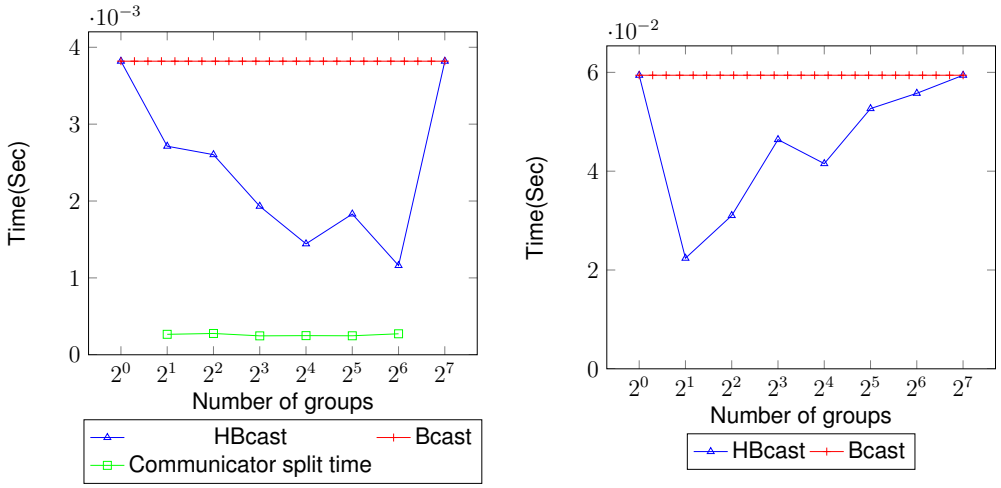


Figure 3.7: Hierarchical native MPI broadcast. m=16kB (left) and m=16MB (right), p=128.

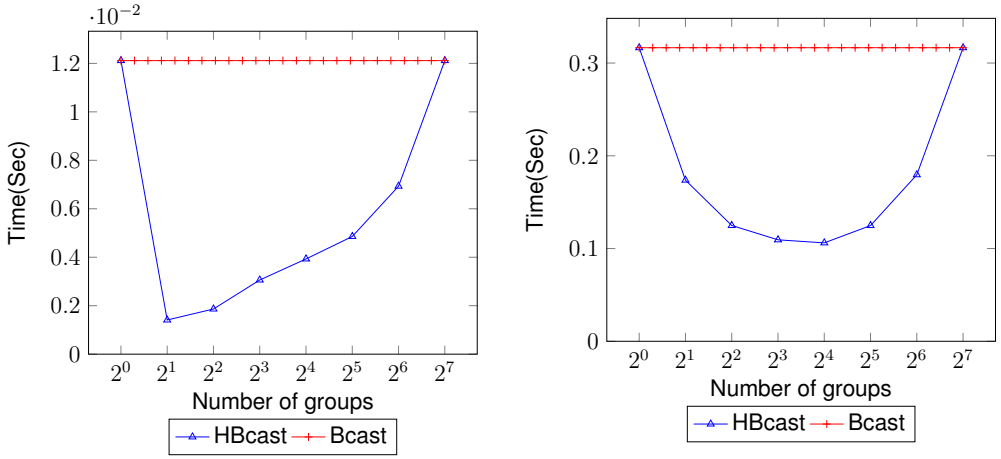


Figure 3.8: Hierarchical chain broadcast. m=16kB (left) and m=16MB (right), p=128.

Figure 3.9 show experiments with the pipeline broadcast algorithm and its hierarchical transformation. This time the speedup is more than 30 times with a message size of 16kB and more than 5 times with 16MB. Figure 3.10 shows the speedup for different numbers of processes for a fixed message size of 16MB and the speedup for different message sizes on 128 nodes.

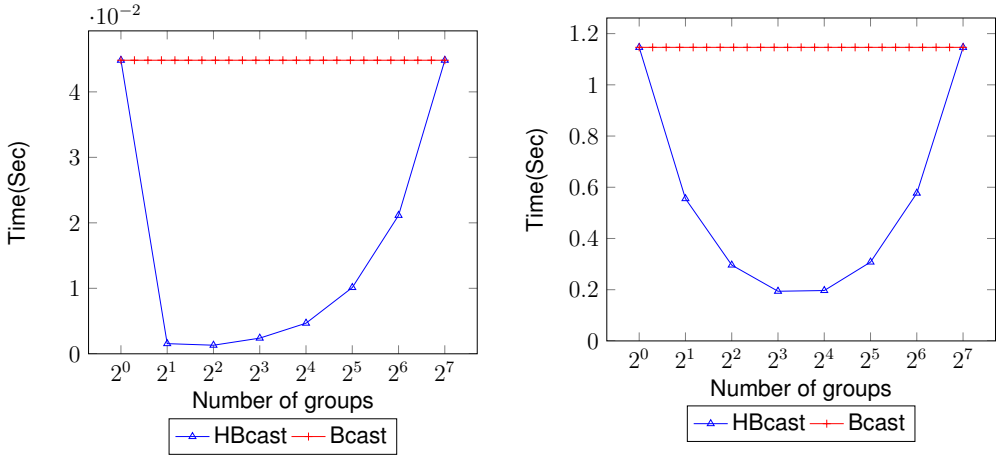


Figure 3.9: Hierarchical pipeline broadcast on Grid'5000. m=16kB (left) and m=16MB (right), p=128.

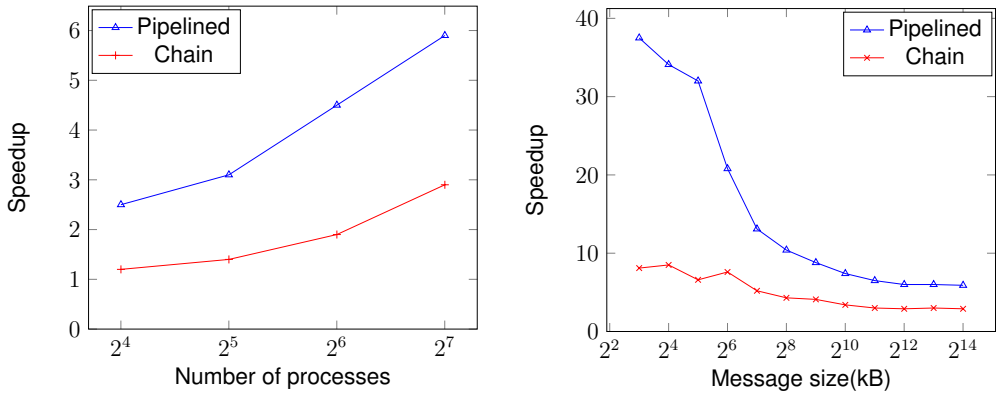


Figure 3.10: Speedup of HBcast over Bcast on Grid'5000. On the left the message size is fixed to 16MB, on the right the number of processes is fixed to 128.

3.1.2.4 Experiments on Grid'5000: One Process per Core

This section presents experiments with a one process per core configuration, or equivalently four processes per node. Figure 3.11 shows experimental results for the chain and pipeline broadcasts on 512 cores with message sizes of 16kB and 16MB. Figure 3.12 shows the speedup of the hierarchical chain and the hierarchical pipeline broadcast algorithms for a fixed message size of 16Mb and a power-of-two number of processes varying from 32 to 512 and speedup with message sizes from 16kB up to 16MB on 512 cores.

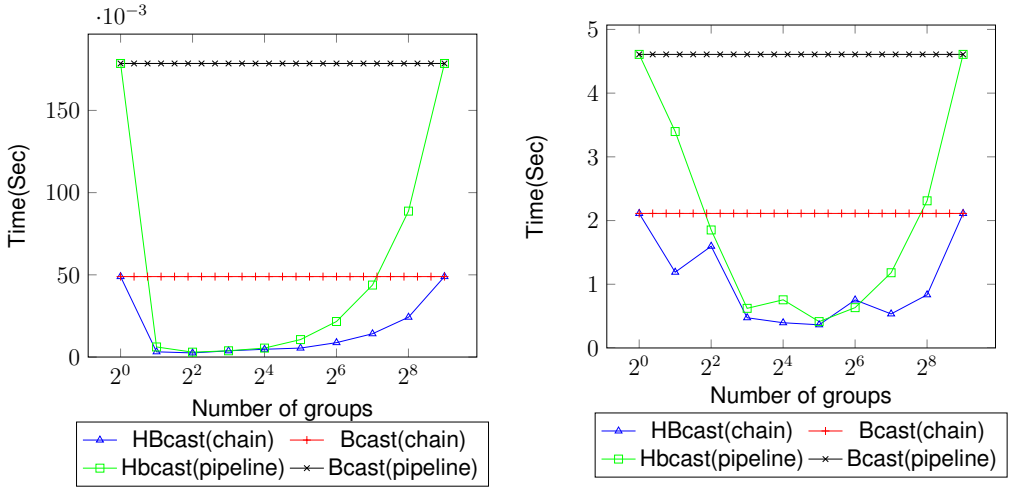


Figure 3.11: Hierarchical broadcast on Grid'5000, m=16kB (left) and m=16MB (right), p=512

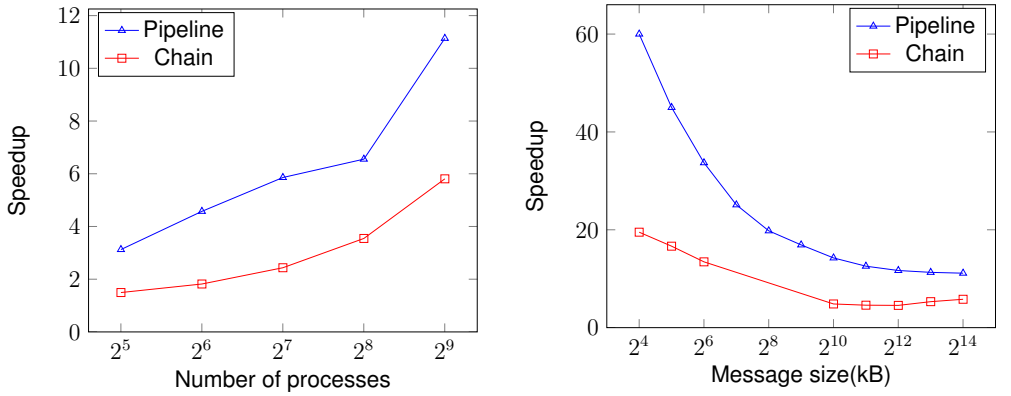


Figure 3.12: Speedup of Hbcast over Bcast. On the left the message size is fixed to 16MB and on the right the number of processes is fixed to 512.

3.2 Hierarchical Transformation of MPI Reduction Algorithms

3.2.1 Hierarchical Transformation of MPI Reduce algorithms

Reduce is important and commonly used collective operation in the Message Passing Interface (MPI) [16]. A five-year profiling study [97] demonstrates that MPI reduction operations are the most used collective operations. In the reduce operation each node i owns a vector x_i of n elements. After completion of the operation all the vectors are reduced element-wise to a single n-element vector which is owned by a specified root process.

Optimization of MPI collective operations has been an active research topic since the advent of MPI in 1990s. Many general and architecture-specific collective algorithms have been proposed and implemented in the state-of-the-art MPI implementations. As we have seen in the previous section the hierarchical topology-oblivious transformation of existing broadcast algorithms demonstrates as a new promising approach to optimization of MPI collective communication algorithms and MPI-based applications. Our study shows that by using this approach significant multi-fold performance gains can be achieved, especially on large-scale HPC systems.

We propose a hierarchical optimization of legacy MPI reduce algorithms without redesigning them. The approach is simple and general, allowing for application of the proposed optimization to any existing reduce algorithm. As by design the original algorithm is a particular case of its hierarchically transformed counterpart, the performance of the algorithm will either improve or stay the same in the worst case scenario. Theoretical study of the hierarchical transformation of six reduce algorithms which are implemented in Open MPI is presented. The theoretical results have been experimentally validated on a widely used Grid'5000 [109] infrastructure.

Similar to the optimization of the broadcast operation, the proposed

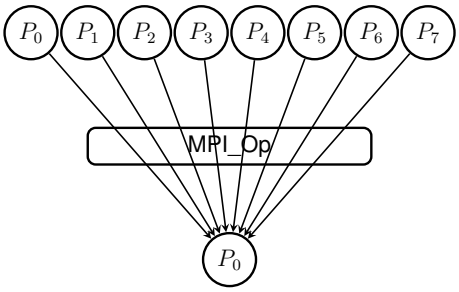


Figure 3.13: Logical arrangement of processes in MPI reduce.

optimization technique of the reduce is also based on the arrangement of the p processes participating in the reduce into logical groups. For simplicity, it is assumed that the number of groups divides the number of MPI processes and can change between one and p. Let G be the number of groups. Then there will be $\frac{p}{G}$ MPI processes per group. Figure 3.13 shows an arrangement of 8 processes in the original MPI reduce operation, and Figure 3.14 shows the arrangement in the hierarchical reduce operation with 2 groups of 4 processes. The hierarchical optimization has two phases: in the first phase, a group leader is selected for each group and the leaders start reduce operation inside their own group in parallel (in this example among 4 processes). In the next phase, the reduce is performed among the group leaders (in this example between 2 processes). The grouping can be done by taking the topology into account as well. However, in this work the grouping is topology-oblivious and the first process in each group is selected as the group leader. In general, different algorithms can be used for reduce operations among group leaders and within each group. This work focuses on the case where the same algorithm is employed at both levels of hierarchy.

Algorithm 2 shows the pseudocode of the hierarchically transformed MPI reduce operation. Line 4 calculates the root for the reduce between the groups. Then line 5 creates a sub-communicator of G processes between the groups, and line 6 creates a sub-communicator of $\frac{p}{G}$ processes inside the groups. Our implementation uses the MPI_Comm_split MPI routine to create new sub-communicators.

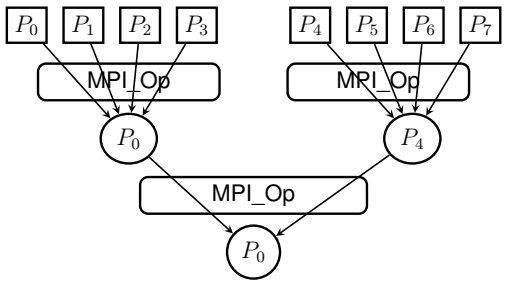


Figure 3.14: Logical arrangement of processes in hierarchical MPI reduce.

Data: p - Number of processes
Data: G - Number of groups
Data: $sendbuf$ - Send buffer
Data: $recvbuf$ - Receive buffer
Data: $count$ - Number of entries in send buffer (integer)
Data: $datatype$ - Data type of elements in send buffer
Data: op - MPI reduce operation handle
Data: $root$ - Rank of reduce root
Data: $comm$ - MPI communicator handle
Result: The root process has the reduced message
begin

```
1  MPI_Comm comm_outer      /* communicator between the groups */
2  MPI_Comm comm_inner       /* communicator inside the groups */
3  int root_outer           /* root of reduce between the groups */
4  root_outer = Calculate_Root_Outer(G, p, root, comm)
5  comm_outer = Create_Comm_Between_Groups(G, p, root_outer, comm)
6  comm_inner = Create_Comm_Inside_Groups(G, p, root, comm)
7  void* tmpbuf
8  MPI_Reduce(sendbuf, tmpbuf, count, datatype, op, root, comm_inner)
9  MPI_Reduce(tmpbuf, recvbuf, count, datatype, op, root_outer,
   comm_outer)
```

end

Algorithm 2: Hierarchical optimization of MPI reduce operation.

3.2.1.1 Theoretical Analysis

- Hierarchical Transformation of Flat Tree Reduce Algorithm
  The hierarchical transformation creates two phases of the reduce operation: inside the groups and outside the groups. In the first phase the reduce operations inside each group happen among $\frac{p}{G}$ processes in parallel. Then, in the next phase the operation continues among G processes which are leaders of their own groups. The cost of these two reduce operations will be $(G-1)\times(\alpha+m\times\beta+m\times\gamma)$ and $\left(\frac{p}{G}-1\right)\times(\alpha+m\times\beta+m\times\gamma)$ respectively. Thus, the overall run time can be seen as a function of G:

$$F(G)=\left(G+\frac{p}{G}-2\right)\times(\alpha+m\times\beta+m\times\gamma) \tag{3.10}$$

  The derivative of the function is $\left(1-\frac{p}{G^2}\right)\times(\alpha+m\times\beta+m\times\gamma)$ and it is easy to show that $p=\sqrt{G}$ is the minimum point of the function in the interval $(1,p)$. Therefore, the optimal value of the $F(G)$ function will be as follows:

$$F(\sqrt{p})=(2\times\sqrt{p}-2)\times(\alpha+m\times\beta+m\times\gamma) \tag{3.11}$$

- Hierarchical Transformation of Pipeline Reduce Algorithm
  If we apply the pipeline algorithm inside and outside the groups and sum these two costs up, the overall run time will be as follows:

$$F(G)=\left(2\times X+G+\frac{p}{G}-4\right)\times\left(\alpha+\frac{m}{X}\times\beta+\frac{m}{X}\times\gamma\right) \tag{3.12}$$

  Similar to the previous algorithm we can show that the optimal value of the cost function is the following:

$$F(\sqrt{p})=(2\times X+2\sqrt{p}-4)\times\left(\alpha+\frac{m}{X}\times\beta+\frac{m}{X}\times\gamma\right) \tag{3.13}$$

- Hierarchical Transformation of Binary Reduce Algorithm
  For simplicity, we will take $p+1\approx p$ in the formula 2.14. Then the cost of

the reduce operations among the groups and inside the groups will be as follows respectively: $2\times \log_2(G)\times(\alpha + m\times\beta + m\gamma)$ and $2\times \log_2(\frac{p}{G})\times(\alpha + m\times\beta + m\gamma)$. If we add these two terms, the overall cost of the hierarchical transformation of the binary tree algorithm will be equal to the cost of the original algorithm.

- Hierarchical Transformation of Binomial Reduce Algorithm
  Similarly to the binary reduce algorithm, the cost function of the binomial tree will not change after hierarchical transformation.

- Hierarchical Transformation of Rabenseifner's Reduce Algorithm
  By applying the formula 2.16 among G groups (G processes) and inside each group of $\frac{p}{G}$ processes, we can find the run time of hierarchical transformation of Rabenseifner's algorithm. Unlike the previous algorithms, now the theoretical cost increases in comparison to the original Rabenseifner's algorithm. Therefore, theoretically the hierarchical reduce implementation should use the number of groups equals to one, in which case the hierarchical algorithm retreats to the original algorithm.

$$2\times \log_2(p)\times\alpha + 2\times m\times\beta\times\left(2-\frac{G}{p}-\frac{1}{G}\right) + m\times\gamma\times\left(2-\frac{G}{p}-\frac{1}{G}\right) \tag{3.14}$$

3.2.1.2 Experimental Study

The Graphene cluster from Nancy site of the Grid'5000 infrastructure was our main test-bed in the experiments with MPI reduce. The experiments conducted using Open MPI 1.4.5 which comes with several reduce algorithms such as linear (flat), chain, pipeline, binary, binomial, and in-order binary tree algorithms. More recent versions of Open MPI provides platform/architecture specific algorithms as well, some of which are reduce algorithms for Infiniband networks, and the Cheetah framework for multicore architectures. In this work, we do not consider the reduce implementations that are optimized specifically to specific type of platforms. We used the

same approach as described in MPIBlib [34] to benchmark our experiments. During the experiments, the mentioned reduce algorithms were selected by using Open MPI MCA (Modular Component Architecture) *coll_tuned_use_dynamic_rules* and *coll_tuned_reduce_algorithm* parameters. MPI_MAX operation has been used in the experiments. We used two experimental settings, one process per core and one process per node with the Infiniband-20G network. A power-of-two number of processes were used in the experiments.

Experiments: One Process per Core. The nodes in the Graphene cluster are organized into four groups and connected to four switches. The switches in turn are connected to the main Nancy router. We have used 10 patterns of process to core mappings, but we will show experimental results only with one such mappings where the processes are grouped by their rank in increasing order. The measurements with different groupings showed similar performance.

The theoretical and experimental results showed that the hierarchical approach mainly improves the algorithms which assume flat arrangements of the processes, such as linear, chain and pipeline. On the other hand native Open MPI reduce operation consists of different algorithms where a specific algorithm is selected depending on the message size, the count and the number of processes sent to the MPI_Reduce function. This means the hierarchical transformation can improve the native reduce operation as well. The algorithms used in the Open MPI decision function are linear, chain, binomial, binary/in-order binary and pipeline reduce algorithms which can be used with different sizes of segmented messages.

Figure 3.15 shows experiments with default Open MPI reduce operation with a message of size 16KB where the best performance is achieved when the group size is 1 or p, in which case the hierarchical reduce obviously turns into the original non-hierarchical reduce. Here for different numbers of groups the Open MPI decision function selected different reduce algorithms. Namely, if the number of groups is 8 or 64 then Open MPI selects the binary tree reduce algorithm between the groups and inside the groups respectively. In

Table 3.1: Open MPI algorithm selection in HReduce. m=16MB, p=512.

Groups	Inside groups	Between groups
1	-	Pipeline 32KB
2	Pipeline 32KB	Pipeline 64KB
4	Pipeline 32KB	Pipeline 64KB
8	Pipeline 32KB	Pipeline 64KB
16	Pipeline 64KB	Pipeline 64KB
32	Pipeline 64KB	Pipeline 64KB
64	Pipeline 64KB	Pipeline 32KB
128	Pipeline 64KB	Pipeline 32KB
256	Pipeline 64KB	Pipeline 32KB

all other cases the binomial tree reduce algorithm is used. The reduction of the communication time can be significant, for example, in case of 16MB it is about 30 times. This improvement does not come solely from the hierarchical optimization itself, but also because of the number of groups in the hierarchical reduce resulted in Open MPI decision function to select the pipeline reduce algorithm with different segment sizes for each groups. The selection of the algorithms for different number of groups is described in Table 3.1.

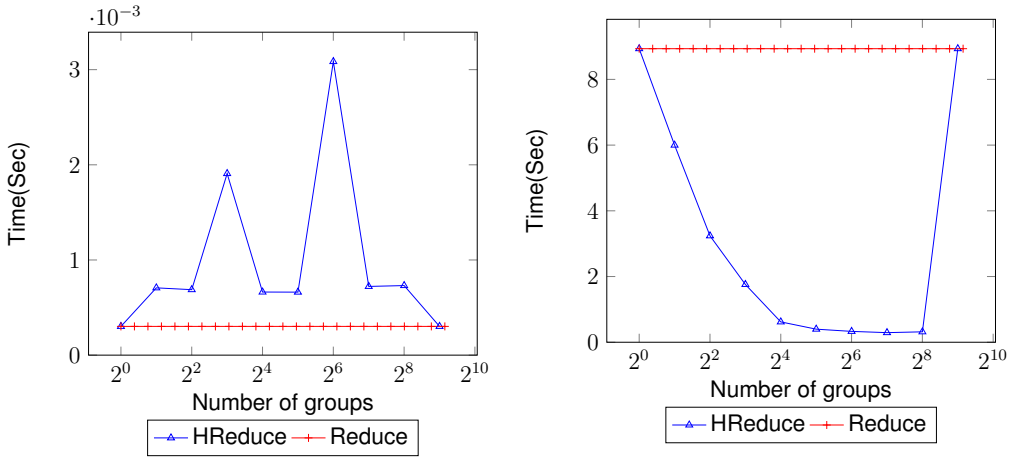


Figure 3.15: Hierarchical native Open MPI reduce operation on 512 cores with message sizes of 16KB (left) and 16MB (right)

As the MPI_Comm_split is a collective operation, it is expected that the

overhead from the split operation should affect reduce operations with smaller message sizes. Figure 3.16 validates this with experimental results. The hierarchical reduce operation of 1KB message with the underlying native reduce achieved its best performance when the number of groups was one as the overhead from the split operation itself was higher than the reduce.

It is interesting to study the pipeline algorithm with different segment sizes as this algorithm is used for large message sizes in Open MPI. Figure 3.16 presents experiments with the hierarchical pipeline reduce with a message size of 16KB with 1KB segmentation. We selected the segment sizes using Open MPI *coll_tuned_reduce_algorithm_segmentsize* MCA parameter. Figure 3.17 shows the performance of the pipeline algorithm with segment sizes of 32KB and 64KB. In the first case, we see a 26.5 times improvement, while with the 64KB the improvement is 18.5 times.

Figure 3.18 demonstrates speedup of the hierarchical transformation of native Open MPI reduce operation, linear, chain, pipeline, binary, binomial, and in-order binary reduce algorithms with message sizes starting from 16KB up to 16MB. Except binary, binomial and in-order binary reduce algorithms, there is a significant performance improvement. In the figure, NT is native Open MPI reduce operation, LN is linear, CH is chain, PL is pipeline with 32KB segmentation, BR is binary, BL is binomial, and IBR denotes in-order binary tree reduce algorithm. We would like to highlight one important point that Figure 3.18 does not compare the performance of different Open MPI reduce algorithms, it rather shows the speedup of their hierarchical transformations. Each of these algorithms can be better than the others in some specific settings depending on the message size, number of processes, underlying network and so on. At the same time, the hierarchical transformation of these algorithms will either improve their performance or be equally fast.

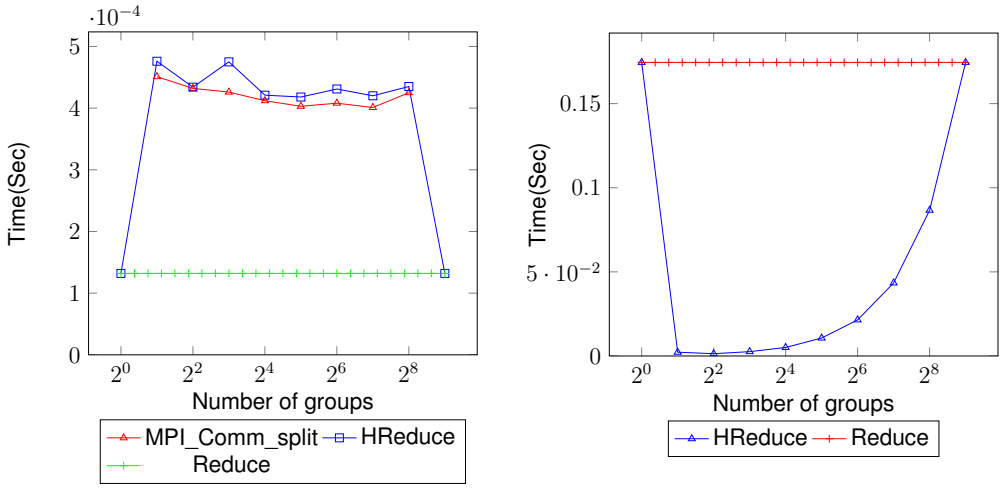


Figure 3.16: Time spent on MPI_Comm_split and hierarchical native reduce with a message size of 1KB (left), and time spent on hierarchical pipeline reduce with a message size of 16KB with 1KB segments on 512 cores

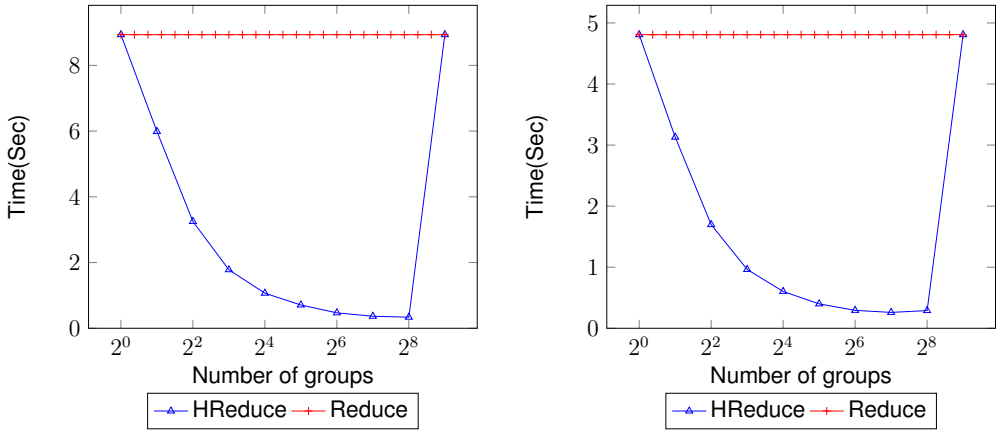


Figure 3.17: Hierarchical pipeline reduce with a message size of 16MB, segment sizes of 32KB (left) and 64KB (right) on 512 cores

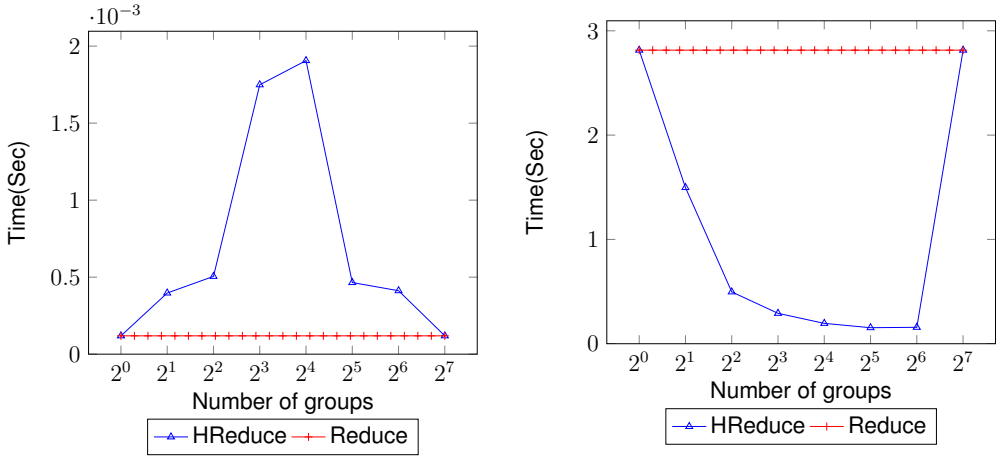


Figure 3.19: Hierarchical native reduce on 128 cores with message sizes of 16KB (left) and 16MB (right)

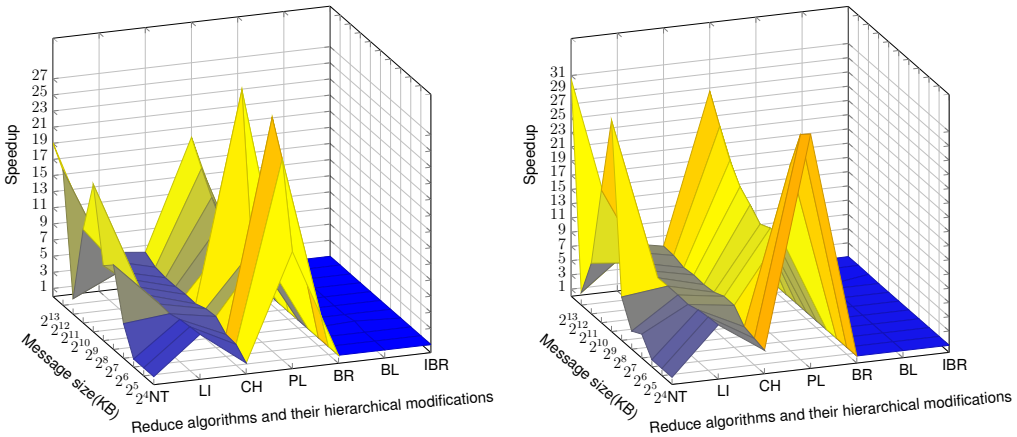


Figure 3.18: Speedup on 256(left) and 512(right) cores, one process per core.

Experiments: One Processes per Node. The experiments with one process per node showed a similar trend to that of with the one process per core setting. The performance of linear, chain, pipeline and native Open MPI reduce operations can be improved by the hierarchical approach. Figure 3.19 presents experiments on 128 nodes with message sizes of 16KB and 16MBy. In the first setting, the Open MPI decision function uses the binary tree algorithm when the number of processes is 8 between or inside groups, in all other cases the binomial tree is used.

The pipeline algorithm has similar performance improvement to that of with 512 processes, Figure 3.20 shows experiments with a message of size 16MB segmented by 32KB and 64KB sizes. The labels on the x axis has the same meaning as in the previous section.

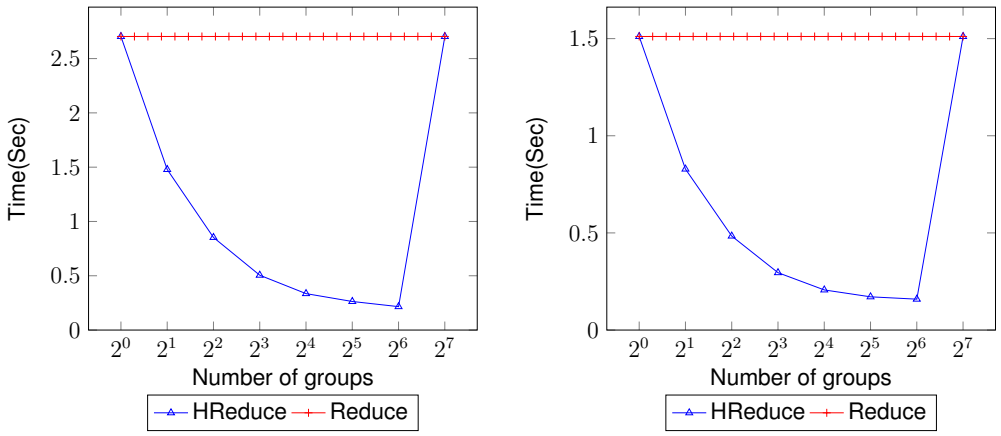


Figure 3.20: Hierarchical pipeline reduce. m=16MB, segment 32KB (left) and 64KB (right). p=128.

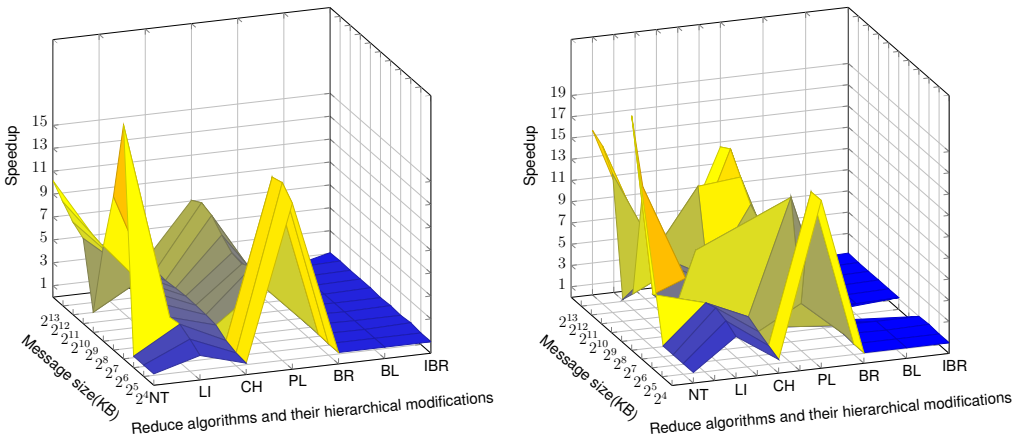


Figure 3.21: Speedup on 64(left) and 128(right) cores. 1 process per node.

Figure 3.21 presents speedup of the hierarchical transformations of all the reduce algorihms from Open MPI "TUNED" component with message sizes from 16KB up to 16MB on 64 (left) and 128 (right) nodes. Again, the reduce algorithms wich have "flat" design and Open MPI default reduce operation can be significantly improved.

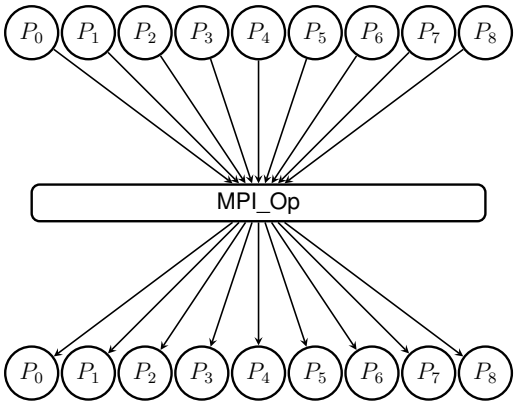


Figure 3.22: Logical arrangement of processes in MPI allreduce.

3.2.2 Hierarchical Transformation of MPI Allreduce

The allreduce operation has a more complex communication pattern than the broadcast and reduce operations. Therefore its hierarchical transformation is not as trivial as it was in the previous cases. The main difficulty comes from the fact that in our design we are trying not to introduce a new allreduce algorithm but rather use existing algorithms underneath. To be more clear, in the case of allreduce we would only like to use the allreduce communication operation both inside and between groups. If it was not the case, it would be possible to design a hierarchical allreduce in very different ways. One example of hierarchical allreduce implementation could be using a hierarchical reduce followed by a hierarchical broadcast.

The design of the hierarchical allreduce follows a similar design philosopy to the hierarchical broadcast and hierarchical reduce operations. Namely, the main idea is to organize the processes into logical groups. The grouping results in two-level hierarchy and the allreduce operation are performed in two phases. In the first phase the operation is operated on the processes inside each group independently. Later on, as soon as this phase finishes, the processes at the same index position from different groups start allreduce operation among them on the partially reduced value.

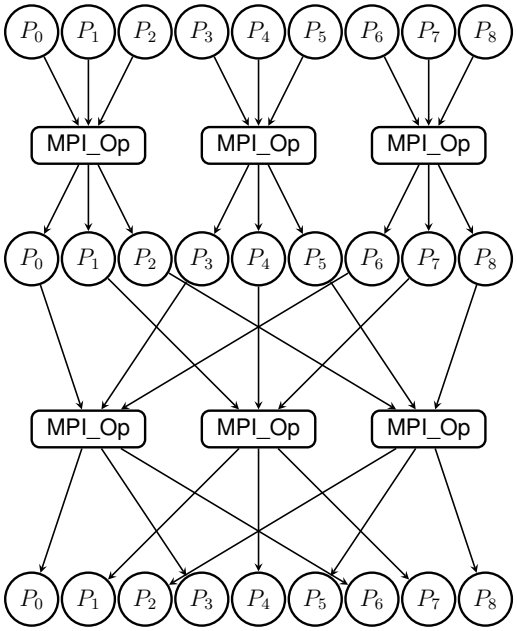


Figure 3.23: Logical arrangement of processes in hierarchical MPI allreduce.

Data: p - Number of processes
Data: G - Number of groups
Data: $sendbuf$ - Send buffer
Data: $recvbuf$ - Receive buffer
Data: $count$ - Number of entries in send buffer (integer)
Data: $datatype$ - Data type of elements in send buffer
Data: op - MPI operation handle
Data: $comm$ - MPI Communicator
Result: All the group members have the reduced message in their receive buffer
begin

```
1  MPI_Comm comm_outer      /* communicator between the groups */
2  MPI_Comm comm_inner       /* communicator inside the groups */
3  comm_outer = Create_Comm_Between_Groups(G, p, root_outer, comm)
4  comm_inner = Create_Comm_Inside_Groups(G, p, root, comm)
5  void* tmpbuf
6  MPI_Allreduce(sendbuf, tmpbuf, count, datatype, op, comm_inner)
7  MPI_Allreduce(tmpbuf, recvbuf, count, datatype, op, comm_outer)
```

end

Algorithm 3: Hierarchical transformation of MPI allreduce operation.

3.2.2.1 Hierarchical Transformation of Recursive Doubling Allreduce Algorithm

Like most of the logarithmic algorithms the cost of the hierarchical transformation of this algorithm is equal to the cost of the original algorithm. Thus, an implementation of the hierarchical algorithm may use the number of groups equal to one and fall back to the underlying algorithm in which case its cost will be as follows:

$$F(p) = \log_2 p \times (\alpha + m \times \beta + m \times \gamma) \tag{3.15}$$

As we can see the theoretical cost does not depend on the number of groups.

3.2.2.2 Hierarchical Transformation of Rabensifner's Allreduce Algorithm

After the hierarchical transformation, Rabensifner's allreduce algorithm will have the following theoretical cost:

$$F(G) = 2 \times \log_2 p \times \alpha + 2 \times m \times \beta \times \left(2 - \frac{1}{G} - \frac{G}{p}\right) + m \times \gamma \times \left(2 - \frac{1}{G} - \frac{G}{p}\right) \tag{3.16}$$

The derivative of the function is given below:

$$F'_G(G) = 2 \times m \times \beta \times \left(\frac{1}{G^2} - \frac{1}{p}\right) + m \times \gamma \times \left(\frac{1}{G^2} - \frac{1}{p}\right) \tag{3.17}$$

It is clear that $G = \sqrt{p}$ is the critical point in $(1, p)$ and it is the local maximum of the function in the interval. The value of the function at this point will be the following:

$$F(\sqrt{p}) = 2 \times \log_2 p \times \alpha + 4 \times m \times \beta \times \left(1 - \frac{1}{\sqrt{p}}\right) + 2 \times m \times \gamma \times \left(1 - \frac{1}{\sqrt{p}}\right) \tag{3.18}$$

Thus, the hierarchical transformation does not improve the original algorithm.

3.2.2.3 Hierarchical Transformation of Ring Allreduce Algorithm

We analyse the hierarchical transformation of the ring allreduce algorithm using the same technique we have used to analyse hierarchical broadcast and reduce algorithms. Namely, we apply the formula 2.20 among G processes outside groups and among $\frac{p}{G}$ processes inside the groups and assume that $\lceil \frac{m}{p} \rceil = \frac{m}{p}$. Then if we sum those two costs up the theoretical cost of the hierarchical algorithm can be derived as follows:.

$$F(G) = 2\times(G-1)\times\left(\alpha + \frac{m}{G}\times\beta\right) + 2\times\left(\frac{p}{G}-1\right)\times\left(\alpha + m\times\frac{G}{p}\times\beta\right) + \\ + (G-1)\times\gamma\times\frac{m}{G} + \left(\frac{p}{G}-1\right)\times\gamma\times m\times\frac{G}{p} \quad (3.19)$$

It can easily be shown that $G = \sqrt{p}$ is the extremum point of the $F(G)$ function in the interval $(1, p)$ and the function attains its minimum at that point. Thus, the optimal value of the function will be as follows:

$$F(G) = 4\times(\sqrt{p}-1)\times\alpha + 4\times\left(1-\frac{1}{\sqrt{p}}\right)\times m\times\beta + 2\times\left(1-\frac{1}{\sqrt{p}}\right)\times m\times\gamma \quad (3.20)$$

3.2.2.4 Hierarchical Segmented Ring Allreduce Algorithm

The theoretical cost of the hierarchical segmented ring allreduce can be derived in a similar way. Let us assume that $\lceil \frac{m}{p} \rceil = \frac{m}{p}$. Then the total cost of the algorithm will be equal to the following function:

$$F(G) = \left(G + \frac{p}{G} + 2\times X - 4\right)\times\left(\alpha + \frac{m}{X}\times\beta + \frac{m}{X}\times\gamma\right) + \left(G + \frac{p}{G} - 2\right)\times\alpha + \\ + \left(2 - \frac{1}{G} - \frac{G}{p}\right)\times m\times\beta \quad (3.21)$$

It is easy to find that the derivative of this function is as follows:

$$F'(G) = \frac{(G^2-p)\times(2\times p\times X\times\alpha + m\times(p-X)\times\beta + m\times p\times\gamma)}{G^2\times P\times X} \quad (3.22)$$

Thus, $G = \sqrt{p}$ will be the local minimum point of the function in $(1, p)$ if $2{\times}p{\times}X{\times}\alpha + m{\times}(p - X){\times}\beta + m{\times}p{\times}\gamma > 0$ and the minimum value of the function will be as below:

$$F(\sqrt{p}) = (2{\times}\sqrt{p} + 2{\times}X - 4) \times \left(\alpha + \frac{m}{X}{\times}\beta + \frac{m}{X}{\times}\gamma\right) + \\ + (2{\times}\sqrt{p} - 2) \times\alpha + 2{\times}\left(1 - \frac{1}{\sqrt{p}}\right) \times m{\times}\beta \quad (3.23)$$

3.2.2.5 Experiments on Grid'5000

The experiments with allreduce have been conducted on the Graphene cluster using Open MPI version 1.8.4 in two experimental settings, one process per node (each node consists of four processes) and one process per core (equivalently four process per node). Our study covers basic linear, non-overlapping (Open MPI tuned broadcast + tuned reduce), recursive doubling, ring, segmented ring and Open MPI default allreduce operation. The experiments performed with different message sizes starting from 4kB up to 16MB on multiple number of processes in the range of 8 and 512. In addition, the experiments includes results with different segment sizes for the segmented ring algorithm. Figure 3.24 shows experiments with the ring algorithm for message sizes 4KB and 16MB on 512 cores. In case of large messages there is no performance improvement over the original ring algorithm. However, for small messages the hierarchical transformation can improve the ring algorithm. For instance, when the message size is 4KB there is more than 2.5 times of performance improvement. However if the message is greater than 16KB we do not get performance improvements on the Graphene cluster.

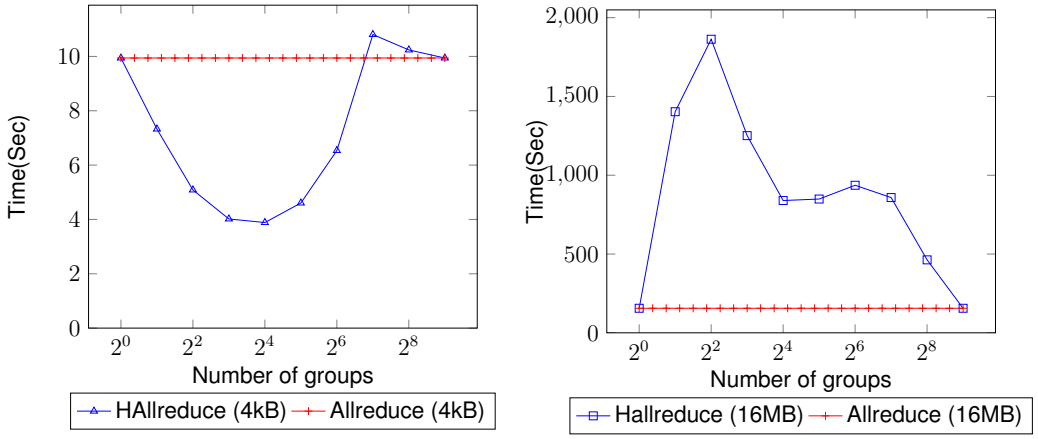


Figure 3.24: Hierarchical ring allreduce algorithm on 512 cores. Message size: 4KB and 16MB (right).

The tendency with the segmented ring algorithm is similar to that of the ring algorithm, where there are performance improvements for small messages. There is 30% improvement when the message size is 64KB, and 2 times improvement for a message size of 4KB (Figure 3.25). There is no improvement for messages larger than 64KB. Figure 3.26 demonstrates experiments with the linear allreduce algorithm (on the left) and speedup of hierarchical allreduce with different message sizes and algorithms (on the right). The abbreviations on the figure are the following: NT - native Open MPI allreduce operation, BL - basic linear, NOV - non-overlapping, RD - recursive doubling, RG - ring, and finally SRG - segmented ring algorithm. It can be seen that the improvement with the basic linear and non-overlapping algorithm can be more than 7 times. There is no improvement with the recursive doubling algorithm, while the ring and the segmented ring algorithms can be improved for small messages.

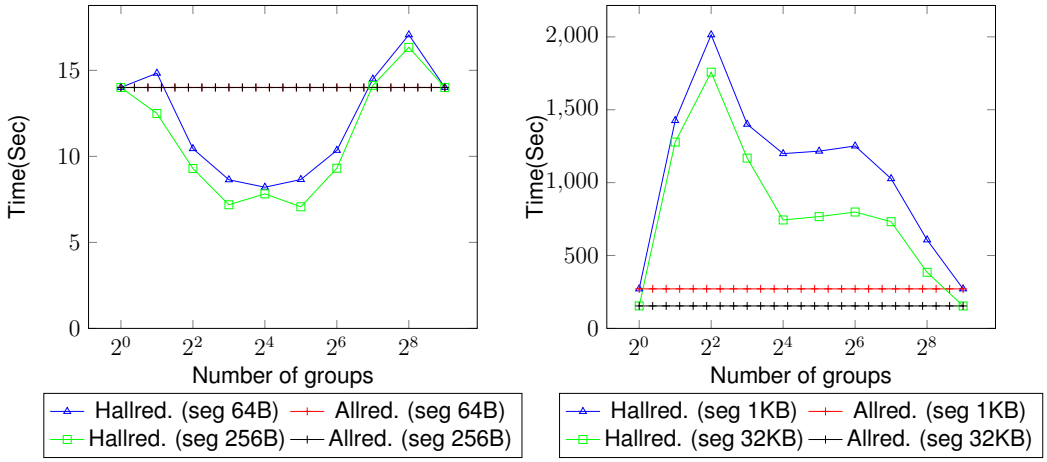


Figure 3.25: Hierarchical segmented ring allreduce algorithm on 512 cores. Message size: 4KB (left) and 16MB (right). Segment size: 1KB and 4KB.

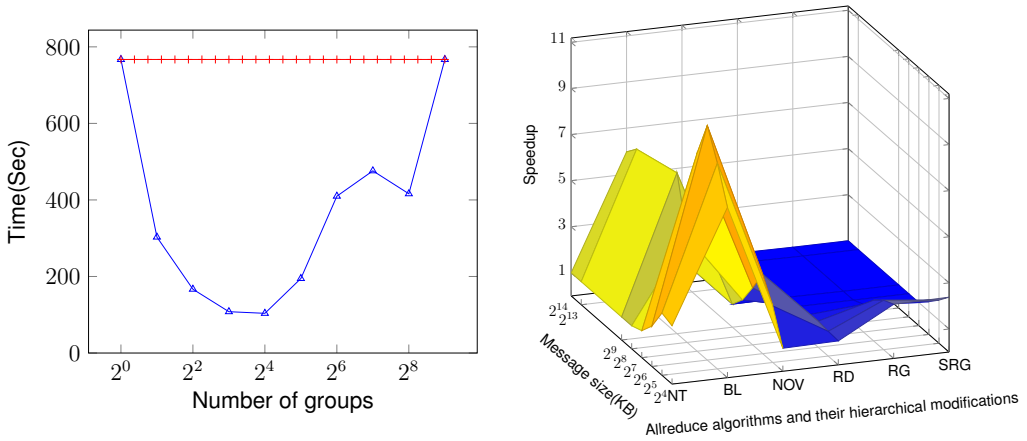


Figure 3.26: Hierarchical linear allreduce performance with a message of size 16KB (left) and speedup of hierarchical allreduce on 512 cores with different algorithms (right), one process per core.

3.3 Hierarchical Transformation of MPI Scatter and Gather Operations

The hierarchical transformation of the scatter operation is quite similar to that of the broadcast operation. Namely, there are two phases again, in the first phase the scatter operation is performed among groups and after that in the second phase the operation is continued inside each group in parallel. The pseudocode of hierarchical scatter is given in Algorithm 4.

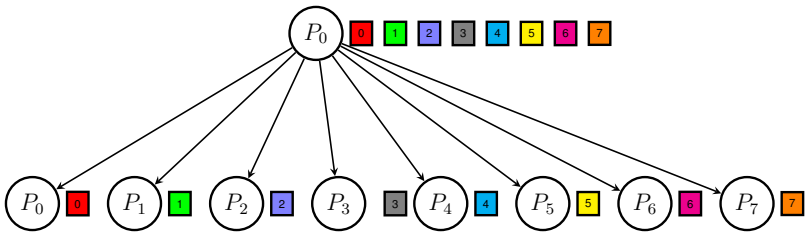


Figure 3.27: Logical arrangement of processes in MPI scatter.

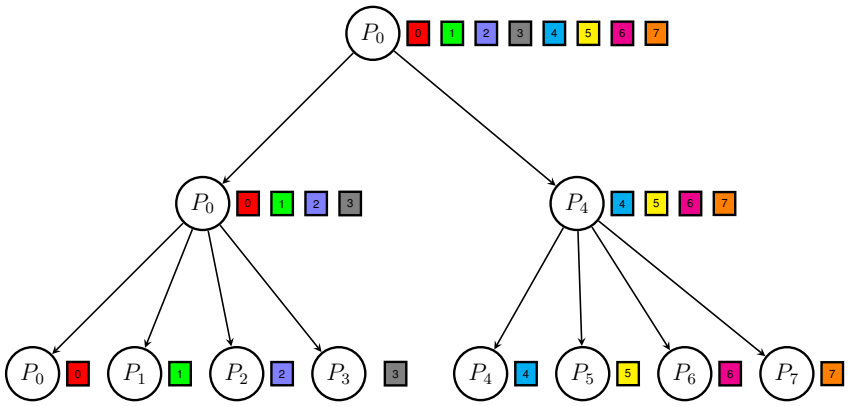


Figure 3.28: Logical arrangement of processes in hierarchical MPI scatter.

The hierarchical gather resembles the hierarchical reduce transformation in the sense that we first do gather operation inside each group then the locally gathered data further gathered into the final root process. Algorithm 5 illustrates the pseudocode of the hierarchical gather operation.

Data: p - Number of processes
Data: G - Number of groups
Data: $sendbuf$ - Send buffer
Data: $sendcount$ - Number of entries in send buffer (integer)
Data: $sendtype$ - Data type of entries in send buffer
Data: $recvbuf$ - Receive buffer
Data: $recvcount$ - Number of elements in receive buffer
Data: $recvtype$ - Data type of entries in receive buffer
Data: $root$ - Rank of scatter root
Data: $comm$ - MPI Communicator
Result: All the processes in the group receive $recvcount$ elements
begin

```
1  MPI_Comm comm_outer        /* communicator among the groups */
2  MPI_Comm comm_inner       /* communicator inside the groups */
3  int root_inner          /* root of broadcast inside the groups */
4  root_inner = Calculate_Root_Inner(G, p, root, comm)
5  comm_outer = Create_Comm_Between_Groups(G, p, root, comm)
6  comm_inner = Create_Comm_Inside_Groups(G, p, root_inner, comm)
   void* tmpbuf
7  MPI_Scatter(sendbuf, p/G *sendcount, sendtype, tmpbuf, p/G *recvcount,
   recvtype, root, comm_outer)
8  MPI_Scatter(tmpbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
   root_inner, comm_inner)
```

end

Algorithm 4: Hierarchical transformation of MPI scatter operation.

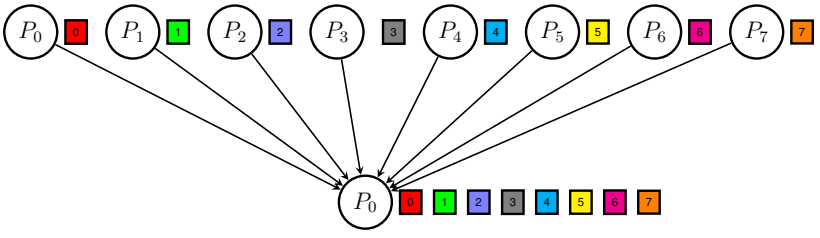


Figure 3.29: Logical arrangement of processes in MPI gather.

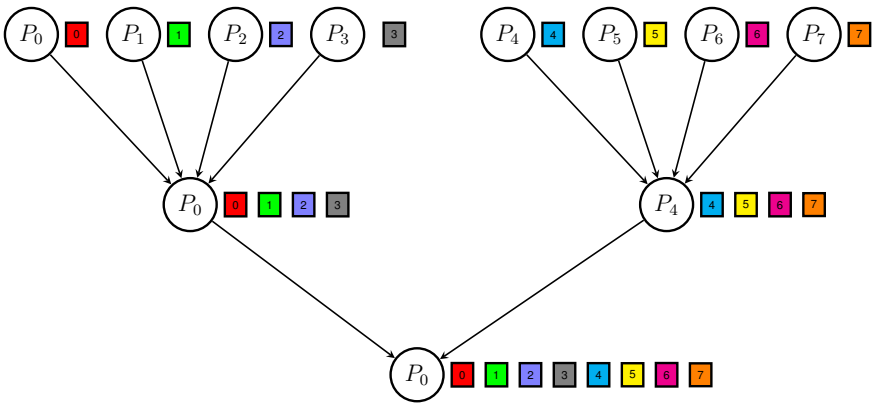


Figure 3.30: Logical arrangement of processes in hierarchical MPI gather.

Data: p - Number of processes
Data: G - Number of groups
Data: $sendbuf$ - Send buffer
Data: $sendcount$ - Number of entries in send buffer (integer)
Data: $sendtype$ - Data type of entries in send buffer
Data: $recvbuf$ - Receive buffer
Data: $recvcount$ - Number of elements in receive buffer
Data: $recvtype$ - Data type of entries in receive buffer
Data: $root$ - Rank of gather root
Data: $comm$ - MPI Communicator
Result: $root$ process receives $recvcount$ elements from all the processes in the group
begin

```
1  MPI_Comm comm_outer          /* communicator among the groups */
2  MPI_Comm comm_inner          /* communicator inside the groups */
3  int root_inner               /* root of broadcast inside the groups */
4  root_inner = Calculate_Root_Inner(G, p, root, comm)
5  comm_outer = Create_Comm_Between_Groups(G, p, root, comm)
6  comm_inner = Create_Comm_Inside_Groups(G, p, root_inner, comm)
   void* tmpbuf
7  MPI_Gather(sendbuf, sendcount, sendtype, tmpbuf, recvcount, recvtype,
   root, comm_outer)
8  MPI_Gather(tmpbuf, p/G *sendcount, sendtype, recvbuf, p/G *recvcount,
   recvtype, root_inner, comm_inner)
```

end

Algorithm 5: Hierarchical transformation of MPI gather operation.

3.3.1 Theoretical Analysis

3.3.1.1 Hierarchical Linear Scatter and Gather Algorithms

As we have seen the theoretical costs of the linear scatter and gather algorithms are equal. Therefore it is the same for their hierarchical transformations as well. If we use the formula 2.22 in both levels of the hierarchically transformed algorithms their cost will be as follows:

$$F(G) = \left(G + \frac{p}{G} - 2\right) \times (\alpha + m \times \beta) \tag{3.24}$$

The derivative of the $F(G)$ function is $\left(1 - \frac{p}{G^2}\right) \times (\alpha + m \times \beta)$. Therfore, the function gets its local minimum in $(1, p)$ at $G = \sqrt{p}$ and the minimum value of the function in this interval will be as follows:

$$(2 \times \sqrt{p} - 2) \times (\alpha + m \times \beta) \tag{3.25}$$

3.3.1.2 Hierarchical Linear with Synchronization Algorithm

The cost of the hierarchical linear with synchronization algorithm can be derived as below:

$$F(G) = 2 \times \alpha + \left(G + \frac{p}{G} - 2\right) \times (2 \times \alpha + m \times \beta) \tag{3.26}$$

From this formula we can find the optimal value of the $F(G)$ function for a fixed p and it will be as follows:

$$2 \times \alpha + (2 \times \sqrt{p} - 2) \times (2 \times \alpha + m \times \beta) \tag{3.27}$$

3.3.1.3 Hierarchical Binomial Gather and Scatter Algorithms

The hierarchical algorithms for binomial gather and scatter operations will have the same theoretical cost as the original algorithms have the same cost.

$$F(G) = \log_2 G \times \alpha + (G - 1) \times m \times \beta + \log_2 \frac{p}{G} \times \alpha + \left(\frac{p}{G} - 1\right) \times m \times \beta \tag{3.28}$$

It can be easily shown that the minium value of the $F(G)$ function in the inerval $(1, p)$ is the following:

$$\log_2 p \times \alpha + (2 \times \sqrt{p} - 2) \times m \times \beta \tag{3.29}$$

3.3.1.4 Hierarchical Minimum Spanning Tree Gather and Scatter Algorithms

The hierarchical transformation does not improve the performance of the MST scatter and gather algorithms. The performance cost of the MST algorithm for both operations is the same, thus the hierarchical transformations will have the same cost as well:

$$F(G) = \log_2 p \times \alpha + (2 - \frac{1}{G} - \frac{G}{p}) \times m \times \beta \tag{3.30}$$

This function attains its maximum in the interval $(1, p)$, and the minimums are achieved either when $G = 1$ or $G = \sqrt{p}$.

3.3.2 Experiments

The experimental study of the MPI gather and scatter operations was conducted on the Graphene cluster of Grid'5000 platform. We used two experimental settings, one process per core (i.e. four processes per node) and one process per node. Our study encompasses all the scatter and gather algorithms implemented in Open MPI version 1.8.4. The MPICH implementation contains only one algorithm, a binomial tree algorithm both for scatter and gather.

3.3.2.1 Experiments with Hierarchical Gather: One Process per Core Setting

We have studied all the mentioned gather and scatter algorithms using different sizes of messages and number of processes. The hierarchical linear with synchronization gather algorithm divides the message into a given number of segments. Therefore, we show experiments with different segment

sizes for this algorithm. Figure 3.31 presents the results with the total message sizes of 128MB and 256MB on 512 cores. In this case the message sizes per point-to-point operation are 256KB and 512KB respectively and the segment size is 4KB. The improvement in the best case can be up to 2.6 times when the total message size is 256MB and the segment size is 4KB. The trend is similar on a smaller number of processes. Figure 3.32 demonstrates experimental results on 64 cores for message sizes 128MB and 256MB with 4KB segments. Figure 3.33 presents the results with the same setting while changing the segment sizes to 32KB and 64KB. Experimental results with linear and binomial algorithms are given on Figure 3.36. In the case of binomial algorithm, there is no improvement. However the linear algorithm is improved by 30% when the message size is 128MB and by 17% when it is 256MB.

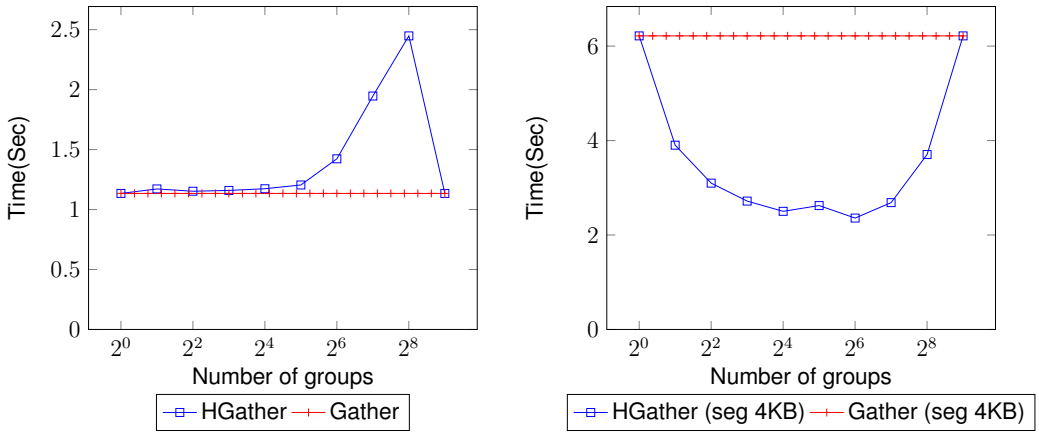


Figure 3.31: Hierarchical linear with synchronization gather algorithm on 512 cores. Total message size: 128MB (left) and 256MB (right). Message size per point-to-point communication: 256KB (left) and 512KB (right). Segment size: 4KB.

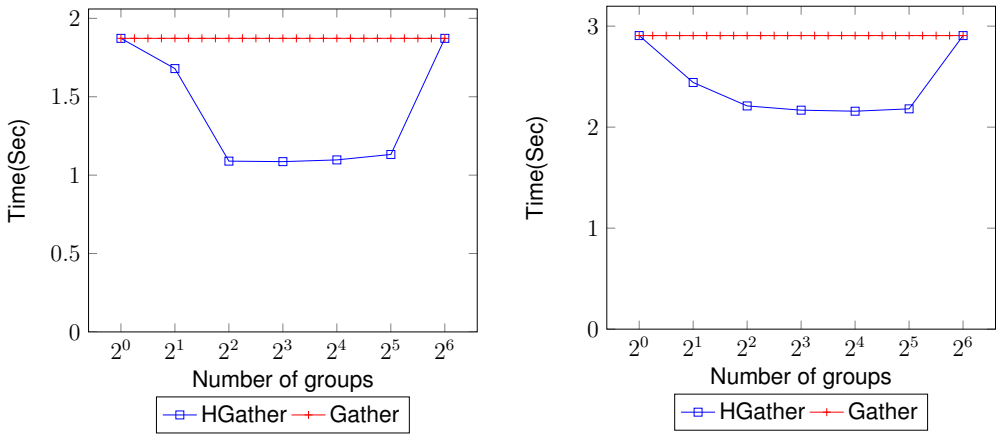


Figure 3.32: Hierarchical linear with synchronization gather algorithm on 64 cores. Total message size: 128MB (left) and 256MB (right). Message size per point-to-point communication: 2MB (left) and 4MB (right). Segment size: 4KB.

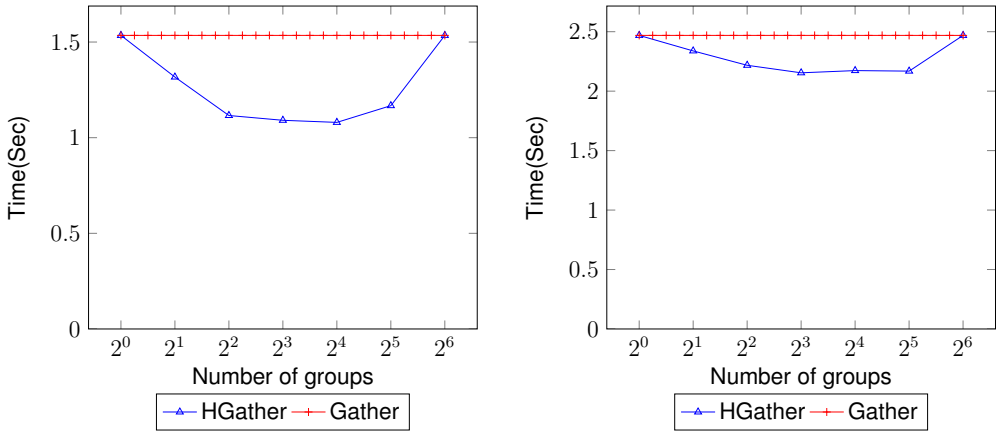


Figure 3.33: Hierarchical linear with synchronization gather algorithm on 64 cores. Total message size: 128MB (left) and 256MB (right). Message size per point-to-point communication: 2MB (left) and 4MB (right). Segment size: 32KB.

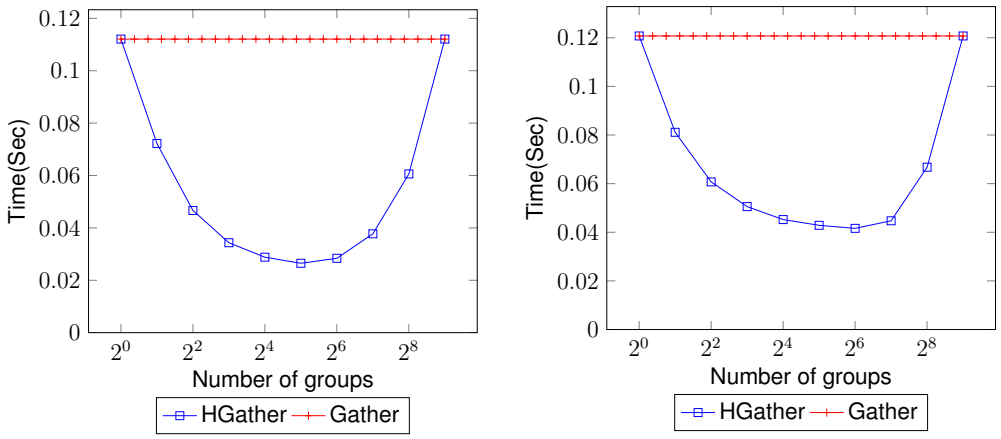


Figure 3.34: Hierarchical linear with synchronization gather algorithm on 512 cores. Total message size: 2MB (left) and 4MB (right). Message size per point-to-point communication: 4KB (left) and 8KB (right). Segment size: 1KB.

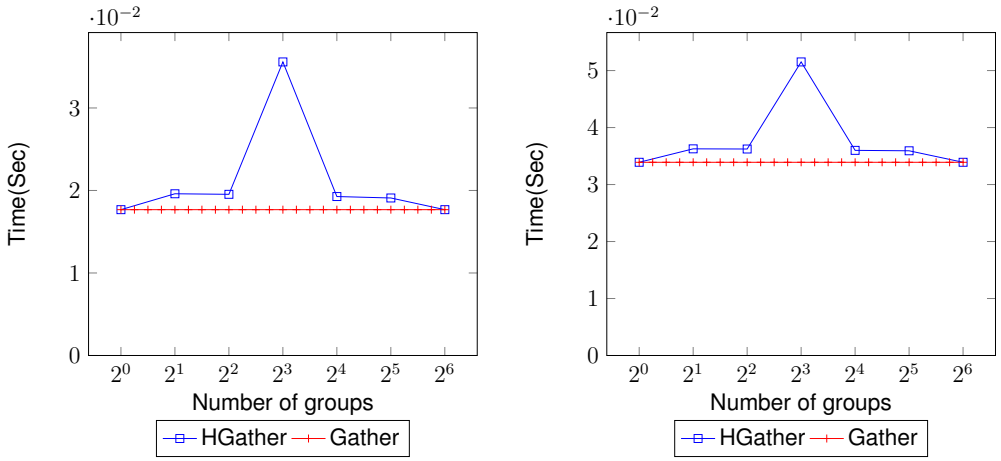


Figure 3.35: Hierarchical linear with synchronization gather algorithm on 64 cores. Total message size: 2MB (left) and 4MB (right). Message size per point-to-point communication: 32KB (left) and 64KB (right). Segment size: 1KB.

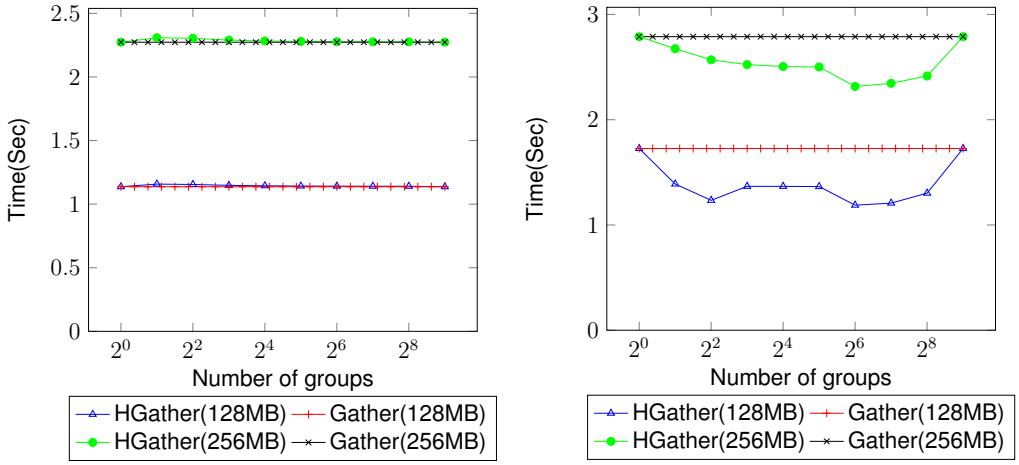


Figure 3.36: Hierarchical binomial (left) and linear (right) gather algorithms on 512 cores. Total message size: 128MB and 256MB. Message size per point-to-point communication: 32KB (left) and 64KB (right).

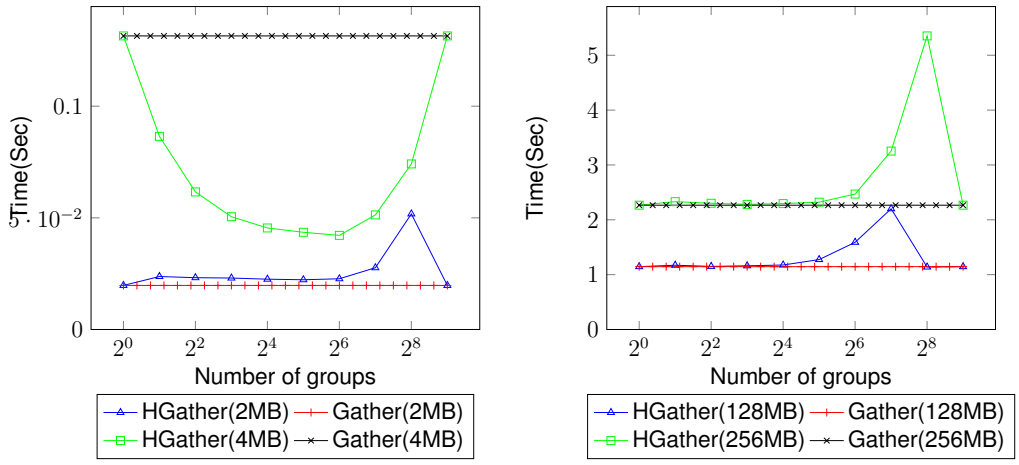


Figure 3.37: Hierarchical native Open MPI gather operation on 512 cores. Total message size: 2MB and 4MB (left), 128MB and 256MB (right). Message size per point-to-point communication: 4KB and 8KB (left) and 32KB and 64KB (right).

3.3.2.2 Experiments with Hierarchical Gather: One Process per Node Setting

The overall tendency with one process per node setting is quite similar to that of the one process per core setting. The performance of linear with

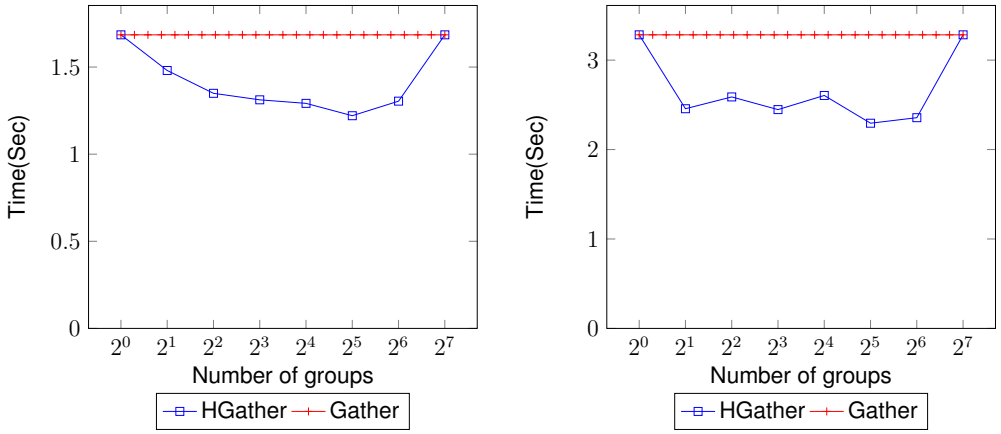


Figure 3.38: Hierarchical linear with synchronization gather algorithm on 128 nodes (one process per node). Total message size: 128MB (left) and 256MB (right). Message size per point-to-point communication: 1MB (left) and 2MB (right). Segment size: 1KB.

synchronization and simple linear algorithm is improved, while the binomial tree gather algorithm can not be improved with our hierarchical transformation. The improvement of the linear with synchronization algorithm on 128 nodes with a message size of 128MB and 1KB of segment size is 27.5% . The improvement with 256MB is 30% (Figure 3.38). Our observation shows that the segment size can change the behaviour significantly. For example, if we change the segment size in the experimental settings to 32KB and 64KB, we can see that despite there is more than 38% improvement in the first case, there is no improvement in the second case (Figure 3.39).

3.3.2.3 Experiments with Hierarchical Scatter: One Process per Core Setting

The hierarchical transformation improves scatter algorithms as well, however, the improvement is not significant as it was in the other MPI collective operations. In this section we present experimental study with linear, binomial and native Open MPI scatter operation. Figure 3.40 shows the results on 512 cores using messages of sizes 2MB, 4MB, 128MB and 256MB. The improvements are 10%, 17%, 24% and 24% respectively.

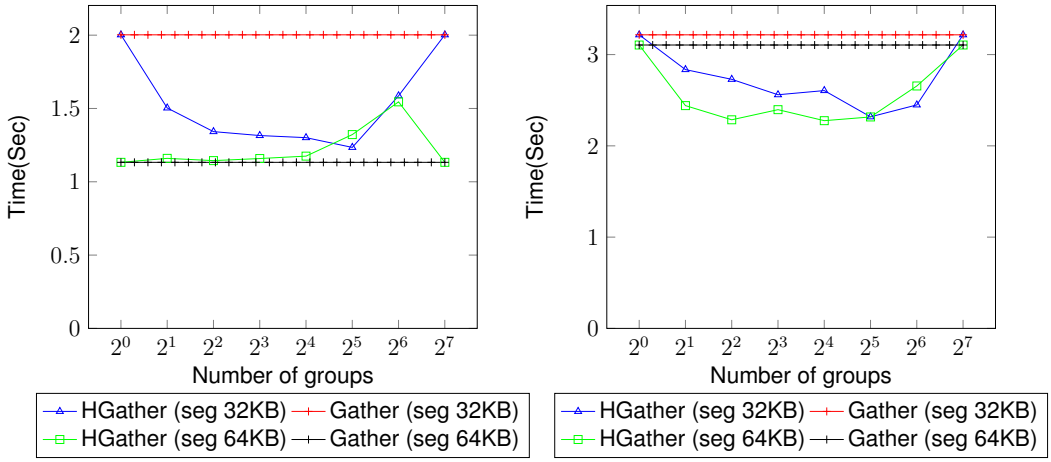


Figure 3.39: Hierarchical linear with synchronization gather algorithm on 128 nodes (one process per node). Total message size: 128MB (left) and 256MB (right). Message size per point-to-point communication: 1MB (left) and 2MB (right). Segment size: 32KB and 64KB.

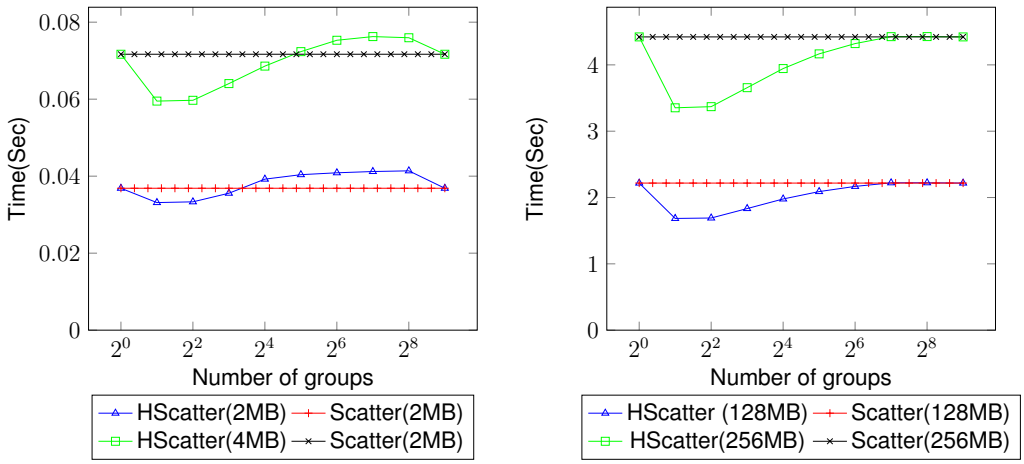


Figure 3.40: Hierarchical binomial scatter algorithm on 512 cores. Total message size: 2MB and 4 MB (left), 128MB and 256MB (right). Message size per point-to-point communication: 4KB and 8KB (left), 256KB and 512KB (right).

Figure 3.41 demonstrates experiments with the same setting as before but using linear scatter algorithm. This time the improvement with a message size of 2MB is 21%, there is about 9% improvement when the message is 4MB and 256MB. The performance does not improve when the message is

128MB. The results are the similar with the native scatter operation (see Figure 3.42). The same tendency continues on smaller number of processes. Figures 3.43 3.44 3.45 show experimental results on 64 processes using binomial, linear and native scatter operation respectively.

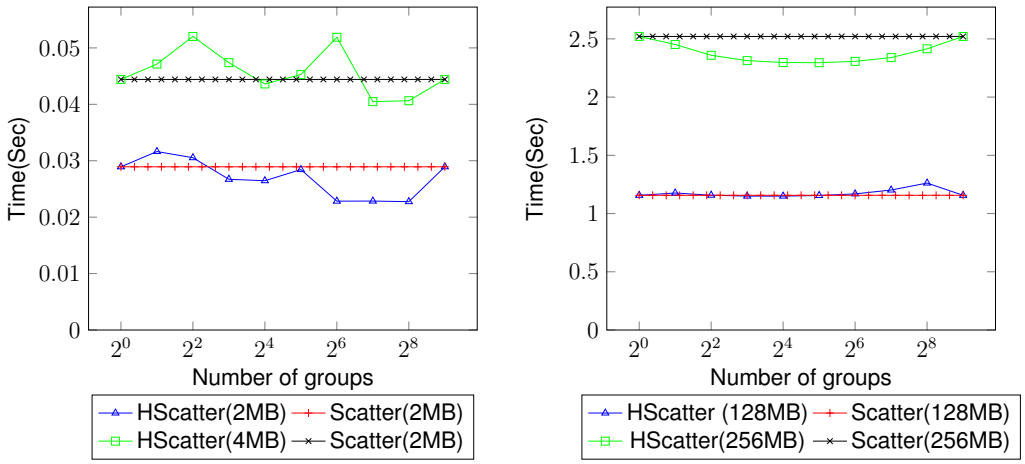


Figure 3.41: Hierarchical linear scatter algorithm on 512 cores. Total message size: 2MB and 4 MB (left), 128MB and 256MB (right). Message size per point-to-point communication: 4KB and 8KB (left), 256KB and 512KB (right).

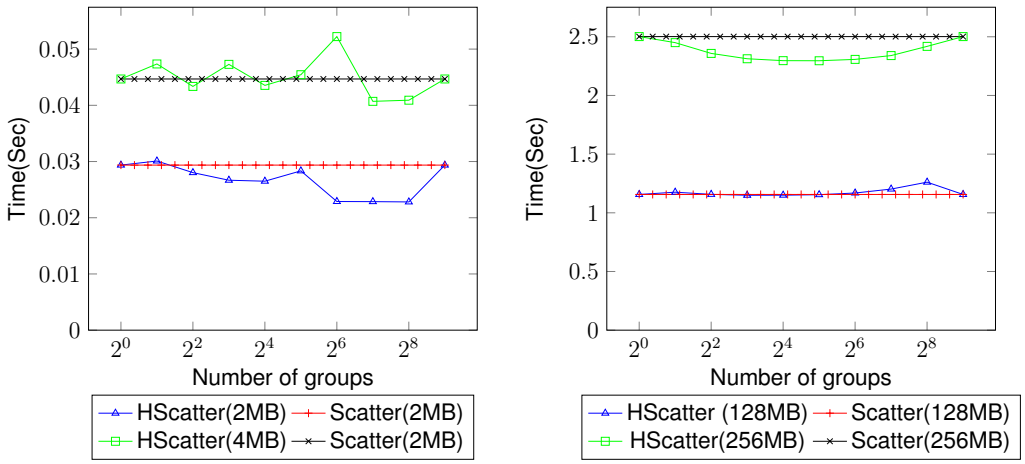


Figure 3.42: Hierarchical native Open MPI scatter operation on 512 cores. Total message size: 2MB and 4 MB (left), 128MB and 256MB (right). Message size per point-to-point communication: 4KB and 8KB (left), 256KB and 512KB (right).

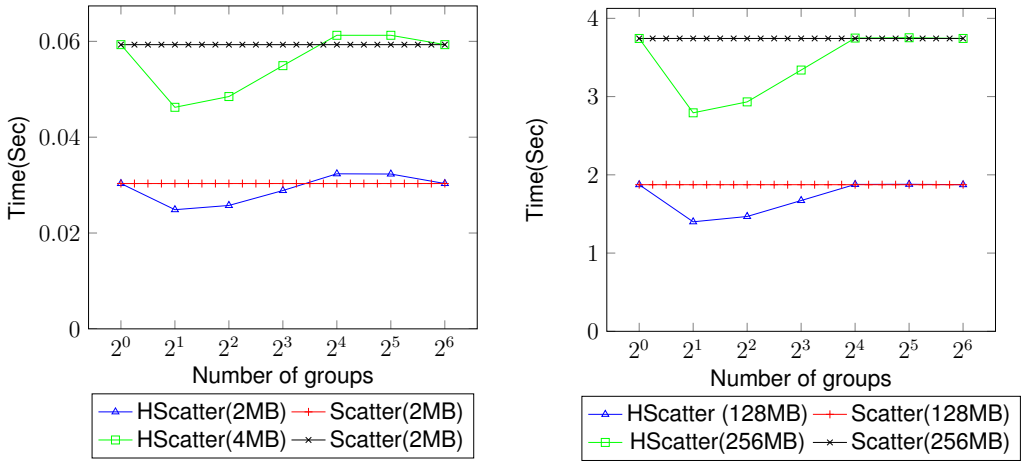


Figure 3.43: Hierarchical binomial scatter algorithm on 64 cores. Total message size: 2MB and 4 MB (left), 128MB and 256MB (right). Message size per point-to-point communication: 32KB and 64KB (left), 2MB and 4MB (right).

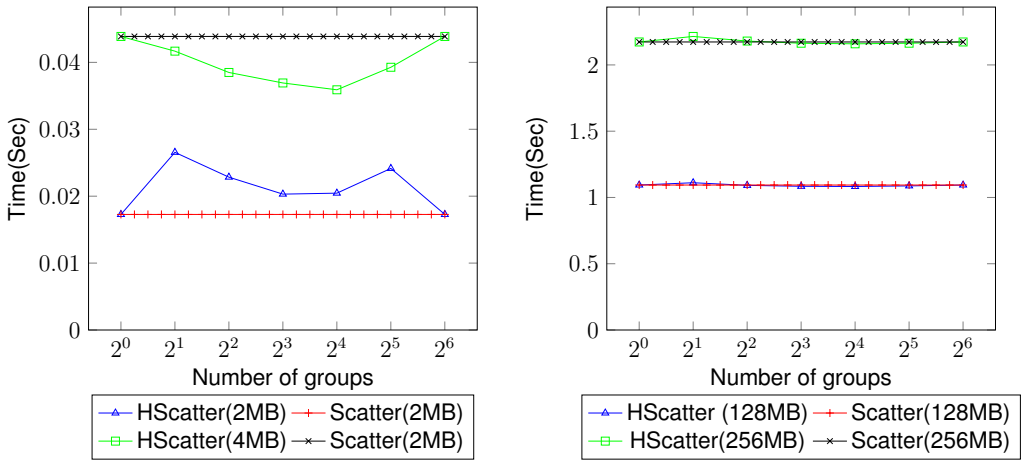


Figure 3.44: Hierarchical linear scatter algorithm on 64 cores. Total message size: 2MB and 4 MB (left), 128MB and 256MB (right). Message size per point-to-point communication: 32KB and 64KB (left), 2MB and 4MB (right).

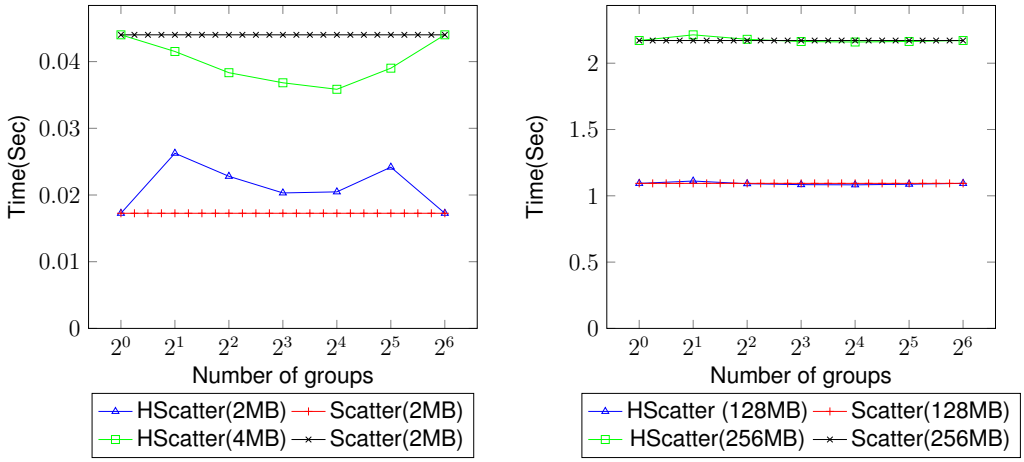


Figure 3.45: Hierarchical native Open MPI scatter operation on 64 cores. Total message size: 2MB and 4 MB (left), 128MB and 256MB (right). Message size per point-to-point communication: 32KB and 64KB (left), 2MB and 4MB (right).

3.3.3 Conclusion

This section has presented the topology-oblivious hierarchical optimization of MPI broadcast, reduce, allreduce, scatter and gather operations. The approach we propose is a traditional methodology widely used for dealing with the complexity of coordination and management of a large number of actors. According to this hierarchical technique, thousands or millions of actors are structured, and instead of interacting with a large number of peers, they coordinate their activities with one superior and a small number of peers and inferiors. This way the overhead of interaction is significantly reduced.

Most existing optimization methodologies are not general purpose because their main focus are specific applications, topologies or platforms. Therefore, such approaches have to rely on sophisticated analytical models and techniques to achieve highly accurate performance prediction. We try to balance the trade-off between the accuracy and generality of performance prediction of our optimization technique on a variety of platforms. In light of this goal, we mainly focus on the scale of the platform while ignoring its complexity by employing the simplest communication performance model in our optimization techniques. The in-detail experimental study with its theoretical basis shows that our hierarchical approach can bring multi-fold performance gains.

In the next section we study optimization of a state-of-the-art parallel matrix multiplication algorithm using the same approach we demonstrated in this section.

Chapter 4

Applications

4.1 Parallel Matrix Multiplication

In this chapter we demonstrate how the hierarchical approach can be applied to optimization of the execution of parallel matrix-matrix multiplication on large-scale HPC platforms. We choose matrix multiplication for two reasons. First of all, it is important in its own right as a computational kernel of many scientific applications. Secondly, it is a popular representative for other scientific applications. It is widely accepted that if an optimization method works well for matrix multiplication, it will also work well for many other scientific applications.

The contributions of this chapter are as follows:

- We introduce a new design to parallel matrix multiplication algorithm by introducing a two-level virtual hierarchy into the two-dimensional arrangement of processors. We apply our approach to the SUMMA algorithm [60], which is a state of the art algorithm.
- We theoretically prove that hierarchical SUMMA (HSUMMA) reduces the communication cost of SUMMA and then provide experimental results on a cluster of Grid'5000 and BlueGene/P, which are reasonably representative and span a good spectrum of loosely and tightly coupled platforms. We compare HSUMMA with SUMMA because it is the most general and scalable parallel matrix multiplication algorithm, which

decreases its per-processor memory footprint with the increase of the number of processors for a given problem size, and is widely used in modern parallel numerical linear algebra packages such as ScaLAPACK. In addition, because of its practicality SUMMA is used as a starting point for implementation of parallel matrix multiplication on specific platforms. As a matter of fact, the most used parallel matrix multiplication algorithms for heterogeneous platforms [110] [9] are based on SUMMA as well. Therefore, despite being introduced in the mid-1990s, SUMMA is still a state-of-the-art algorithm.

- Despite the study presented here has been conducted in the context of parallel matrix multiplication, according to the previous chapters the proposed optimization technique is not application-bound and can be applied to other parallel applications intensively using collective communication operations such as broadcast, reduction and scatter/gather operations.

4.1.1 Serial Matrix Multiplication Optimization

Matrix multiplication is a very important kernel in many numerical linear algebra algorithms and is one of the most studied problems in high-performance computing. Different approaches have been proposed to optimize matrix multiplication through the improvement of spatial and temporal locality. Blocking (or tiling) [111] is one such basic technique. Despite its generality, blocking is architecture dependent. Cache-oblivious algorithms [112], on the other hand, offer an architecture independent alternative to the blocked algorithms by using the divide-and-conquer paradigm. However, a recent study [113] shows that even highly optimized cache-oblivious programs perform considerably slower than their cache-conscious (i.e. based on blocking) counterparts. A related idea to the cache-oblivious methods is to use a recursive structure for the matrices [114]. However, traditional implementations of the Basic Linear Algebra Subroutines (BLAS) library [115] are mainly based on the blocking approach and thus need optimization on a specific hardware platform. Therefore, automatic

optimization of matrix multiplication on a range of platforms has been an active research area. One such example is ATLAS [116] which provides C and Fortran77 interfaces to a portably efficient BLAS implementation and automatically generates optimized numerical software for a given processor architecture as a part of the software installation process. The GotoBLAS [117] library offers another high-performance implementation of matrix multiplication for a variety of architectures.

4.1.2 Parallel Matrix Multiplication Optimization

Parallel matrix multiplication has also been thoroughly investigated over the past three decades. As a result, many parallel matrix multiplication algorithms have been developed for distributed memory, shared memory and hybrid platforms. In this section, we only outline the algorithms designed for distributed memory platforms.

Cannon's algorithm [118], which was introduced in 1967, was the first efficient algorithm for parallel matrix multiplication providing theoretically optimal communication cost. However, this algorithm requires a square number of processors which makes it impossible to be used in a general purpose library. Fox's algorithm [119], which was introduced later, has the same restriction. PUMMA [120] and BiMMeR [121] extend Fox's algorithm for a general 2-D processor grid by using block-cyclic data distribution and torus-wrap data layout respectively.

The 3D algorithm [122], which dates back to the 1990s, organizes the p processors as $p^{\frac{1}{3}} \times p^{\frac{1}{3}} \times p^{\frac{1}{3}}$ 3D mesh and achieves a factor of $p^{\frac{1}{6}}$ less communication cost than 2D parallel matrix multiplication algorithms. However, in order to get this improvement the 3D algorithm requires $p^{\frac{1}{3}}$ extra copies of the matrices. That means that on one million cores the $3D$ algorithm will require 100 extra copies of the matrices which would be a significant problem on some platforms. Therefore, the 3D algorithm is only practical for relatively small matrices.

Another method to improve the performance of parallel matrix multiplication is overlapping communication and computation. That approach

was introduced by Agarwal et al. [123] in 1994 and according to the authors this optimization hides almost all of the communication cost with the computation for larger matrices.

In the mid-1990s, SUMMA [60] was introduced for a general $P \times Q$ processor grid. Like PUMMA and BiMMeR, SUMMA also solves the difficulty of Cannon's and Fox's algorithms and perfectly balances the computation load. However, SUMMA is simpler, more general and more efficient than the previous algorithms. For these reasons, it is used in ScaLAPACK [124], the most popular parallel numerical linear algebra package. The implementation of SUMMA in ScaLAPACK uses block-cyclic distribution and a modified communication scheme to overlap the communication and computation. The version of SUMMA modified this way was introduced as DIMMA [125]. Depending on the shape of the processor grid and matrix size, the performance of DIMMA can be better or worse than that of SUMMA. In its best case, the performance improvement of DIMMA over SUMMA was 10% on Intel Paragon [125].

A more recent algorithm, SRUMMA [126], was proposed in 2004 and has algorithmic efficiency equivalent to that of Cannon's algorithm on clusters and shared memory systems. This algorithm uses block-checkerboard distribution of the matrices and overlaps communication with computations by using remote memory access (RMA) communication rather than message passing.

A recently introduced 2.5D algorithm [127] generalizes the 3D algorithm by parameterizing the extent of the third dimension of the processor arrangement: $\frac{p}{c}^{\frac{1}{2}} \times \frac{p}{c}^{\frac{1}{2}} \times c$, $c \in [1, p^{\frac{1}{3}}]$. While reducing the memory footprint compared with the 3D algorithm, it will still be efficient only if there is free amount of extra memory to store c copies of the matrices. At the same time, it is expected that exascale systems will have a dramatically shrinking memory space per core [128]. Therefore, the 2.5D algorithm cannot be scalable on future exascale systems.

4.1.3 SUMMA Algorithm

SUMMA [60] implements the matrix multiplication $C = A \times B$ over a two-dimensional $p = s \times t$ processor grid. For simplicity, we assume that the matrices are square $n \times n$ matrices. These matrices are distributed over the processor grid by block-checkerboard distribution.

We can see the size of the matrices as $\frac{n}{b} \times \frac{n}{b}$ by introducing a block of size b. Then each element in A, B, and C is a square $b \times b$ block, and the algorithm operates on blocks rather than on scalar elements. For simplicity, we assume that n is a multiple of b. SUMMA can be formulated as follows: The algorithm consists of $\frac{n}{b}$ steps. At each step

- Each processor holding part of the pivot column of the matrix A horizontally broadcasts its part of the pivot column along the processor row.
- Each processor holding part of the pivot row of the matrix B vertically broadcasts its part of the pivot row along the processor column.
- Each processor updates each block in its C rectangle with one block from the pivot column and one block from the pivot row, so that each block $c_{ij}, (i,j) \in (1, ..., \frac{n}{b})$ of matrix C will be updated as $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$.
- After $\frac{n}{b}$ steps of the algorithm, each block c_{ij} of matrix C will be equal to $\sum_{k=1}^{\frac{n}{b}} a_{ik} \times b_{kj}$

Figure 4.1 shows the communication patterns in SUMMA on 6×6 processors grid.

4.1.3.1 SUMMA Algorithm on Heterogeneous Platforms

On heterogeneous platforms the processors run at different speeds. Therefore, the traditional homogeneous algorithms can not be efficient as they distribute data equally to every processor in the system. In order to use the heterogeneity efficiently the input data should be partitioned according to

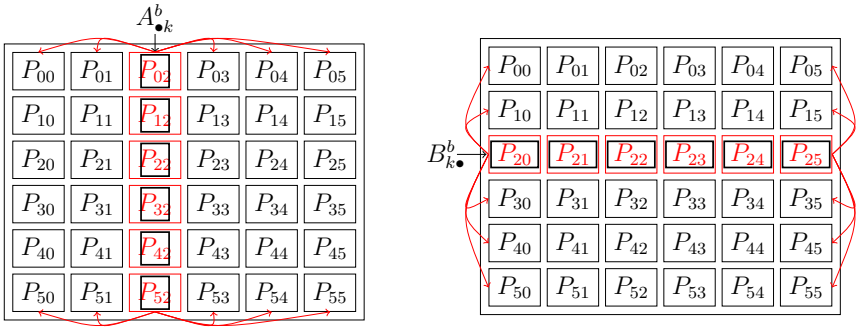


Figure 4.1: Horizontal communications of matrix A and vertical communications of matrix B in SUMMA. The pivot column $A^b_{\bullet k}$ of $\frac{n}{\sqrt{P}} \times b$ blocks of matrix A is broadcast horizontally. The pivot row $B^b_{k\bullet}$ of $b \times \frac{n}{\sqrt{P}}$ blocks of matrix B is broadcast vertically.

the processors speed. The simplest performance model, capturing this feature and abstracting from the others, sees a heterogeneous network of computers as a set of interconnected processors, each of which is characterized by a single positive constant representing its speed. Two main parameters of this model include:

- p, the number of processors
- S = s_1, s_2, ..., s_p, the speeds of the processors.

The speed of the processors can be either absolute or relative. The absolute speed of the processors is the number of computational units performed by the processor per one time unit. The relative speed of the processor can be obtained by the normalization of its absolute speed so that $\sum_{i=1}^{p} s_i = 1$.

Despite this performance model does not have any parameter to describe communication network, it has been proved that the communication cost of parallel matrix algorithms can be taken into account by using this simple model [9]. Beaumont [110] proposed an algorithm which finds an optimal distribution for n independent units of computation over p processors of speeds s_1, ...s_p. Modifications of SUMMA for heterogeneous platforms typically use the following general design [129].

- Matrices A, B, and C are identically partitioned into equal rectangular generalized blocks
- The generalized blocks are identically partitioned into rectangles so that
  - There is a one-to-one mapping between the rectangles and the processors
  - The area of each rectangle is (approximately) proportional to the speed of the processor that has the rectangle.
- Then, the algorithm follows the steps of its homogeneous prototype; namely, at each step
  - The pivot column of $b \times b$ blocks of matrix A is broadcast horizontally
  - The pivot row of $b \times b$ blocks of matrix B is broadcast vertically
  - Each processor updates each block in its C partition with one block from the pivot column and one block from the pivot row.

It is obvious that, on the heterogeneous platforms, the efficiency of the algorithms highly depends on the data partitioning as it affects the communication cost.

A vast amount of research have been conducted into data partitioning on heterogeneous platforms to minimize communication cost such as in [110][130]. The studies in [131][132][133] focus on data partitioning on multicore and multi-GPU systems using functional performance models [134][135]. In [136][137] the authors demonstrate that on real-life heterogeneous platforms optimal non-rectangular partitioning can significantly outperform the traditional optimal rectangular one.

4.2 Hierarchical SUMMA

Let $p = s \times t$ processors be distributed over the same two-dimensional virtual processor grid as in SUMMA, the matrices be square $n \times n$ matrices, b be the block size. Let the distribution of the matrices be the same as in SUMMA.

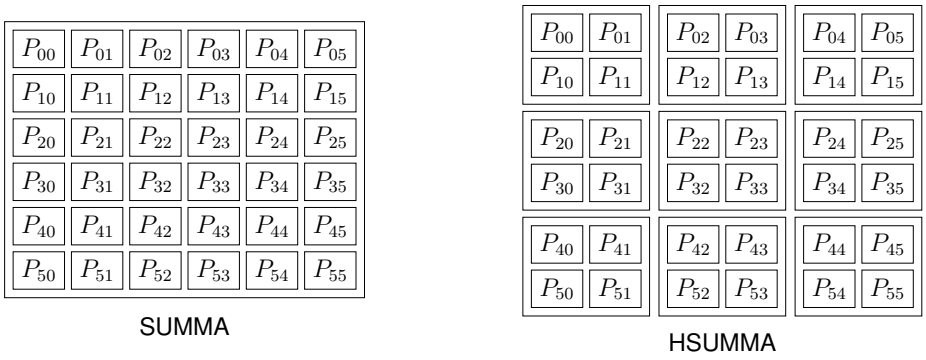


Figure 4.2: SUMMA and HSUMMA. HSUMMA groups 6×6 *processors into* 3×3 *groups,* 2×2 *processors per group.*

Hierarchical SUMMA (HSUMMA) partitions the virtual $s \times t$ processor grid into a higher level $I \times J$ arrangement of rectangular groups of processors, so that inside each group there will be a two-dimensional $\frac{s}{I} \times \frac{t}{J}$ grid of processors. Figure 4.2 compares the arrangement of processors in SUMMA with HSUMMA. In this example, a 6×6 grid of processors is arranged into two-level 3×3 grids of groups and 2×2 grid of processors inside a group.

Let $P_{(x,y)(i,j)}$ denote the processor (i,j) inside the group (x,y). HSUMMA splits the communication phase of the SUMMA algorithm into two phases and consists of $\frac{n}{b}$ steps. The pseudocode for HSUMMA is Algorithm 6 and it can be summarized as follows:

- Horizontal broadcast of the pivot column of the matrix A is performed as follows:

    1. First, each processor $P_{(k,y)(i,j)}, k \in (1, ..., I)$ holding part of the pivot column of the matrix A horizontally broadcasts its part of the pivot column to the processors $P_{(k,z)(i,j)}$, $z{\neq}y, z \in (1, ..., I)$ in the other groups. (Line 6 - 9)
    2. Now, inside each group (x, y) processor $P_{(x,y)(i,j)}$ has the required part of the pivot column of the matrix A and it further horizontally broadcasts it to the processors $P_{(x,y)(i,c)}$, $c{\neq}j, c \in (1, ..., \frac{s}{I})$ inside the group. (Line 15 - 17)

- Vertical broadcast of the pivot row of the matrix B is performed as follows:

  1. First, each processor $P_{(x,k)(i,j)}, k \in (1, ..., I)$ holding part of the pivot row of the matrix B vertically broadcasts its part of the pivot row to the processors $P_{(z,k)(i,j)}$, $z \neq k, z \in (1, ..., I)$ in the other groups. (Line 10 - 13)
  2. Now, inside each group (x, y) processor $P_{(x,y)(i,j)}$ has the required part of the pivot row of the matrix B and it further vertically broadcast it to the processors $P_{(x,y)(r,j)}$, $r \neq j, r \in (1, ..., \frac{t}{J})$ inside the group. (Line 18 - 20)

- Each processor inside a group updates each block in its C rectangle with one block from the pivot column and one block from the pivot row, so that each block $c_{ij}, (i, j) \in (1, ..., \frac{n}{b})$ of matrix C will be updated as $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$. (Line 21)

- After $\frac{n}{b}$ steps (Line 5) of the algorithm, each block c_{ij} of matrix C will be equal to $\sum_{k=1}^{\frac{n}{b}} a_{ik} \times b_{kj}$

It is assumed that only one broadcast algorithm is used in all the steps of the algorithm and there is no barrier between the communications at the hierarchies. The communication phases described above are illustrated in Figure 4.3 and Figure 4.4. In general the block size between groups, M, and the block size inside a group, b, are different. In this case the size of sent data between the groups is at least the same as the size of data sent inside a group. Apparently, $b \leq M$. Then, the number of steps at the higher level will be equal to the number of blocks between groups: $\frac{n}{M}$. In each iteration between the groups, the number of steps inside a group will be $\frac{M}{b}$, so the total number of steps of HSUMMA, $\frac{n}{M} \times \frac{M}{b}$, will be the same as the number of steps of SUMMA. The amount of data sent will be also the same as in SUMMA.

In addition, SUMMA is a special case of HSUMMA. Indeed, when the number of groups, G, is equal to one or to the total number of processors, p, HSUMMA and SUMMA become equivalent. This means that even if there appears a highly efficient broadcast algorithm, the use of which makes

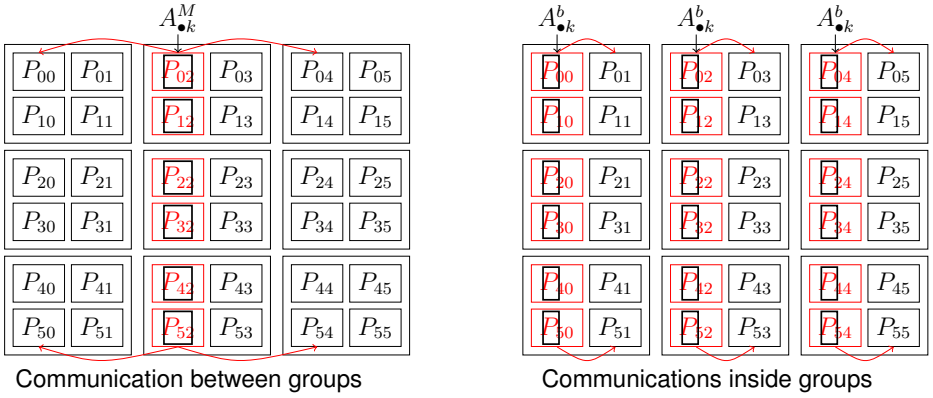


Figure 4.3: Horizontal communications of matrix A in HSUMMA. The pivot column $A^M_{\bullet k}$ of $\frac{n}{\sqrt{P}} \times M$ blocks of matrix A is broadcast horizontally between groups. Upon receipt of the pivot column data from the other groups, the local pivot column $A^b_{\bullet k}, (b \leq M)$ of $\frac{n}{\sqrt{P}} \times b$ blocks of matrix A is broadcast horizontally inside each group.

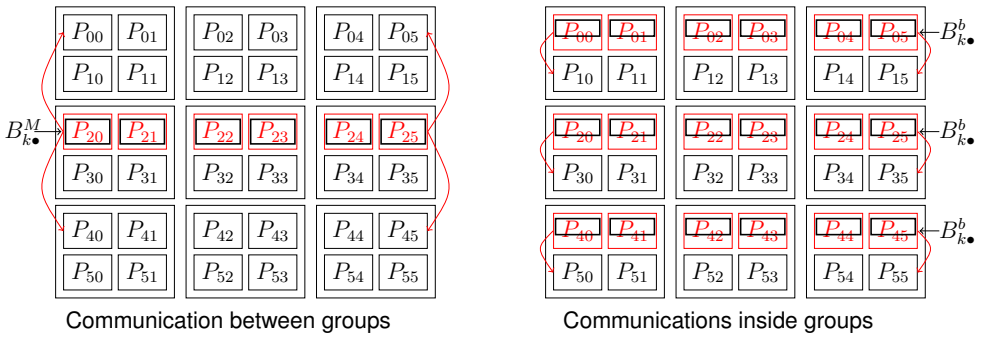


Figure 4.4: Vertical communications of matrix B in HSUMMA. The pivot row $B^M_{k\bullet}$ of $M \times \frac{n}{\sqrt{P}}$ blocks of matrix B is broadcast vertically between groups. Upon receipt of the pivot row data from the other groups, the local pivot row $B^b_{k\bullet}$ of $b \times \frac{n}{\sqrt{P}}, (b \leq M)$ blocks of matrix B is broadcast vertically inside each group.

SUMMA outperform HSUMMA for any $G \in (1, p)$, we should just use HSUMMA with $G = 1$.

4.2.1 Theoretical Analysis

4.2.1.1 Analysis of SUMMA

Let the $n \times n$ matrices be distributed over a two-dimensional $\sqrt{p} \times \sqrt{p}$ grid of processors and let the block size be b. After distributing the matrices over the processors grid each processor will have a $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ part of the matrices. This algorithm has $\frac{n}{b}$ steps. In each step, the processors broadcast a pivot row of matrix B and a pivot column of matrix A. In our analysis, we assume that these two communications steps are serialized. The computation cost of one step is $O(2 \times \frac{n^2}{p} \times b)$. Hence, the overall computation cost will be $O(\frac{2n^3}{p})$.

For this analysis the network congestion is neglected. Broadcasting a pivot row (column) is broken down into a set of parallel broadcasts along the processor columns (rows). The size of data transferred by each such individual broadcast is $\frac{n}{\sqrt{p}} \times b$. The total communication cost of SUMMA can be computed by multiplying the communication cost of each step by the number of steps depending on the broadcast algorithm.

- The communication cost of broadcasting a pivot row or a pivot column with the pipelined linear tree broadcast in one step will be as follows:
  $\left(X + \sqrt{p} - 1\right) \times \left(\alpha + \beta \times \frac{n}{\sqrt{p}X} \times b\right)$
- The communication cost of broadcasting a pivot row or a pivot column with the scatter-algather broadcast in one step will be as follows:
  $(\log_2\left(\sqrt{p}\right) + \sqrt{p} - 1) \times \alpha + 2(1 - \frac{1}{\sqrt{p}})\beta \times \frac{n}{\sqrt{p}} \times b$

If we sum the costs of the vertical and horizontal communications, and take into account that there are $\frac{n}{b}$ steps in total, then the overall communication costs will be as follows:

- Communication cost of SUMMA with the pipelined linear tree broadcast:
  $2\left(X + \sqrt{p} - 1\right) \times \left(\alpha \times \frac{n}{b} + \beta \times \frac{n^2}{\sqrt{p}X}\right)$

```
/*The A,B,C matrices are distributed on a virtual 2-D grid of
p = s×t processors.
Here are the instructions executed by the processor P(x,y)(i,j)
(this is the processor (i,j) inside the group (x,y)).*/
Data: NB_Block_Group: Number of steps in the higher level
Data: NB_Block_Inside: Number of steps in the lower level
Data: (M, L, N): Matrix dimensions
Data: A, B: two input sub-matrices of size (M/s × L/t, L/s × N/t)
Result: C: result sub-matrix of size M/s × N/t
begin
1   MPI_Comm group_col_comm        /* communicator between P(*,y)(i,j)
    processors */
2   MPI_Comm group_row_comm        /* communicator between P(x,*)(i,j)
    processors */
3   MPI_Comm col_comm     /* communicator of P(x,y)(*,j) processors */
4   MPI_Comm row_comm     /* communicator of P(x,y)(i,*) processors */
5   for iter_group = 0; iter_group < NB_Block_Group; iter_group++ do
6       if i == Pivot_inside_group_col(iter_group) then
7           if x == Pivot_group_col(iter_group) then
8               Copy_Block_group( Block_group_A, A, iter_group )
            end
9           MPI_Bcast(Block_group_A, Type_Block_group_A,
            Pivot_group_col(iter_group), group_row_comm)
        end
10      if j == Pivot_inside_group_row(iter_group) then
11          if y == Pivot_group_row(iter_group) then
12              Copy_Block_group( Block_group_B, B, iter_group )
            end
13          MPI_Bcast(Block_group_B, Type_Block_group_B,
            Pivot_group_row(iter_group), group_col_comm)
        end
14      for iter = 0; iter < NB_Block_Inside; iter++ do
15          if i == Pivot_inside_group_col(iter) then
16              Copy_Block_A( Block_A, Block_group_A, iter)
            end
17          MPI_Bcast(Block_A, Type_Block_A, Pivot_col(iter), row_comm)

18          if j == Pivot_inside_group_row(iter) then
19              Copy_Block_B( Block_B, Block_group_B, iter)
            end
20          MPI_Bcast(Block_B, Type_Block_B, Pivot_row(iter), col_comm)

21          DGemm(Block_A, Block_B, C)
        end
    end
end
```

Algorithm 6: Hierarchical SUMMA algorithm.

- Communication cost of SUMMA with the scatter-allgather broadcast: $(\log_2(p) + 2(\sqrt{p}-1))\alpha \times \frac{n}{b} + 4(1-\frac{1}{\sqrt{p}})\beta \times \frac{n^2}{\sqrt{p}}$

4.2.1.2 Analysis of HSUMMA

To simplify the analysis, let us assume that there are G groups arranged as a $\sqrt{G} \times \sqrt{G}$ grid of processors groups. Let M denote the block size between groups (we also call such a block an outer block), b be the block size inside a group, and $n \times n$ be the size of the matrices.

HSUMMA has two communication phases: communication between groups (i.e. outer communication) and inside groups (i.e. inner communication). The outer communication phase has $\frac{n}{M}$ steps which are called outer steps. Each outer block belongs to $\sqrt{p}$ processors. Thus, in one outer step each processor, which owns a part of the pivot column, horizontally broadcasts this part (of size $\frac{n \times M}{\sqrt{p}}$) to $\sqrt{G}$ processors. Similarly, each processor, owning a part of the pivot row, will vertically broadcast its part (of size $\frac{n \times M}{\sqrt{p}}$) to $\sqrt{G}$ processors.

Inside one group, processors are arranged in a grid of size $\frac{\sqrt{p}}{\sqrt{G}} \times \frac{\sqrt{p}}{\sqrt{G}}$. Upon the receipt of the outer block, in the same way horizontal and vertical brodcasts are performed inside each group. The communications inside different groups happen in parallel as they are completely independent of each other. Inside a group there will be $\frac{M}{b}$ steps which we call inner steps. In each inner step, a data block of matrix A of size $\frac{n \times b}{\sqrt{p}}$ is broadcast horizontally to $\frac{\sqrt{p}}{\sqrt{G}}$ processors, and a data block of matrix B of size $\frac{n \times b}{\sqrt{p}}$ is broadcast vertically to $\frac{\sqrt{p}}{\sqrt{G}}$ processors. Upon the receipt of the required data, each processor updates its result by using a dgemm routine.

The total number of steps is $\frac{n}{b}$, and the overall computation cost again will be $O(\frac{2n^3}{p})$ as the computation cost in one inner step is $O(2 \times \frac{n^2}{p} \times b)$.

The overall communication cost inside a group will be the sum of the horizontal and vertical communication costs inside the group, multiplied by the number of inner steps. In the same way, the overall communication cost between the groups will be equal to the sum of the horizontal and vertical communication costs between the groups, multiplied by the number of outer

steps. The total communication cost of HSUMMA will be the sum of the overall inner and outer communication costs. If we put the corresponding amount of communicated data and the number of communicating processors in the formulas for the costs of the pipelined linear tree algorithm and the scatter-allgather algorithm, the resulting communication costs will be as follows:

- Inner Communication cost (inside groups):
  - Pipelined Linear Tree:
    $2\left(X+\sqrt{\frac{p}{G}}-1\right)\times\left(\alpha\times\frac{n}{b}+\beta\times\frac{n^2}{\sqrt{p}X}\right)$
  - Scatter-allgather broadcast:
    $\left(\log_2\left(\frac{p}{G}\right)+2\left(\frac{\sqrt{p}}{\sqrt{G}}-1\right)\right)\times\alpha\times\frac{n}{b}+4(1-\frac{\sqrt{G}}{\sqrt{p}})\times\frac{n^2}{\sqrt{p}}\beta$
- Outer Communication cost (between groups):
  - Pipelined Linear Tree:
    $2\left(X+\sqrt{G}-1\right)\times\left(\alpha\times\frac{n}{M}+\beta\times\frac{n^2}{\sqrt{p}X}\right)$
  - Scatter-allgather broadcast:
    $\left(\log_2\left(G\right)+2\left(\sqrt{G}-1\right)\right)\times\alpha\times\frac{n}{M}+4(1-\frac{1}{\sqrt{G}})\times\frac{n^2}{\sqrt{p}}\beta$

4.2.1.3 Theoretical Prediction

One of the goals of this section is to demonstrate that independent of the broadcast algorithm employed by SUMMA, HSUMMA will either outperform SUMMA, or be at least equally fast. This section introduces a general model for broadcast algorithms, and theoretically predicts SUMMA and HSUMMA. In the model we assume no contention and assume all the links are homogeneous. We show that even this simple model can predict the extremums of the communication cost function.

Again, we assume that the time taken to send a message of size m between any two processors is modeled as $T(m)=\alpha+m\times\beta$, where α is the latency and β is the reciprocal bandwidth.

We model a broadcast time for a message of size m among p processors by formula (4.1). This model generalizes all homogeneous broadcast

algorithms, such as flat, binary, binomial, linear, and scatter/allgather broadcast algorithms:

$$T_{bcast}(m,p) = L(p)\times\alpha + m\times W(p)\times\beta \tag{4.1}$$

In (4.1) we assume that $L(1) = 0$ and $W(1) = 0$. It is also assumed that $L(p)$ and $W(p)$ are monotonic and differentiable functions in the interval $(1, p)$ and their first derivatives are constants or monotonic in the interval $(1, p)$.

By using this general broadcast model the communication cost of HSUMMA can be expressed as a sum of the latency cost and the bandwidth cost:

$$T_{HS}(n,p,G) = T_{HS_l}(n,p,G) + T_{HS_b}(n,p,G) \tag{4.2}$$

Here $G \in [1, p]$ and $b{\leq}M$. The latency cost $T_{HS_l}(n,p,G)$ and the bandwidth cost $T_{HS_b}(n,p,G)$ will be given by the following formulas:

$$T_{HS_l}(n,p,G) = 2n\left(\frac{1}{M}\times L(\sqrt{G}) + \frac{1}{b}\times L(\frac{\sqrt{p}}{\sqrt{G}})\right)\alpha \tag{4.3}$$

$$T_{HS_b}(n,p,G) = 2\frac{n^2}{\sqrt{p}}\times\left(W(\sqrt{G}) + W(\frac{\sqrt{p}}{\sqrt{G}})\right)\beta \tag{4.4}$$

If we take $b = M$ the latency cost $T_{HS_l}(n,p,G)$ changes and becomes as follows:

$$T_{HS_l}(n,p,G) = 2n\left(\frac{1}{M}\times L(\sqrt{G}) + \frac{1}{M}\times L(\frac{\sqrt{p}}{\sqrt{G}})\right)\alpha \tag{4.5}$$

However, the bandwidth cost will not change as it does not depend on the block sizes.

The comparison of Formula 4.3 and Formula 4.5 suggests that with decrease of b the latency cost will increase. This means that $b = M$ will be the optimal value for b. We will validate this prediction in the experimental part. Therefore, in the following analysis we take $M = b$.

It is clear that $T_S(n,p)$ (i.e. SUMMA) is a special case of $T_{HS}(n,p,G)$ (i.e. HSUMMA) when $G = 1$ or $G = p$.

Let us investigate extremums of T_{HS} as a function of G for a fixed p and n.

Then, for $M = b$ we can get the following derivatives:

$$\frac{\partial T_{HS}}{\partial G} = \frac{n}{b} \times L_1(p, G)\alpha + \frac{n^2}{\sqrt{p}} \times W_1(p, G)\beta \tag{4.6}$$

Here, $L_1(p, G)$ and $W_1(p, G)$ are defined as follows:

$$L_1(p, G) = \left(\frac{\partial L(\sqrt{G})}{\partial \sqrt{G}} \times \frac{1}{\sqrt{G}} - \frac{\partial L(\frac{\sqrt{p}}{\sqrt{G}})}{\partial \frac{\sqrt{p}}{\sqrt{G}}} \times \frac{\sqrt{p}}{G\sqrt{G}} \right) \tag{4.7}$$

$$W_1(p, G) = \left(\frac{\partial W(\sqrt{G})}{\partial \sqrt{G}} \times \frac{1}{\sqrt{G}} - \frac{\partial W(\frac{\sqrt{p}}{\sqrt{G}})}{\partial \frac{\sqrt{p}}{\sqrt{G}}} \times \frac{\sqrt{p}}{G\sqrt{G}} \right) \tag{4.8}$$

It can be easily shown that, if $G = \sqrt{p}$ then $L_1(p, G) = 0$ and $W_1(p, G) = 0$, thus, $\frac{\partial T_{HS}}{\partial G} = 0$. In addition, $\frac{\partial T_{HS}}{\partial G}$ changes the sign in the interval $(1, p)$ depending on the value of G. That means that $T_{HS}(n, p, G)$ has extremum at $G = \sqrt{p}$ for fixed n and p. The expression of $\frac{\partial T_{HS}}{\partial G}$ shows that, depending on the ratio of α and β the extremum can be either minimum or maximum in the interval $(1, p)$. If $G = \sqrt{p}$ is the minimum point it means that with $G = \sqrt{p}$ HSUMMA will outperform SUMMA, otherwise HSUMMA with $G = 1$ or $G = p$ will have the same performance as SUMMA.

Now let us apply this analysis to the HSUMMA communication cost function obtained for *scatter-allgather* broadcast algorithm (see Section 4.2.1.2) again assuming $b = M$ for simplicity. We will have:

$$\frac{\partial T_{HS}}{\partial G} = \frac{G - \sqrt{p}}{G\sqrt{G}} \times \left(\frac{n\alpha}{b} - 2\frac{n^2}{p} \times \beta \right) \tag{4.9}$$

It is clear that if $G = \sqrt{p}$ then $\frac{\partial T_{HS}}{\partial G} = 0$. Depending on the ratio of α and β, the communication cost as a function of G has either minimum or maximum in the interval $(1, p)$.

- If

$$\frac{\alpha}{\beta} > 2\frac{nb}{p} \tag{4.10}$$

then $\frac{\partial T_{HS}}{\partial G} < 0$ in the interval $(1, \sqrt{p})$ and $\frac{\partial T_{HS}}{\partial G} > 0$ in $(\sqrt{p}, p)$. Thus T_{HS}

has the minimum in the interval $(1, p)$ and the minimum point is $G = \sqrt{p}$.

- If

$$\frac{\alpha}{\beta} < 2\frac{nb}{p} \tag{4.11}$$

then T_{HS} has the maximum in the interval $(1, p)$ and the maximum point is $G = \sqrt{p}$. The function gets its minimum at either $G = 1$ or $G = p$.

If we take $G = \sqrt{p}$ in the HSUMMA communication cost function (see Section 4.2.1.2) and assume the above conditions, the optimal communication cost function will be as follows:

$$\left(\log_2(p) + 4\left(\sqrt[4]{p} - 1\right)\right) \times \frac{n}{b} \times \alpha + 8\left(1 - \frac{1}{\sqrt[4]{p}}\right) \times \frac{n^2}{\sqrt{p}} \times \beta \tag{4.12}$$

We will use the scatter-allgather model to predict the performance on future exascale platforms.

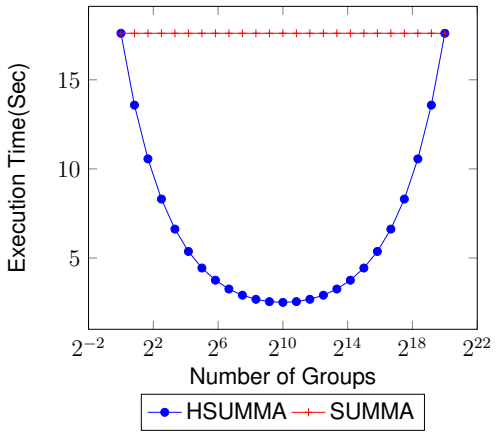
Now, let us take the communication cost function of HSUMMA with the *pipelined-linear* tree broadcast(see Section 4.2.1.2) and find the extremum of the function in $(1, p)$.

$$\frac{\partial T_{HS}}{\partial G} = \frac{G - \sqrt{p}}{G\sqrt{G}} \times \left(\frac{n}{b} \times \alpha + \frac{n^2}{\sqrt{p}X} \times \beta\right) \tag{4.13}$$

In the same way it can be proved that with the pipelined linear tree broadcast, independent of α and β, $G = \sqrt{p}$ is the minimum point of the communication function in $(1, p)$. A theoretical analysis of HSUMMA with the binomial tree broadcast can be found in [14].

4.2.1.4 Prediction on Exascale

We use parameters obtained from a recent report on exascale architecture roadmap[138] to predict performance of HSUMMA on exascale platforms.



- Total flop rate (γ): $1E18$ flops
- Latency: 500 ns,
- Bandwidth: 100 GB/s
- Problem size: $n = 2^{22}$,
- Number of processors: $p = 2^{20}$
- Block size: $b = M = 256$

Figure 4.5: Prediction of SUMMA and HSUMMA on Exascale. p=1048576.

Figure 4.5 shows that, theoretically, HSUMMA with any number of groups, outperforms SUMMA. It is worth mentioning that if the number of groups is equal to 1 or p, then HSUMMA will be equivalent to SUMMA, as in that case there is no hierarchy. Thus, theoretically, the communication cost function of HSUMMA has a parabola-like shape. In the next sections we will see that experimental results validate this theoretical prediction.

4.2.2 Experiments on BlueGene/P

Some of our experiments were carried out on the Shaheen BlueGene/P at the Supercomputing Laboratory at King Abdullah University of Science&Technology (KAUST) in Thuwal, Saudi Arabia. Shaheen is a 16-rack BlueGene/P. Each node is equipped with four 32-bit, 850 Mhz PowerPC 450 cores and 4GB DDR memory. VN (Virtual Node) mode with torus connection was used for the experiments. The Blue Gene/P architecture provides a three-dimensional point-to-point Blue Gene/P torus network which interconnects all compute nodes and global networks for collective and interrupt operations. Use of this network is integrated into the BlueGene/P MPI implementation.

All the sequential computations in our experiments were performed by using the DGEMM routine from the IBM ESSL library. We have implemented

SUMMA with block-checkerboard and block-cyclic distributions for comparison with HSUMMA. However, the data distribution in SUMMA does not change its performance on the BG/P. It may improve its performance if a modified communication pattern is used, as proposed in the DIMMA [125] algorithm. DIMMA was implemented in ScaLAPACK as a slight optimization of SUMMA, therefore, we also use ScaLAPACK(version 1.8.0) for the comparison with HSUMMA.

The benefit of HSUMMA comes from the optimal number of groups. Therefore, it is interesting to see how different numbers of groups affect the communication cost of HSUMMA on a large platform. Figure 4.6 shows HSUMMA on 16384 cores. In order to have a fair comparison again we use the same block size inside a group and between the groups. The figure shows that the execution time of SUMMA is 50.2 seconds. On the other hand, the minimum execution time of HSUMMA is 21.26 when G=512. Thus, the execution time of HSUMMA is 2.36 times less than that of SUMMA on 16384 cores. It is worth noting that different number of groups in HSUMMA does not affect the computation time, so all these reductions in the execution time come solely from the reduction of the communication time. In addition, according to our experiments, the improvement is 1.2 times on 2048 cores and the performance of HSUMMA and SUMMA are almost the same on BlueGene/P cores smaller than 2048. The zigzags in the figure can be explained by the fact that mapping communication layouts to network hardware on BlueGene/P impacts the communication performance, which was observed by P. Balaji et al. [139] as well. When we group processors we do not take into account the network topology. However, according to our preliminary observations these zigzags can be eliminated by taking the topology into account while grouping. Figure 4.6 represents scalability comparison of HSUMMA with SUMMA from communication point of view. Here, we use SUMMA both with block-checkerboard and block-cyclic distributions. It can be seen that HSUMMA is more scalable than SUMMA, and this pattern suggests that the communication performance of HSUMMA rapidly improves compared to that of SUMMA as the number of cores increases.

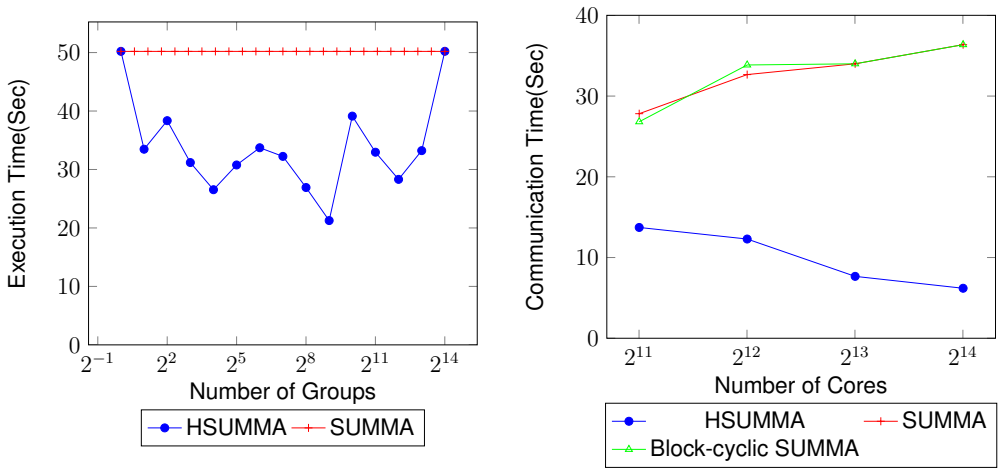


Figure 4.6: On the left execution times of SUMMA and HSUMMA on BG/P are given. On the right communication times of SUMMA, block-cyclic SUMMA and HSUMMA are given. $b = M = 256$, $n = 65536$.

According to the theoretical predictions, with some application/platform settings HSUMMA may not reduce the communication cost of SUMMA. We experimentally observed these phenomena on a smaller number of cores on the BG/P. Figure 4.7 illustrates one such experiment on 1024 cores, where the best performance of HSUMMA was achieved with G=1 and G=p. In this experiment, the interconnect type used between base partitions of the BG/P was a mesh as the minimum number of cores to use a torus interconnect is 2048.

4.2.2.1 Effect of Different Block Sizes

Theoretically the increase of the block size inside the groups should decrease the communication cost of HSUMMA. This section validates that by experimental results.

Figure 4.8 shows experimental results of HSUMMA with different block sizes inside the groups while the block size between the groups is fixed to 256 and the number of groups is fixed to 4. It can be seen that the communication time slightly decreases as the block size increases. Another interesting result is that, the relative performance of HSUMMA for different numbers of groups does not depend on the block size inside the groups. In

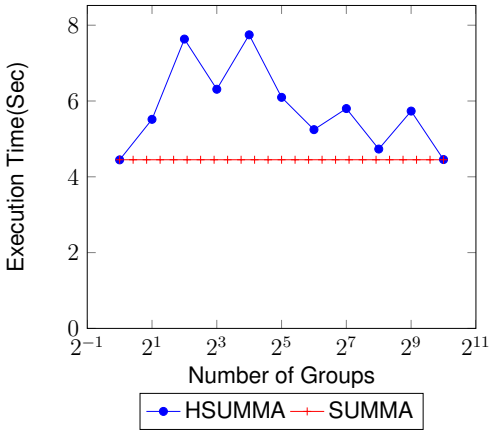


Figure 4.7: Execution time of HSUMMA and SUMMA on 1024 cores on BG/P. $b = M = 256$, $n = 16384$.

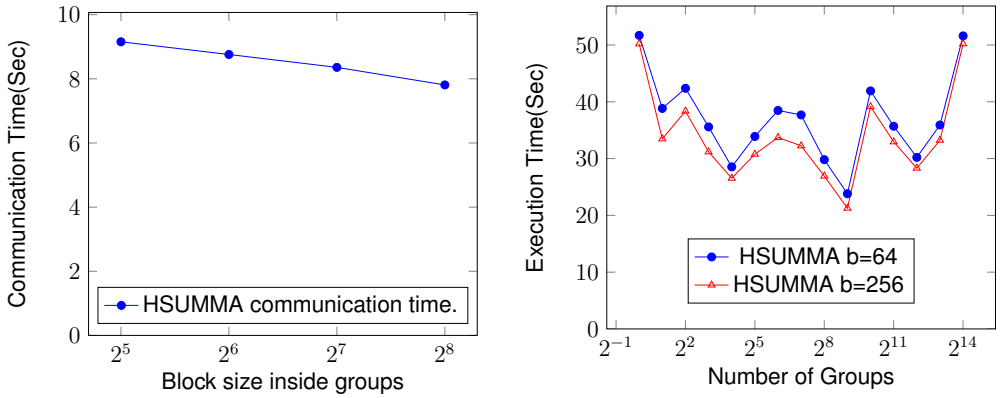


Figure 4.8: HSUMMA on 16384 cores on BG/P. $M = 256$ and $n = 65536$. On the left communication time is shown for a fixed block size of 256 between groups while changing the block size inside groups. On the right, the same setting is used to compare the performance with a block size of 64 and 256 inside groups.

particular, this means that the optimal value of G does not depend on the block size inside the groups, and therefore, any block size can be used in the procedure searching for the optimal value of G.

4.2.2.2 Comparison with ScaLAPACK

This section compares HSUMMA with the PDGEMM routine from the ScaLAPACK (ver. 1.8.0) library. The results of the corresponding experiments

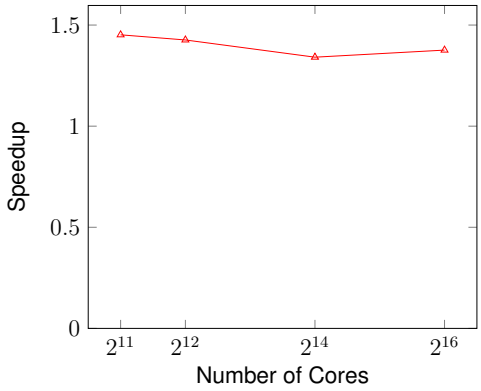


Figure 4.9: Speedup of HSUMMA over ScaLAPACK on BG/P. $b = M = 256$ *and* $n = 65536$.

are shown in Figure 4.9. Unfortunately, IBM PESSL is not available on the BG/P and therefore we cannot provide experimental results with PDGEMM from the PESSL library. However, it is known [140] that, unlike LU decomposition, PDGEMM from PESSL does not have any improvement over PDGEMM from ScaLAPACK. Moreover, the ScaLAPACK library on the BG/P uses a DGEMM from the IBM ESSL library which is optimized for Blue Gene.

4.2.3 Experiments on Grid'5000

Some of our experiments were carried out on the Grid'5000 infrastructure in France. Our experiments were performed on the Nancy site which is composed of three clusters: Graphene, Griffon and Graphite. We used the Graphene cluster for the experiments. The cluster is equipped with 144 nodes and each node has a disk of 320 GB storage, 16GB of memory and 4-cores of CPU Intel Xeon X3440. The nodes in the Graphene cluster have one 20GB Infiniband and are interconnected via Gigabyte Ethernet. We used multi-threaded dgemm from the GotoBlas2 library [117] for the sequential operations, MPICH 3.0.1 and OpenMPI 1.4.5 for MPI implementation, and our implementations of the matrix multiplication algorithms. The size of the matrices in our experiments on Grid'5000 was 8192×8192. The experiments with OpenMPI have been done with both Ethernet and Infiniband networks.

Here, we are not trying to compare different MPI implementations. Instead,

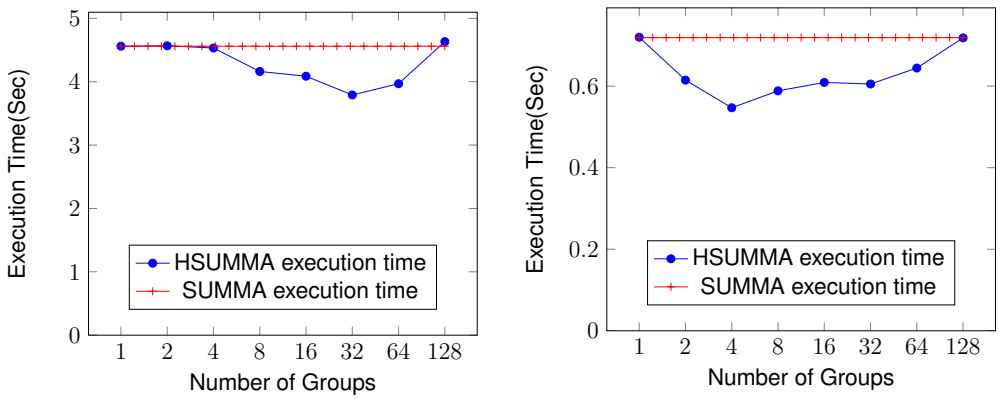


Figure 4.10: Experiments with OpenMPI on Grid'5000 with Ethernet (left) and Infiniband (right) networks. $b = M = 256$, $n = 8192$ *and* $p = 128$.

we show that the benefit of HSUMMA over SUMMA does not depend on the MPI implementation. Figure 4.10 shows that HSUMMA reduces the execution time of SUMMA by 16.8 percent on 128 nodes with an Ethernet network. The improvement with an Infiniband network is 24 percent.

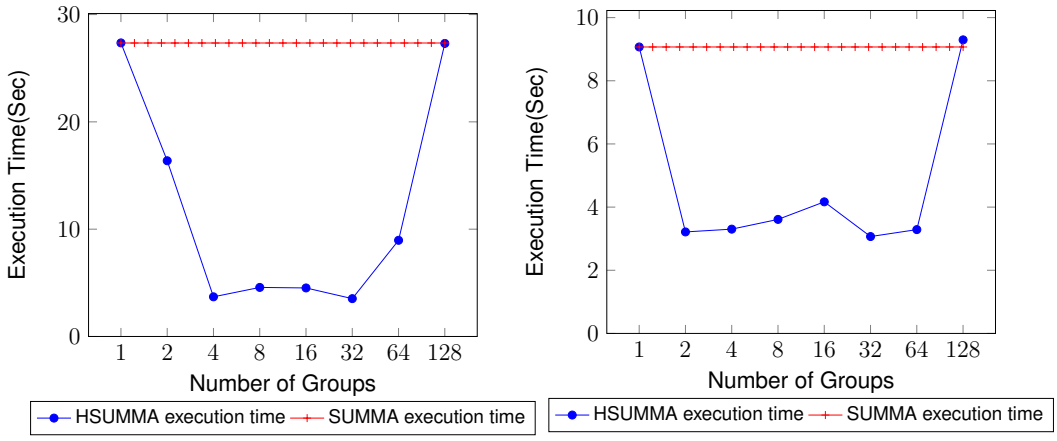


Figure 4.11: Experiments with MPICH on Grid'5000 with Ethernet network. $n = 8192$, $p = 128$, $b = 64$ *(left) and* $b = 256$ *(right)*

On the other hand, the improvement with MPICH is 7.75 times with a block size of 64 (see Figure 4.11) and 2.96 times with a block size of 256. This big difference comes from the MPI broadcast algorithm selection in MPICH depending on the message size and the number of processes. We did not fix the broadcast algorithm and allowed MPICH to decide which one to

use. In these experiments, the default values of MPICH parameters (e.g. BCAST_SHORT_MSG_SIZE, BCAST_MIN_PROCS) were used.

4.3 Conclusion

We can conclude that our two-level hierarchical approach to parallel matrix multiplication significantly reduces the communication cost on large platforms such as BlueGene/P. The experiments show that HSUMMA achieves 2.08 times and 5.89 times less communication time than SUMMA on 2048 cores and on 16384 cores respectively. Moreover, the overall execution time of HSUMMA is 1.2 times less than the overall execution time of SUMMA on 2048 cores, and 2.36 times less on 16384 cores. This trend suggests that, while the number of processors increases, HSUMMA will be more scalable than SUMMA. In addition, our experiments on Grid'5000 show that HSUMMA can be effective on small platforms as well.

Chapter 5

Hierarchical MPI Software Design

The hierarchical design of the MPI collective operations introduce the number of groups as a parameter. Despite we have studied analytical methods to estimate the optimal number of groups in the hierarchical algorithms, it is not always trivial to come up with a general model which can be accurate on all kinds of platforms. Therefore we have designed a software called Hierarchical MPI (HiMPI) that can automatically select the optimal number of groups during run time from multiple iterations of the given collective operation.

The current version of HiMPI supports MPI broadcast, reduce, allreduce, scatter, and gather operations. At the same time, we are actively developing the software to improve it and planning to incorporate the other MPI collective operations into the library.

HiMPI employs and extends MPIBlib [34] as the underlying benchmarking library.

5.1 MPIBlib

MPIBlib is an MPI benchmarking library that provides several methods of measurement both for point-to-point and collective communication operations. Unlike many other MPI benchmarking tools, MPIBlib supports both operation-specific and operation-independent measurements. The operation-specific timing methods can be particularly efficient in self-adaptive

applications.

The communication experiments in each benchmark can be performed in two ways using either fixed or variable number of repetitions. The user can control the accuracy of the estimations in the case of variable number of repetitions by providing the following inputs:

- The minimum (*min_reps*) and maximum (*max_reps*) number of iterations.
- The maximum error, ϵ, $(0 < \epsilon < 1)$.

The output of the measurements will be the actual number of repetitions and the error. If *min_reps* equals to *max_reps* then the benchmark will have a fixed number of repetitions and the estimation can be done within a confidence interval $(1-\epsilon)$. When *min_reps*$\leq$*max_reps*, the experiments will continue until the sample satisfies the Student's t-test or the number of repetitions reaches its maximum.

Three timing methods, global, maximum and root timings, are provided in MPIBlib. The GNU Scientific Library (GSL) [141] is used for statistical analysis. The library consists of mainly three libraries:

- mpiblib - benchmarking library
- mpiblib_p2p - point-to-point communication algorithms
- mpiblib_coll - collective communication algorithms

HiMPI excludes the communication libraries and only uses the benchmarking part of MPIBlib. Another main feature of MPIBlib is that it is open for extensions in such a way that the user can add new communication operations into the set of operations that can be benchmarked by the library. We especially benefit from this feature of MPIBlib and extend it to add our hierarchical collective algorithms for MPI broadcast, reduce, allreduce, scatter, and gather into the set of operations that can be benchmarked.

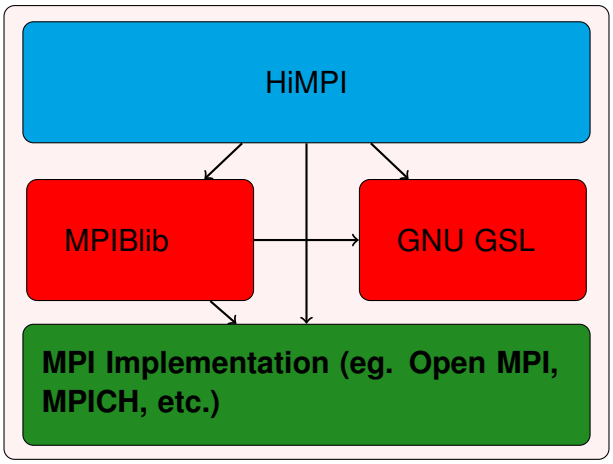


Figure 5.1: High-Level View of HiMPI Design

5.2 HiMPI - Hierarchical MPI

Finding the optimal number of groups in HiMPI is the vital part of its design. The trivial method would be running few iterations of benchmarking at the start of each hierarchical collective operation. However, in this case the overhead of the benchmarking can be significant. Therefore, we introduce HiMPI software tool to optimize automatic finding of the best number of groups during run time. HiMPI is designed as a GNU Autotools [142] project, implemented in the C programming language [143] and uses the MPIBlib library for benchmarking. A high level design of HiMPI is given in Figure 5.1. Finding the optimal number of groups incorporated mainly into the HiMPI initialization (HiMPI_Init function) where we do statistically rigorous benchmarking by means of the MPIBlib library for configurable range of message sizes and number of processes starting from some configurable initial value and up to the maximal number of MPI processes. The results of these measurements are saved in configuration files and the same measurements are not performed in future runs if the application is launched with a message size and number of processes that already exist in the configuration file. This behaviour is fully flexible and can be controlled by the user through configuration flags during installation time or environment

variables before each run. In addition, the user has a choice to disable the measurements in HiMPI_Init and do measurements in the first call to each HiMPI collective operation. In this case the benchmarking is performed only for the message size and the number of processes of the collective operation and the result again is saved in a configuration file. This method can be faster if the same setting is going to be used over again. Another configuration option provided to the users is that generation of configuration files can be completely disabled and be given manually. This method may not sound attractive to end users but it gives more flexibility to users who would like to do some research on top of HiMPI. The other duty of the HiMPI_Init is to define the required internal data structures which will be released by HiMPI_Finalize function call.

Currently, the only way to integrate HiMPI into an application is by inserting calls to HiMPI APIs, then compile and link with the library at build time. We are planning to add the interposition feature in near future, which will let the application developers to use HiMPI without changing a single line of code.

5.2.1 The HiMPI API

The HiMPI API is designed to be compatible with the MPI standard. It means that the APIs for HiMPI collective operations, HiMPI initialization and finalization have exactly the same method signatures. That being said, using HiMPI follows the same template as using MPI.

5.2.1.1 The HiMPI Initialization and Finalization

In order to use the HiMPI collective operations the application should initialize HiMPI. After finishing all the operations HiMPI should be finalized. The following code fragment shows a simple example how it can be done:

Listing 5.1: Finding the Optimal Number of Groups

```
/* Include the HiMPI header */
#include "himpi.h"
```

```
...

/* HiMPI initialisation */
HiMPI_Init(...);

...

/* HiMPI finalisation */
HiMPI_Finalize();
```

It is worth mentioning that an application should not call MPI_Init and MPI_Finalize if it uses HiMPI_Init and HiMPI_Finalize.

5.2.1.2 The HiMPI Collective Operations

- HiMPI Broadcast:

```
int HiMPI_Bcast(void *buffer, int count, MPI_Datatype
    datatype, int root, MPI_Comm comm)
```

- HiMPI Reduce:

```
int HiMPI_Reduce(void *snd_buffer, void* rcv_buffer, int
    count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm
    comm)
```

- HiMPI Allreduce

```
int HiMPI_Allreduce(void *sendbuf, void* recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

- HiMPI Gather

```
int HiMPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype
    sendtype, void *recvbuf, int recvcnt, MPI_Datatype
    recvtype, int root, MPI_Comm comm)
```

- HiMPI Scatter

```
int HiMPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype
    sendtype, void *recvbuf, int recvcnt, MPI_Datatype
    recvtype, int root, MPI_Comm comm)
```

The meanings of the parameters in these functions are exactly the same in their corresponding MPI collective operations. We can see that the number of groups is not provided as input and it is transparent to application developers. Internally HiMPI uses *get_himpi_group* function to find the optimal number of groups. The method has been implemented as in Listing 5.2.

Listing 5.2: Finding Optimal Number of Groups

```
1  int get_himpi_group(int msg_size, int root, MPI_Comm comm_world,
       int num_levels, int alg_in, int alg_out, himpi_operations op_id)
       {
2    MPIB_result result;
3    MPIB_precision precision;
4    MPIB_getopt_precision_default(&precision);
5    int num_procs;
6    MPI_Comm_size(comm_world, &num_procs);
7    int g, num_groups = 0;
8    for (g = 1; g < num_procs; g++) {
9       if (num_procs % g == 0)
10         num_groups++;
11   }
12   double* g_times = (double*) calloc(num_groups, sizeof(double));
13   if (g_times == NULL) {
14      fprintf(stderr, "[get_hbcast_group]:Can't allocate memory for
           g_times\n");
15      return -1;
16   }
17   int i = 0;
18   for (g = 1; g < num_procs; g++) {
19      if (num_procs % g == 0) {
```

```
20          MPIB_coll_container* container;
21          switch (op_id) {
22          case op_bcast:
23             container = (MPIB_coll_container*)
                   MPIB_HBcast_container_alloc(hierarchical_broadcast,
                   g, num_levels, alg_in, alg_out);
24             break;
25          case op_reduce:
26             container = (MPIB_coll_container*)
                   MPIB_HReduce_container_alloc(hierarchical_reduce, g,
                   num_levels, alg_in, alg_out);
27             break;
28          case op_allreduce:
29             container = (MPIB_coll_container*)
                   MPIB_HAllreduce_container_alloc(hierarchical_allreduce,
                   g, num_levels, alg_in, alg_out);
30             break;
31          case op_scatter:
32             container = (MPIB_coll_container*)
                   MPIB_HScatter_container_alloc(hierarchical_scatter,
                   g, num_levels, alg_in, alg_out);
33             break;
34          case op_gather:
35             container = (MPIB_coll_container*)
                   MPIB_HGather_container_alloc(hierarchical_gather, g,
                   num_levels, alg_in, alg_out);
36             break;
37          default:
38             fprintf(stdout, "Unknown operation id: %d\n", op_id);
39             MPI_Abort(MPI_COMM_WORLD, 201);
40             break;
41          }
42          int err = MPIB_measure_max(container, comm_world, root,
               msg_size, precision, &result);
43          g_times[i++] = result.T;
```

```
44        }
45    }
46    int min_idx = gsl_stats_min_index(g_times, 1, num_groups);
47    int group = get_specific_factor(num_procs, min_idx + 1);
48    free(g_times);
49    return group;
50 }
```

The variables needed by MPIBlib are defined between Line 2 and Line 4. After the benchmarking finishes, MPIB_result will consist of the message size, execution time, resolution of MPI_Wtime, number of repetitions the benchmark has actually taken and confidence interval. MPIB_precision contains minimum and maximum number of repetitions, confidence level and relative error that the benchmarking should take into account. We use the default values for the precision, namely, minimum and maximum repetitions are set to 5 and 100 respectively, and confidence level and relative error are set to 0.95 and 0.025 accordingly. Lines 45- 46 create appropriate MPIB container depending on the op_id and uses MPIB_measure_max routine to benchmark the collective operation for each number of groups from one up to the total number of processes. The current implementation supports only number of groups which are factors of the number of processes. It is planned that HiMPI will support any number of groups in future. After finishing benchmarking for all different number of groups, the group which results in the minimum execution time for the given collective operation is found on Line 46 and Line 47.

An example configuration file is given in Table B.1. The first column is the number of processes in an HiMPI collective operation, the second one is the optimal number of groups for this settings of row. Currently, HiMPI supports only one level of hierarchy. The third column shows the number of hierarchies. In each hierarchy of the collective operations it is possible to use different algorithms. This feature is not completely implemented at the moment and is planned to be done in near future. Alg_in shows the collective algorithm id inside groups, Alg_out shows that of between the groups. Finally, the last column shows HiMPI collective operation id, it is defined as follows in himpi.h

file and can be any number between 0 and 5:

```
typedef enum himpi_operations {
   op_bcast, op_reduce, op_allreduce, op_scatter, op_gather, op_all
} himpi_operations;
```

The operation id can be set during installation time via –with-himpi-opid option or can be set before each run via HIMPI_OPID environment variable. The environment variables always overwrite any corresponding variable defined during installation time. Environment variables are read in HiMPI_Init. If operation id is op_all then the measurements will be performed for all the collective operations and appropriate configuration file will be generated per operation id.

Table 5.1: Example HiMPI Configuration File

Num_procs	Num_groups	Num_levels	Msg_size	Alg_in	Alg_out	Operation_id
128	16	1	1024	0	0	0
128	16	1	2048	0	0	0
128	8	1	4048	0	0	0
64	8	1	1024	0	0	0
64	8	1	2048	0	0	0
64	4	1	4048	0	0	0
...	...	...	...	...	...	...

The HiMPI build and run time configuration parameters are explained in appendix B.

5.2.2 Experiments with HiMPI

This section presents an experimental study that investigates HiMPI performance and tradeoffs between the performance and overhead of the automatic selection of the number of groups.

The main idea of HiMPI is centered around finding the optimal number of groups which can be performed either during HiMPI_Init or inside the

hierarchical communication operations itself. The first option can cover a wide range of different configurations for different number of processes and/or message sizes. While the second option is only for the number of processes and the message size that the communication operation supposed to be executed on. Only one of this options can be employed for the given collective operation. As the first option is more general and it is the superset of the second option, the experimental study will cover only the first option, i.e. studying the overhead of HiMPI_Init.

The experiments have been done for the number of processes in the range of 16 and 128 on the Graphene cluster of the Grid'5000 platform. For each number of processes, different message sizes from 16B up to 16MB were used. In all the experiments we have configured the HIMPI_OPID be equal to zero, which means the broadcast operation. The results are based on the flat, chain, pipeline broadcast algorithms and the native Open MPI broadcast operation. In addition, we have conducted experiments on the BlueGene/P platform for number of processes within the range of 256 and 2048 using the linear, scatter-ring-allgather (SRG) and the native BG/P broadcast operation.

Table 5.2 summarizes the time taken by HiMPI_Init for one or several numbers of processes (Num. Processes) and messages starting from 16B (Min Msg. Size) and increasing by twice (Msg. Stride) until 16MB (Max Msg. Size). The fifth column shows the algorithm used for the broadcast operation. The results demonstrate that the execution time of the HiMPI_Init can vary drastically depending on the underlying broadcast algorithm. The hierarchical flat tree algorithm takes the longest benchmarking time in order to find the optimal number of groups, while the hierarchical native broadcast operation takes the shortest time.

Table 5.2: Execution Time of HiMPI_Init on Graphene Cluster of Grid'5000

Num. Processes	Min Msg. Size	Max Msg. Size	Msg. Stride	Algorithm	Time (sec)
128	16B	16MB	2	Native	132.96
128	16B	16MB	2	Flat	1508.63
128	16B	16MB	2	Chain	356.65
128	16B	16MB	2	Pipeline	1309.11
64	16B	16MB	2	Native	34.28
64	16B	16MB	2	Flat	624.18
64	16B	16MB	2	Chain	170.92
64	16B	16MB	2	Pipeline	554.03
32	16B	16MB	2	Native	30.99
32	16B	16MB	2	Flat	224.47
32	16B	16MB	2	Chain	80.91
32	16B	16MB	2	Pipeline	228.90
16; 32; 64; 128	16B	1MB	2	Native	53.27
16; 32; 64; 128	16B	1MB	2	Flat	127.01
16; 32; 64; 128	16B	1MB	2	Chain	44.26
16; 32; 64; 128	16B	1MB	2	Pipeline	141.88
16; 32; 64; 128	16B	64kB	2	Native	20.11
16; 32; 64; 128	16B	64kB	2	Flat	18.12
16; 32; 64; 128	16B	64kB	2	Chain	10.78
16; 32; 64; 128	16B	64kB	2	Pipeline	27.95

As we can see the time spent in the initialization does not take too long for small messages. For instance, the HiMPI_Init time for messages in the range of 16B and 1MB on 16, 32, 64, and 128 processes on the Graphene cluster using the pipeline broadcast algorithm is only 141.88 seconds altogether. On the other hand, initialization using messages ranging from 16B and 16MB just on 128 processes with the same algorithm is 1309.11 seconds (or 21.8 mins). The trends are similar on the BG/P (see Table 5.3). The initialization time can vary depending on the underlying collective algorithm, the message size and the number of processes. Despite in some cases the initialization of the HiMPI can take up to 20 minutes (with the flat tree broadcast) on Grid'5000, and up to

90 minutes (with the linear tree broadcast) on BG/P, this process needs to be done only once for a specific setting on a given cluster and thus the overhead can completely be avoided in any subsequent execution. The initialization time can be very fast in case of the native broadcast operation (which is used if the user does not request any non-default brodcast algorithm). For example, the initialization for messages ranging from 1kB up to 16MB and the number of processes from 16 up to 2048 (only a power-of-two numbers) takes about 3.5 minutes on the BG/P. This time can be reduced further if the initialization process is performed for a lesser number of processes or smaller message sizes.

Table 5.3: Execution Time of HiMPI_Init on BG/P

Num. Processes	Min Msg. Size	Max Msg. Size	Msg. Stride	Algorithm	Time (sec)
16; 32; 64; 128; 256; 512; 1024; 2048	1kB	16MB	2	Native	214.7679
16; 32; 64; 128; 256; 512; 1024; 2048	1kB	16MB	2	Linear	5560.959
16; 32; 64; 128; 256; 512; 1024; 2048	1kB	16MB	2	SRG	226.066
16; 32; 64; 128; 256; 512; 1024	1kB	16MB	2	Native	104.521
16; 32; 64; 128; 256; 512; 1024	1kB	16MB	2	Linear	2758.090
16; 32; 64; 128; 256; 512; 1024	1kB	16MB	2	SRG	126.981
16; 32; 64; 128; 256; 512	1kB	16MB	2	Native	61.829
16; 32; 64; 128; 256; 512	1kB	16MB	2	Linear	1496.145
16; 32; 64; 128; 256; 512	1kB	16MB	2	SRG	82.974
16; 32; 64; 128; 256	1kB	16MB	2	Native	42.427
16; 32; 64; 128; 256	1kB	16MB	2	Linear	656.237
16; 32; 64; 128; 256	1kB	16MB	2	SRG	53.578

Chapter 6

Conclusion

The proposed hierarchical approach to optimize the communication cost of MPI collective operations is simple, general and effective in many cases. Being topology-oblivious makes it applicable to a variety of platforms. Moreover, the transformation uses existing algorithms underneath which means that any lower-level platform dependent optimization of the underlying algorithm can still be effective after the hierarchical transformation. If for any reason depending on the topology or some other platform restrictions the transformation breaks the underlying algorithm, the hierarchical algorithm can fall back into the original algorithm through parametrized number of groups and be equally fast as the underlying algorithm. The experimental study demonstrates that the hierarchical broadcast, reduce, allreduce, scatter, and gather algorithms provide significant reduction in the communication time. The multifold improvement of performance is achieved especially for algorithms which use fully or partly flat arrangement of processes. Some of these algorithms are widely used and implemented (e.g. pipeline and scatter-ring-algather broadcast) in the state-of-the-art MPI implementations. The gain can be up to 30 times in some cases. Our application study in the context of dense parallel matrix multiplication on distributed memory platforms shows that the optimization technique can be applied to any application that uses MPI broadcast, or other collective operations.

6.0.3 Future Work

Our study of the hierarchical transformation of MPI collective operations opens up several promising research questions in this direction. Here we discuss these research questions as part of our future research work. The hierarchical approach can be applied to the other MPI collective operations and scientific applications employing collective communication patterns in their design.

For the time being, HiMPI is a reference implementation and provides only basic functionality and does not support using different collective algorithms at the different levels of the hierarchy. The state-of-the-art MPI implementations, Open MPI and MPICH, do not provide APIs to use a specific collective algorithm. Despite they provide configuration parameters (for example, Open MPI MCA parameters) to force a specific collective algorithm depending on the message and communicator size, this option is not flexible enough to be incorporated into our general-purpose library. Open MPI supports a rule based configuration file to be specified as MCA parameter in order to select a desired collective algorithm. Since the hierarchical algorithms can have the number of groups as parameter and the value of this parameter can change between 1 and the number of processes, p, the generation of the rule based configuration files for each possible values of the number of groups and its combination with different message sizes would require playing with lots of different configuration files. MPI-3 comes with tools interface, MPI_T, which lets users access and modify MPI performance and control variables. Control variables are able to control the behaviour of the MPI implementation (e.g. Open MPI mca parameters). After this feature is fully supported in near future, the employment of different Open MPI collective algorithms in each hierarchy of the hierarchical collectives can be realized in better and more reliable way.

Non-blocking collective communication operations are another new feature of the MPI-3 standard that we believe can be used to optimize the hierarchical algorithms further by overlapping the communications at the different levels of the hierarchy.

Despite our study focuses on one-level hierarchical optimization, the

approach can be applied in a multilevel hierarchical way.

Bibliography

[1] *Top 500 supercomputer sites*. [Online]. Available: http://www.top500.org/ (cit. on pp. iii, 1).

[2] P. Husbands and J. C. Hoe, "Mpi-start: Delivering network performance to numerical applications," in *Supercomputing, 1998. SC98. IEEE/ACM Conference on*, IEEE, 1998, pp. 17–17 (cit. on p. 2).

[3] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "Magpie: Mpi's collective communication operations for clustered wide area systems," in *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '99, ACM, 1999, pp. 131–140, ISBN: 1-58113-100-3. DOI: 10.1145/301104.301116 (cit. on pp. 2, 21).

[4] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan, "Exploiting hierarchy in parallel computer networks to optimize collective operation performance," in *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, 2000, pp. 377–384. DOI: 10.1109/IPDPS.2000.846009 (cit. on p. 2).

[5] I. Foster and N. Karonis, "A grid-enabled mpi: Message passing in heterogeneous distributed computing systems," in *Supercomputing, 1998.SC98. IEEE/ACM Conference on*, 1998, pp. 46–46. DOI: 10.1109/SC.1998.10051 (cit. on p. 2).

[6] R. Graham, M. Venkata, J. Ladd, P. Shamis, I. Rabinovitz, V. Filipov, and G. Shainer, "Cheetah: A framework for scalable hierarchical collective operations," in *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, 2011, pp. 73–83. DOI: 10.1109/CCGrid.2011.42 (cit. on pp. 2, 23, 29).

[7] E. Solomonik, A. Bhatele, and J. Demmel, "Improving communication performance in dense linear algebra via topology aware collectives," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, ACM, 2011, 77:1–77:11. DOI: 10.1145/2063384.2063487 (cit. on p. 3).

[8] T. Malik, V. Rychkov, A. Lastovetsky, and J.-N. Quintin, "Topology-aware optimization of communications for parallel matrix multiplication on hierarchical heterogeneous hpc platform," in *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, 2014, pp. 39–47. DOI: 10.1109/IPDPSW.2014.10 (cit. on p. 3).

[9] Lastovetsky A. and Dongarra J., *High Performance Heterogeneous Computing*. Wiley, 2009, p. 267 (cit. on pp. 3, 14, 91, 95).

[10] Hockney R. W., "The communication challenge for mpp: intel paragon and meiko cs-2," *Parallel Computing*, vol. 20, no. 3, pp. 389–398, 1994 (cit. on pp. 4, 11, 20).

[11] K. Hasanov, J.-N. Quintin, and A. Lastovetsky, "High-level topology-oblivious optimization of mpi broadcast algorithms on extreme-scale platforms," in *Euro-Par 2014: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science, vol. 8806, Springer International Publishing, 2014, pp. 412–424, ISBN: 978-3-319-14312-5. DOI: 10.1007/978-3-319-14313-2_35 (cit. on p. 5).

[12] K. Hasanov, J.-N. Quintin, and A. Lastovetsky, "Topology-oblivious optimization of mpi broadcast algorithms on extreme-scale platforms," *Simulation Modelling Practice and Theory*, vol. 58, pp. 30 –39, 2015,

ISSN: 1569-190X. DOI: 10.1016/j.simpat.2015.03.005 (cit. on p. 5).

[13] K. Hasanov and A. Lastovetsky, "Hierarchical optimization of mpi reduce algorithms," in *Parallel Computing Technologies*, ser. Lecture Notes in Computer Science, vol. 9251, Springer International Publishing, 2015, pp. 21–34, ISBN: 978-3-319-21908-0. DOI: 10.1007/978-3-319-21909-7_3 (cit. on p. 5).

[14] J.-N. Quintin, K. Hasanov, and A. Lastovetsky, "Hierarchical parallel matrix multiplication on large-scale distributed memory platforms," in *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ser. ICPP '13, IEEE Computer Society, 2013, pp. 754–762, ISBN: 978-0-7695-5117-3. DOI: 10.1109/ICPP.2013.89 (cit. on pp. 5, 106).

[15] K. Hasanov, J.-N. Quintin, and A. Lastovetsky, "Hierarchical approach to optimization of parallel matrix multiplication on large-scale platforms," *The Journal of Supercomputing*, vol. 71, no. 11, pp. 3991–4014, 2014, ISSN: 0920-8542. DOI: 10.1007/s11227-014-1133-x (cit. on p. 5).

[16] (). Message passing interface forum, [Online]. Available: http://www.mpi-forum.org/ (cit. on pp. 6, 28, 36, 57).

[17] V. S. Sunderam, "Pvm: a framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2, no. 4, pp. 315–339, Nov. 1990, ISSN: 1040-3108. DOI: 10.1002/cpe.4330020404 (cit. on p. 7).

[18] D. Gelernter, "Generative communication in linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 80–112, Jan. 1985, ISSN: 0164-0925. DOI: 10.1145/2363.2433 (cit. on p. 7).

[19] N. J. Carriero, D. Gelernter, T. G. Mattson, and A. H. Sherman, "The linda alternative to message-passing systems," *Parallel Computing*, vol. 20, no. 4, pp. 633–655, Apr. 1994, ISSN: 0167-8191. DOI: 10.1016/0167-8191(94)90032-9 (cit. on p. 7).

[20] D. Arapov, A. Kalinov, A. Lastovetsky, I. Ledovskih, and T. Lewis, “A programming environment for heterogeneous distributed memory machines,” in *Proceedings of the 6th Heterogeneous Computing Workshop (HCW’97)*, IEEE, 1997, pp. 32–45 (cit. on p. 8).

[21] A. Lastovetsky, “Adaptive parallel computing on heterogeneous networks with mpc,” *Parallel Computing*, vol. 28, no. 10, pp. 1369–1407, 2002, ISSN: 0167-8191. DOI: 10.1016/S0167-8191(02)00159-X (cit. on p. 8).

[22] A. Lastovetsky and R. Reddy, “Hmpi: Towards a message-passing library for heterogeneous networks of computers,” in *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, IEEE Computer Society, 2003 (cit. on p. 8).

[23] A. Lastovetsky and R. Reddy, “Heterompi: Towards a message-passing library for heterogeneous networks of computers,” *Journal of Parallel and Distributed Computing*, vol. 66, no. 2, pp. 197–220, 2006 (cit. on p. 8).

[24] L. V. Kale and S. Krishnan, “Charm++: a portable concurrent object oriented system based on c++,” in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA ’93, Washington, D.C., USA, 1993, pp. 91–108, ISBN: 0-89791-587-9. DOI: 10.1145/165854.165874 (cit. on p. 9).

[25] B. Stroustrup, *The C++ programming language*. Pearson Education, 2013 (cit. on p. 9).

[26] C. Huang, O. Lawlor, and L. KalÃl’, “Adaptive mpi,” English, in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, vol. 2958, Springer Berlin Heidelberg, 2004, pp. 306–322, ISBN: 978-3-540-21199-0. DOI: 10.1007/978-3-540-24644-2_20 (cit. on p. 9).

[27] (). Pgas forum, [Online]. Available: http://www.pgas.org/ (cit. on p. 9).

[28] H. Richardson, “High performance fortran: History, overview and current developments,” 1.4 TMC-261, Thinking Machines Corporation, Tech. Rep., 1996 (cit. on p. 10).

[29] R. W. Numrich and J. Reid, “Co-array fortran for parallel programming,” *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, Aug. 1998, ISSN: 1061-7264. DOI: 10.1145/289918.289920 (cit. on p. 10).

[30] ——, “Co-arrays in the next fortran standard,” *SIGPLAN Fortran Forum*, vol. 24, no. 2, pp. 4–17, Aug. 2005, ISSN: 1061-7264. DOI: 10.1145/1080399.1080400 (cit. on p. 10).

[31] D. C. K. Y. E. B. W. Carlson J. Draper and K. Warren, “Introduction to upc and language specification,” IDA Center for Computing Sciences, Tech. Rep. CCS-TR-99-157, 1999 (cit. on p. 10).

[32] D. Callahan, B. Chamberlain, and H. Zima, “The cascade high productivity language,” in *High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on*, 2004, pp. 52–60. DOI: 10.1109/HIPS.2004.1299190 (cit. on p. 10).

[33] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: an object-oriented approach to non-uniform cluster computing,” *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005, ISSN: 0362-1340. DOI: 10.1145/1103845.1094852 (cit. on p. 10).

[34] A. Lastovetsky, V. Rychkov, and M. O’Flynn, “Mpiblib: Benchmarking mpi communications for parallel computing on homogeneous and heterogeneous clusters,” in *Proceedings of the 15th European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer-Verlag, 2008, pp. 227–238, ISBN: 978-3-540-87474-4. DOI: 10.1007/978-3-540-87475-1_32 (cit. on pp. 11, 52, 62, 114).

[35] Q. O. Snell, A. R. Mikler, and J. L. Gustafson, “Netpipe: A network protocol independent performance evaluator,” in *In IASTED International Conference on Intelligent Information Management and Systems*, 1996 (cit. on p. 11).

[36] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, “Logp: Towards a realistic model of parallel computation,” in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’93, ACM, 1993, pp. 1–12, ISBN: 0-89791-589-5. DOI: 10.1145/155332.155333 (cit. on pp. 11, 12).

[37] D. Culler, A. C. Dusseau, R. P. Martin, and K. E. Schauser, “Fast parallel sorting under logp: From theory to practice,” *Portability and Performance for Parallel Processing*, 1994 (cit. on p. 12).

[38] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. 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*, ser. SPAA ’95, ACM, 1995, pp. 95–105, ISBN: 0-89791-717-0. DOI: 10.1145/215399.215427 (cit. on p. 12).

[39] T. Kielmann, H. Bal, and K. Verstoep, “Fast measurement of logp parameters for message passing platforms,” in *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, ser. Lecture Notes in Computer Science, vol. 1800, Springer Berlin Heidelberg, 2000, pp. 1176–1183, ISBN: 978-3-540-67442-9. DOI: 10.1007/3-540-45591-4_162 (cit. on p. 12).

[40] F. Ino, N. Fujimoto, and K. Hagihara, “Loggps: A parallel computational model for synchronization analysis,” in *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, ser. PPoPP ’01, Snowbird, Utah, USA: ACM, 2001, pp. 133–142, ISBN: 1-58113-346-4. DOI: 10.1145/379539.379592 (cit. on p. 12).

[41] T. Hoefler, T. Schneider, and A. Lumsdaine, "Loggopsim: Simulating large-scale applications in the loggops model," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10, Chicago, Illinois: ACM, 2010, pp. 597–604, ISBN: 978-1-60558-942-8. DOI: 10.1145/1851476.1851564 (cit. on p. 12).

[42] F. Cappello, P. Fraigniaud, B. Mans, and A. Rosenberg, "Hihcohp-toward a realistic communication model for hierarchical hyperclusters of heterogeneous processors," in *Parallel and Distributed Processing Symposium., Proceedings 15th International*, 2001, 6 pp.–. DOI: 10.1109/IPDPS.2001.924978 (cit. on p. 13).

[43] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990, ISSN: 0001-0782. DOI: 10.1145/79173.79181 (cit. on p. 13).

[44] A. Bar-Noy and S. Kipnis, "Designing broadcasting algorithms in the postal model for message-passing systems," *Mathematical systems theory*, vol. 27, no. 5, pp. 431–452, 1994, ISSN: 0025-5661. DOI: 10.1007/BF01184933 (cit. on p. 13).

[45] M. Banikazemi, V. Moorthy, and D. Panda, "Efficient collective communication on heterogeneous networks of workstations," in *Parallel Processing, 1998. Proceedings. 1998 International Conference on*, 1998, pp. 460–467. DOI: 10.1109/ICPP.1998.708518 (cit. on p. 13).

[46] A. L. Rosenberg, "Optimal sharing of partitionable workloads in heterogeneous networks of workstations (extended abstract)," in *Intl. Wkshp. on Cluster Computing – Technologies, Environments, and Applications (CC-TEA'2000). In Intl. Conf. on Parallel and Distr. Processing Techniques and Applications (PDPTA'2000)*, 2000, pp. 413–419 (cit. on p. 13).

[47] C. Lin, "Efficient broadcast in a heterogeneous network of workstations using two sub-networks," in *Parallel Architectures,*

Algorithms and Networks, 2004. Proceedings. 7th International Symposium on, 2004, pp. 273–279. DOI: 10.1109/ISPAN.2004.1300492 (cit. on p. 13).

[48] P. Bhat, V. Prasanna, and C. Raghavendra, "Adaptive communication algorithms for distributed heterogeneous systems," in *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, 1998, pp. 310–321 (cit. on p. 13).

[49] P. Bhat, C. Raghavendra, and V. Prasanna, "Efficient collective communication in distributed heterogeneous systems," in *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*, 1999, pp. 15–24. DOI: 10.1109/ICDCS.1999.776502 (cit. on p. 13).

[50] A. Lastovetsky, I.-H. Mkwawa, and M. O'Flynn, "An accurate communication model of a heterogeneous cluster based on a switch-enabled ethernet network," in *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, vol. 2, 2006, 6 pp.–. DOI: 10.1109/ICPADS.2006.24 (cit. on p. 13).

[51] A. Lastovetsky and V. Rychkov, "Building the communication performance model of heterogeneous clusters based on a switched network," in *Cluster Computing, 2007 IEEE International Conference on*, 2007, pp. 568–575. DOI: 10.1109/CLUSTR.2007.4629284 (cit. on p. 13).

[52] A. L. Lastovetsky, V. Rychkov, and M. O'Flynn, "Revisiting communication performance models for computational clusters," in *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, 2009, pp. 1–11. DOI: 10.1109/IPDPS.2009.5160918 (cit. on p. 13).

[53] C. A. Moritz and M. I. Frank, "Logpc: Modeling network contention in message-passing programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 4, pp. 404–415, Apr. 2001, ISSN: 1045-9219. DOI: 10.1109/71.920589 (cit. on p. 13).

[54] A. Lastovetsky and M. O'Flynn, "A performance model of many-to-one collective communications for parallel computing," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370574 (cit. on p. 13).

[55] M. Martinasso and J.-F. Mï£¡haut, "A contention-aware performance model for hpc-based networks: A case study of the infiniband network," in *Euro-Par 2011 Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 6852, Springer Berlin Heidelberg, 2011, pp. 91–102, ISBN: 978-3-642-23399-9. DOI: 10.1007/978-3-642-23400-2_10 (cit. on p. 13).

[56] J. Zhu, A. Lastovetsky, S. Ali, R. Riesen, and K. Hasanov, "Asymmetric communication models for resource-constrained hierarchical ethernet networks," *Concurrency and Computation: Practice and Experience*, vol. 27, pp. 1575–1590, 2015. DOI: 10.1002/cpe.3343 (cit. on p. 14).

[57] J. Zhu, A. Lastovetsky, S. Ali, and R. Riesen, "Communication models for resource constrained hierarchical ethernet networks," in *Euro-Par 2013: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science, vol. 8374, Springer Berlin Heidelberg, 2014, pp. 259–269, ISBN: 978-3-642-54419-4. DOI: 10.1007/978-3-642-54420-0_26 (cit. on p. 14).

[58] P. Velho, L. M. Schnorr, H. Casanova, and A. Legrand, "On the validity of flow-level tcp network models for grid and cloud simulations," *ACM Transactions on Modeling and Computer Simulation*, vol. 23, no. 4, Dec. 2013, ISSN: 1049-3301. DOI: 10.1145/2517448 (cit. on p. 14).

[59] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005. DOI: 10.1177/1094342005051521 (cit. on pp. 14, 26, 29, 32, 47, 48).

[60] R. A. Van De Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol.

9, no. 4, pp. 255–274, 1997, ISSN: 1096-9128. DOI: 10.1002/(SICI)1096-9128(199704)9:4<255::AID-CPE250>3.0.CO;2-2 (cit. on pp. 14, 90, 93, 94).

[61] V. Bala, J. Bruck, R. Cypher, P. Elustondo, C.-T. Ho, C.-T. Ho, S. Kipnis, and M. Snir, "Ccl: A portable and tunable collective communication library for scalable parallel computers," in *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, 1994, pp. 835–844. DOI: 10.1109/IPPS.1994.288208 (cit. on pp. 14, 21, 28).

[62] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: Past, present and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003, ISSN: 1532-0634. DOI: 10.1002/cpe.728. [Online]. Available: http://dx.doi.org/10.1002/cpe.728 (cit. on p. 20).

[63] S. Johnsson and C.-T. Ho, "Optimum broadcasting and personalized communication in hypercubes," *Computers, IEEE Transactions on*, vol. 38, no. 9, pp. 1249–1268, 1989, ISSN: 0018-9340. DOI: 10.1109/12.29465 (cit. on p. 20).

[64] P. Sanders, J. Speck, and J. L. Träff, "Two-tree algorithms for full bandwidth broadcast, reduction and scan," *Parallel Comput.*, vol. 35, no. 12, pp. 581–594, Dec. 2009, ISSN: 0167-8191. DOI: 10.1016/j.parco.2009.09.001 (cit. on pp. 20, 29).

[65] M. Barnett, L. Shuler, R. Van De Geijn, S. Gupta, D. Payne, and J. Watts, "Interprocessor collective communication library (intercom)," in *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, 1994, pp. 357–364. DOI: 10.1109/SHPCC.1994.296665 (cit. on pp. 20, 22, 28).

[66] E. Gabriel, M. Resch, and R. Rühle, "Implementing MPI with optimized algorithms for metacomputing," in *Message Passing Interface Developer's and Users Conference (MPIDC'99)*, 1999, pp. 31–41 (cit. on pp. 21, 28).

[67] B. Lowekamp and A. Beguelin, “Eco: Efficient collective operations for communication on heterogeneous networks,” in *Parallel Processing Symposium, 1996., Proceedings of IPPS ’96, The 10th International*, 1996, pp. 399–405. DOI: 10.1109/IPPS.1996.508087 (cit. on p. 21).

[68] S. S. Vadhiyar, G. E. Fagg, and J. J. Dongarra, “Towards an accurate model for collective communications,” *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 159–167, 2004. DOI: 10.1177/1094342004041297 (cit. on p. 22).

[69] S. Vadhiyar, G. Fagg, and J. Dongarra, “Automatically tuned collective communications,” in *Supercomputing, ACM/IEEE 2000 Conference*, 2000, pp. 3–3. DOI: 10.1109/SC.2000.10024 (cit. on pp. 22, 29).

[70] P. Sanders and J. F. Sibeyn, “A bandwidth latency tradeoff for broadcast and reduction,” *Information Processing Letters*, vol. 86, no. 1, pp. 33 –38, 2003, ISSN: 0020-0190. DOI: http://dx.doi.org/10.1016/S0020-0190(02)00473-8 (cit. on p. 22).

[71] J. L. Trï£¡ff and A. Ripke, “Optimal broadcast for fully connected processor-node networks,” *Journal of Parallel and Distributed Computing*, vol. 68, no. 7, pp. 887 –901, 2008, ISSN: 0743-7315. DOI: http://dx.doi.org/10.1016/j.jpdc.2007.12.001 (cit. on p. 22).

[72] T. Chiba, T. Endo, and S. Matsuoka, “High-performance mpi broadcast algorithm for grid environments utilizing multi-lane nics,” in *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, 2007, pp. 487–494. DOI: 10.1109/CCGRID.2007.59 (cit. on p. 22).

[73] (). Mpich - a portable implementation of mpi, [Online]. Available: http://www.mpich.org/ (cit. on p. 22).

[74] E. Gabriel, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall, “Open mpi: Goals, concept, and design of a next generation mpi implementation,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*,

ser. Lecture Notes in Computer Science, vol. 3241, Springer Berlin Heidelberg, 2004, pp. 97–104, ISBN: 978-3-540-23163-9. DOI: 10.1007/978-3-540-30218-6_19 (cit. on p. 22).

[75] J. WATTS and R. VAN DE GEIJN, "A pipelined broadcast for multidimensional meshes," *Parallel Processing Letters*, vol. 05, no. 02, pp. 281–292, 1995. DOI: 10.1142/S0129626495000266 (cit. on p. 22).

[76] D. Wadsworth and Z. Chen, "Performance of mpi broadcast algorithms," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1–7. DOI: 10.1109/IPDPS.2008.4536478 (cit. on p. 22).

[77] M. Saldana and P. Chow, "Tmd-mpi: An mpi implementation for multiple processors across multiple fpgas," in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, 2006, pp. 1–6. DOI: 10.1109/FPL.2006.311233 (cit. on p. 22).

[78] Y. Peng, M. Saldana, and P. Chow, "Hardware support for broadcast and reduce in mpsoc," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, 2011, pp. 144–150. DOI: 10.1109/FPL.2011.34 (cit. on p. 23).

[79] J. Liu, A. Mamidala, and D. Panda, "Fast and scalable mpi-level broadcast using infiniband's hardware multicast support," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004, pp. 10–. DOI: 10.1109/IPDPS.2004.1302912 (cit. on p. 23).

[80] T. Hoefler, C. Siebert, and W. Rehm, "A practically constant-time mpi broadcast algorithm for large-scale infiniband clusters with multicast," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370475 (cit. on p. 23).

[81] (). Infiniband trade association, [Online]. Available: http://www.infinibandta.org/ (cit. on p. 23).

[82] A. Mamidala, L. Chai, H.-W. Jin, and D. Panda, "Efficient smp-aware mpi-level broadcast over infiniband's hardware multicast," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006, 8 pp.–. DOI: 10.1109/IPDPS.2006.1639562 (cit. on p. 23).

[83] C. Karlsson, T. Davies, C. Ding, H. Liu, and Z. Chen, "Optimizing process-to-core mappings for two dimensional broadcast/reduce on multicore architectures," in *Parallel Processing (ICPP), 2011 International Conference on*, 2011, pp. 404–413. DOI: 10.1109/ICPP.2011.26 (cit. on p. 23).

[84] T. Ma, T. Herault, G. Bosilca, and J. Dongarra, "Process distance-aware adaptive mpi collective communications," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, 2011, pp. 196–204. DOI: 10.1109/CLUSTER.2011.30 (cit. on p. 23).

[85] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "Hwloc: A generic framework for managing hardware affinities in hpc applications," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, 2010, pp. 180–186. DOI: 10.1109/PDP.2010.67 (cit. on p. 23).

[86] H. Zhu, D. Goodell, W. Gropp, and R. Thakur, "Hierarchical collectives in mpich2," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, vol. 5759, Springer Berlin Heidelberg, 2009, pp. 325–326, ISBN: 978-3-642-03769-6. DOI: 10.1007/978-3-642-03770-2_41 (cit. on p. 23).

[87] W. Gropp, "Mpich2: a new start for mpi implementations," English, in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, vol. 2474, Springer Berlin Heidelberg, 2002, pp. 7–7, ISBN: 978-3-540-44296-7. DOI: 10.1007/3-540-45825-5_5 (cit. on p. 23).

[88] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer, "The deep computing messaging framework: Generalized scalable message passing on the blue gene/p supercomputer," in *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ser. ICS '08, ACM, 2008, pp. 94–103, ISBN: 978-1-60558-158-3. DOI: 10.1145/1375527.1375544 (cit. on pp. 23, 48).

[89] K. Kandalla, A. Venkatesh, K. Hamidouche, S. Potluri, D. Bureddy, and D. Panda, "Designing optimized mpi broadcast and allreduce for many integrated core (mic) infiniband clusters," in *High-Performance Interconnects (HOTI), 2013 IEEE 21st Annual Symposium on*, 2013, pp. 63–70. DOI: 10.1109/HOTI.2013.26 (cit. on p. 23).

[90] H. Subramoni, K. Kandalla, J. Vienne, S. Sur, B. Barth, K. Tomko, R. Mclay, K. Schulz, and D. Panda, "Design and evaluation of network topology-/speed- aware broadcast algorithms for infiniband clusters," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, 2011, pp. 317–325. DOI: 10.1109/CLUSTER.2011.43 (cit. on p. 23).

[91] K. Dichev and A. Lastovetsky, "Optimization of collective communication for heterogeneous hpc platforms," in *High-Performance Computing on Complex Environments*. John Wiley and Sons, Inc., 2014, pp. 95–114, ISBN: 9781118711897. DOI: 10.1002/9781118711897.ch6 (cit. on p. 23).

[92] Y. Gong, B. He, and J. Zhong, "Network performance aware mpi collective communication operations in the cloud," *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2013, ISSN: 1045-9219. DOI: 10.1109/TPDS.2013.96 (cit. on p. 23).

[93] T. Gunarathne, J. Qiu, and D. Gannon, "Towards a collective layer in the big data stack," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, 2014, pp. 236–245. DOI: 10.1109/CCGrid.2014.123 (cit. on p. 23).

[94] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008, ISSN: 0001-0782. DOI: 10.1145/1327452.1327492 (cit. on p. 23).

[95] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, “Twister: a runtime for iterative mapreduce,” in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC ’10, Chicago, Illinois: ACM, 2010, pp. 810–818, ISBN: 978-1-60558-942-8. DOI: 10.1145/1851476.1851593 (cit. on p. 24).

[96] Pješivac-Grbović J., “Towards automatic and adaptive optimizations of mpi collective operations,” PhD thesis, University of Tennessee, Knoxville, December, 2007 (cit. on pp. 25, 35, 37, 39, 40).

[97] R. Rabenseifner, “Automatic profiling of mpi applications with hardware performance counters,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, vol. 1697, Springer Berlin Heidelberg, 1999, pp. 35–42, ISBN: 978-3-540-66549-6. DOI: 10.1007/3-540-48158-3_5 (cit. on pp. 28, 57).

[98] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, “Mpi’s reduction operations in clustered wide area systems,” in *In Proc. MPIDC’99, Message Passing Interface Developer’s and User’s Conference*, 1999, pp. 43–52 (cit. on p. 28).

[99] G. Iannello, “Efficient algorithms for the reduce-scatter operation in loggp,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 8, no. 9, pp. 970–982, 1997, ISSN: 1045-9219. DOI: 10.1109/71.615442 (cit. on p. 28).

[100] M. Bernaschi, G. Iannello, and M. Lauria, “Efficient implementation of reduce-scatter in mpi,” in *Parallel, Distributed and Network-based Processing, 2002. Proceedings. 10th Euromicro Workshop on*, 2002, pp. 301–308. DOI: 10.1109/EMPDP.2002.994296 (cit. on p. 29).

[101] P. Patarasuk and X. Yuan, “Bandwidth efficient all-reduce operation on tree topologies,” in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370405 (cit. on p. 29).

[102] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, “On optimizing collective communication,” in *Cluster Computing, 2004 IEEE International Conference on*, 2004, pp. 145–155. DOI: 10.1109/CLUSTR.2004.1392612 (cit. on pp. 29, 36, 39, 40).

[103] R. Rabenseifner, “Optimization of collective reduction operations,” in *Computational Science - ICCS 2004*, ser. Lecture Notes in Computer Science, vol. 3036, Springer Berlin Heidelberg, 2004, pp. 1–9, ISBN: 978-3-540-22114-2. DOI: 10.1007/978-3-540-24685-5_1 (cit. on pp. 29, 32).

[104] J. Hatta and S. Shibusawa, “Scheduling algorithms for efficient gather operations in distributed heterogeneous systems,” in *Parallel Processing, 2000. Proceedings. 2000 International Workshops on*, 2000, pp. 173–180. DOI: 10.1109/ICPPW.2000.869101 (cit. on p. 36).

[105] K. Kandalla, H. Subramoni, A. Vishnu, and D. Panda, “Designing topology-aware collective communication algorithms for large scale infiniband clusters: Case studies with scatter and gather,” in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, 2010, pp. 1–8. DOI: 10.1109/IPDPSW.2010.5470853 (cit. on p. 36).

[106] J. Traff, “Hierarchical gather/scatter algorithms with graceful degradation,” in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004, pp. 80–. DOI: 10.1109/IPDPS.2004.1303019 (cit. on p. 36).

[107] K. Dichev, V. Rychkov, and A. Lastovetsky, “Two algorithms of irregular scatter/gather operations for heterogeneous platforms,” in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, vol. 6305, Springer Berlin Heidelberg,

2010, pp. 289–293, ISBN: 978-3-642-15645-8. DOI: 10.1007/978-3-642-15646-5_31 (cit. on p. 37).

[108] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard, "Grid'5000: A large scale and highly reconfigurable grid experimental testbed," in *Grid Computing, 2005. The 6th IEEE/ACM International Workshop on*, 2005, 8 pp.–. DOI: 10.1109/GRID.2005.1542730 (cit. on p. 52).

[109] (). Grid5000, [Online]. Available: http://www.grid5000.fr (cit. on p. 57).

[110] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Matrix multiplication on heterogeneous platforms," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 12, no. 10, pp. 1033–1051, 2001, ISSN: 1045-9219. DOI: 10.1109/71.963416 (cit. on pp. 91, 95, 96).

[111] F. G. Gustavson, "Cache blocking for linear algebra algorithms," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wa?niewski, Eds., vol. 7203, Springer Berlin Heidelberg, 2012, pp. 122–132, ISBN: 978-3-642-31463-6. DOI: 10.1007/978-3-642-31464-3_13 (cit. on p. 91).

[112] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, ser. FOCS '99, IEEE Computer Society, 1999, pp. 285–297, ISBN: 0-7695-0409-4. DOI: 10.1109/SFFCS.1999.814600 (cit. on p. 91).

[113] Yotov K., Roeder T., Pingali K., Gunnels J., and Gustavson F., "An experimental comparison of cache-oblivious and cache-conscious programs," in *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '07, ACM, 2007, pp. 93–104. DOI: 10.1145/1248377.1248394 (cit. on p. 91).

[114] S. Chatterjee, A. Lebeck, P. Patnala, and M. Thottethodi, "Recursive array layouts and fast matrix multiplication," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 11, pp. 1105–1123, 2002, ISSN: 1045-9219. DOI: 10.1109/TPDS.2002.1058095 (cit. on p. 91).

[115] (). Basic linear algebra routines (blas), [Online]. Available: http://www.netlib.org/blas/ (cit. on p. 91).

[116] Clint W. R. and Dongarra J. J., "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '98, Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–27 (cit. on p. 92).

[117] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, 12:1–12:25, May 2008, ISSN: 0098-3500. DOI: 10.1145/1356052.1356053 (cit. on pp. 92, 111).

[118] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," AAI7010025, PhD thesis, Bozeman, MT, USA, 1969 (cit. on p. 92).

[119] G. Fox, S. Otto, and A. Hey, "Matrix algorithms on a hypercube i: Matrix multiplication," *Parallel Computing*, vol. 4, no. 1, pp. 17 –31, 1987, ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/0167-8191(87)90060-3 (cit. on p. 92).

[120] J. Choi, D. W. Walker, and J. J. Dongarra, "Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers," *Concurrency: Practice and Experience*, vol. 6, no. 7, pp. 543–570, 1994, ISSN: 1096-9128. DOI: 10.1002/cpe.4330060702 (cit. on p. 92).

[121] S. Huss-Lederman, E. M. Jacobson, A. Tsao, and G. Zhang, "Matrix multiplication on the intel touchstone delta," *Concurrency: Practice and Experience*, vol. 6, no. 7, pp. 571–594, 1994, ISSN: 1096-9128. DOI: 10.1002/cpe.4330060703 (cit. on p. 92).

[122] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, “A three-dimensional approach to parallel matrix multiplication,” *IBM Journal of Research and Development*, vol. 39, no. 5, pp. 575–582, Sep. 1995, ISSN: 0018-8646. DOI: 10.1147/rd.395.0575 (cit. on p. 92).

[123] Agarwal. R. C., Gustavson. F. G., and Zubair. M., “A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication,” *IBM Journal of Research and Development*, vol. 38, no. 6, pp. 673–681, Nov. 1994. DOI: 10.1147/rd.386.0673 (cit. on p. 93).

[124] L. Blackford, J. Choi, A. Cleary, E. D’Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley, *ScaLAPACK Users’ Guide*. Society for Industrial and Applied Mathematics, 1997. DOI: 10.1137/1.9780898719642 (cit. on p. 93).

[125] J. Choi, “A new parallel matrix multiplication algorithm on distributed-memory concurrent computers,” in *High Performance Computing on the Information Superhighway, 1997. HPC Asia ’97*, 1997, pp. 224–229. DOI: 10.1109/HPC.1997.592151 (cit. on pp. 93, 108).

[126] M. Krishnan and J. Nieplocha, “Srumma: A matrix multiplication algorithm suitable for clusters and scalable shared memory systems,” in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004, pp. 70–. DOI: 10.1109/IPDPS.2004.1303000 (cit. on p. 93).

[127] E. Solomonik and J. Demmel, “Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms,” in *Euro-Par 2011 Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 6853, Springer Berlin Heidelberg, 2011, pp. 90–109, ISBN: 978-3-642-23396-8. DOI: 10.1007/978-3-642-23397-5_10 (cit. on p. 93).

[128] U.S.Department of Energy, “Exascale programming challenges. ascr exascale programming challenges workshop,” 2011 (cit. on p. 93).

[129] A. Kolinov and A. Lastovetsky, “Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers,” in *High-Performance Computing and Networking*, P. Sloot, M. Bubak, A. Hoekstra, and B. Hertzberger, Eds., ser. Lecture Notes in Computer Science, vol. 1593, Springer Berlin Heidelberg, 1999, pp. 189–200, ISBN: 978-3-540-65821-4. DOI: 10.1007/BFb0100580 (cit. on p. 95).

[130] A. Lastovetsky, “On grid-based matrix partitioning for heterogeneous processors,” in *Parallel and Distributed Computing, 2007. ISPDC'07. Sixth International Symposium on*, IEEE, 2007, pp. 51–51 (cit. on p. 96).

[131] Z. Zhong, V. Rychkov, and A. Lastovetsky, “Data partitioning on heterogeneous multicore platforms,” in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, IEEE, 2011, pp. 580–584 (cit. on p. 96).

[132] Z. Zhong, V. Rychkov, and A. Lastovetsky, “Data partitioning on heterogeneous multicore and multi-gpu systems using functional performance models of data-parallel applications,” in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, 2012, pp. 191–199. DOI: 10.1109/CLUSTER.2012.34 (cit. on p. 96).

[133] D. Clarke, A. Ilic, A. Lastovetsky, and L. Sousa, “Hierarchical partitioning algorithm for scientific computing on highly heterogeneous cpu+ gpu clusters,” in *Euro-Par 2012 Parallel Processing*, Springer, 2012, pp. 489–501 (cit. on p. 96).

[134] A. Lastovetsky and J. Twamley, “Towards a realistic performance model for networks of heterogeneous computers,” in *High Performance Computational Science and Engineering*, Springer, 2005, pp. 39–57 (cit. on p. 96).

[135] A. Lastovetsky and R. Reddy, “Data partitioning with a functional performance model of heterogeneous processors,” *International Journal of High Performance Computing Applications*, vol. 21, no. 1, pp. 76–90, 2007 (cit. on p. 96).

[136] A. DeFlumere, A. Lastovetsky, and B. A. Becker, “Partitioning for parallel matrix-matrix multiplication with heterogeneous processors: the optimal solution,” in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, ser. IPDPSW ’12, IEEE Computer Society, 2012, pp. 125–139, ISBN: 978-0-7695-4676-6. DOI: 10.1109/IPDPSW.2012.12 (cit. on p. 96).

[137] A. DeFlumere and A. Lastovetsky, “Optimal data partitioning shape for matrix multiplication on three fully connected heterogeneous processors,” in *Euro-Par 2014: Parallel Processing Workshops*, Springer, 2014, pp. 201–214 (cit. on p. 96).

[138] M. Kondo, “Report on exascale architecture. iesp meeting. japan.,” 2012 (cit. on p. 106).

[139] P. Balaji, R. Gupta, A. Vishnu, and P. Beckman, “Mapping communication layouts to network hardware characteristics on massive-scale blue gene systems,” *Computer Science - Research and Development*, vol. 26, no. 3-4, pp. 247–256, 2011, ISSN: 1865-2034. DOI: 10.1007/s00450-011-0168-y (cit. on p. 108).

[140] Blackford L. S. and Whaley R. C., “ scalapack evaluation and performance at the dod msrcs,” University of Tennessee, Knoxville, TN, Tech. Rep. LAPACK Working Note No. 136, Technical Report UT CS-98-388, 1998 (cit. on p. 111).

[141] (). The gnu scientific library (gsl), [Online]. Available: http://www.gnu.org/software/gsl/ (cit. on p. 115).

[142] J. Calcote, *Autotools: A Practitioner’s Guide to GNU Autoconf, Automake, and Libtool*. No Starch Press, 2010 (cit. on p. 116).

[143] B. W. Kernighan, D. M. Ritchie, and P. Ejeklint, *The C programming language*. prentice-Hall Englewood Cliffs, 1988, vol. 2 (cit. on p. 116).

[144] P. Sack and W. Gropp, "A scalable mpi_comm_split algorithm for exascale computing," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, vol. 6305, Springer Berlin Heidelberg, 2010, pp. 1–10, ISBN: 978-3-642-15645-8. DOI: 10.1007/978-3-642-15646-5_1 (cit. on p. 151).

[145] A. Moody, D. H. Ahn, and B. R. de Supinski, "Exascale algorithms for generalized mpi_comm_split," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, vol. 6960, Springer Berlin Heidelberg, 2011, pp. 9–18, ISBN: 978-3-642-24448-3. DOI: 10.1007/978-3-642-24449-0_4 (cit. on p. 151).

Appendix A

Possible Overheads in the Hierarchical Design

Our implementations of the hierarchical collective operations use MPI_Comm_split operation to create groups of processes. The obvious questions would be to which extent the split operation can affect the scalability of the hierarchical algorithms. Recent research works show different approaches to improve the scalability of MPI communicator creation operations in terms of run time and memory footprint. The research in [144] introduces a new MPI_Comm_split algorithm, which scales well to millions of cores. The memory usage of the algorithm is $O(\frac{p}{g})$ and the time is $O(g\log_2(p) + \log_2^2(p) + \frac{p}{g}\log_2(g))$, where p is the number of MPI processes, g is the number of processes in the group that perform sorting. More recent research work in [145] improves the previous algorithm with two variants. The first one, which uses a bitonic sort, needs $O(\log_2(p))$ memory and $O(\log_2^2(p))$ time. The second one is a hash-based algorithm and requires $O(1)$ memory and $O(\log_2(p))$ time. Having these algorithms, we can utilize MPI_Comm_split operation in our hierarchical design with negligible overhead of creating MPI sub-communicators. There will not be any overhead at all for large messages as the split operation does not depend on the message size.

Appendix B

HiMPI Configuration Parameters

Table B.1: HiMPI Configuration Parameters

Name	Default Value	Description
HIMPI_MIN_MSG	1kB	The minium message size that will be used to generate configuration file
HIMPI_MAX_MSG	16MB	The maxium message size
HIMPI_MSG_STRIDE	2	The step size in the min and max interval
HIMPI_NUM_LEVELS	1	The number of hierarchies
HIMPI_GENERATE_CONFIG	0	Whether generate configuration file or not
HIMPI_OPID	5	Operations ids specify which collective should be benchmarked during HiMPI_Init. Default value is 5 (op_all) which means all the collectives will be benchmarked.
HIMPI_DEBUG	0	Enable debug mode to see debug information during run time
HIMPI_CONF_FILE	None	If specified the optimal number of groups will be read from that file