Partial Aggregation for Collective Communication in Distributed Memory Machines

Dissertation an der

Fakultät für Mathematik, Informatik und Statistik

der

Ludwig-Maximilians-Universität München

vorgelegt von

Roger Kowalewski

München, den 19. Februar 2021

Partial Aggregation for Collective Communication in Distributed Memory Machines

Dissertation an der

Fakultät für Mathematik, Informatik und Statistik

der

Ludwig-Maximilians-Universität München

vorgelegt von

Roger Kowalewski

Gutachter:	Prof. Dr. Dieter Kranzlmüller
	Prof. Dr. Martin Schulz
Tag der mdl. Prüfung:	03.08.2021

München, den 19. Februar 2021

Eidesstattliche Versicherung

(Siehe Promotionsordnung vom 12.07.11, §8, Abs. 2, Pkt. 5)

Hiermit erkläre ich an Eidesstatt, dass die Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist.

Kowalewski Roger

Name, Vorname

München, 18.09.2021

Ort, Datum

Roger Kowalewski

Unterschrift

Abstract

High Performance Computing (HPC) systems interconnect a large number of Processing Elements (PEs) in high-bandwidth networks to simulate complex scientific problems. The increasing scale of HPC systems poses great challenges on algorithm designers. As the average distance between PEs increases, data movement across hierarchical memory subsystems introduces high latency. Minimizing latency is particularly challenging in collective communications, where many PEs may interact in complex communication patterns. Although collective communications can be optimized for network-level parallelism, occasional synchronization delays due to dependencies in the communication pattern degrade application performance.

To reduce the performance impact of communication and synchronization costs, parallel algorithms are designed with sophisticated latency hiding techniques. The principle is to interleave computation with asynchronous communication, which increases the overall occupancy of compute cores. However, collective communication primitives abstract parallelism which limits the integration of latency hiding techniques. Approaches to work around these limitations either modify the algorithmic structure of application codes, or replace collective primitives with verbose low-level communication calls. While these approaches give fine-grained control for latency hiding, implementing collective communication algorithms is challenging and requires expertise knowledge about HPC network topologies.

A collective communication pattern is commonly described as a Directed Acyclic Graph (DAG) where a set of PEs, represented as vertices, resolve data dependencies through communication along the edges. Our approach improves latency hiding in collective communication through *partial aggregation*. Based on mathematical rules of binary operations and homomorphism, we expose data parallelism in a respective DAG to overlap computation with communication. The proposed concepts are implemented and evaluated with a subset of collective primitives in the Message Passing Interface (MPI), an established communication standard in scientific computing. An experimental analysis with communication-bound microbenchmarks shows considerable performance benefits for the evaluated collective primitives. A detailed case study with a large-scale distributed sort algorithm demonstrates, how partial aggregation significantly improves performance in data-intensive scenarios. Besides better latency hiding capabilities with collective communication primitives, our approach enables further optimizations of their implementations within MPI libraries.

The vast amount of asynchronous programming models, which are actively studied in the HPC community, benefit from partial aggregation in collective communication patterns. Future work can utilize partial aggregation to improve the interaction of MPI collectives with acclerator architectures, and to design more efficient communication algorithms.

Kurzfassung

Hochleistungsrechner setzen sich aus einer Vielzahl an Recheneinheiten (engl. Processing Elements, PEs) zusammen, die in durchsatzoptimierten Netzen miteinander verbunden sind, um komplexe wissenschaftliche Anwendungen ausführen zu können. Das anhaltende Wachstum von Hochleistungsrechnern stellt Algorithmen-Designer vor große Herausforderungen. Da sich die durschnittliche Distanz zwischen den PEs in hierarchischen Speichersystemen vergrößert, nimmt auch die Latenz zur Kommunikation von Daten zu. Das gilt insbesondere für kollektive Kommunikationsoperationen, da gegebenenfalls viele PEs in komplexen Kommunikationsmuster interagieren. Obwohl Algorithmen zur kollektiven Kommunikation für gängige Netztopologien optimiert sind, können gelegentliche Verzögerungen nur weniger PEs die Leistung der gesamten Anwendung beeinträchtigen.

Um die Auswirkungen von Kommunikation auf die Gesamtleistung zu reduzieren, werden ausgefeilte Konzepte zur *Latenzversteckung* (engl. Latency Hiding) in parallelen Algorithmen eingesetzt. Das allgemeine Ziel ist Rechenoperationen mit asynchroner Kommunikation zeitlich zu überlappen, wodurch die Gesamtauslastung erhöht wird. Allerdings abstrahieren kollektive Kommunikationsprimitive den Parallelitätsgrad, was den Einsatz von Techniken zur Latenzversteckung einschränkt. Entsprechende Lösungsansätze modifizieren entweder den Algorithmus einer Anwendung oder ersetzen kollektive Kommunikationsprimitive durch entsprechend viele einfachere Kommunikationsoperationen. Dadurch ergeben sich zwar mehr Möglichkeiten zur effizienten Latenzversteckung, allerdings erfordert dies ein tiefgreifendes Verständnis zugrundeliegender Netztopologien in Hochleistungsrechner.

Kollektive Kommunikationsmuster werden typischerweise durch gerichtete azyklische Graphen repräsentiert, wobei die PEs als Knoten Abhängigkeiten mittels Kommunikation entlang der Kanten auflösen. Unser Ansatz wendet mathematische Regeln von binären Operationen und Homomorphismen auf diesen Graphen an, um die abstrahierte Parallelität in kollektiven Kommunikationsprimitive explizit aufzudecken. Wir verfolgen dabei mehrere Strategien, die unter dem Konzept *partieller Aggregation* zusammengefasst sind. Die eingeführten Strategien wurden mit ausgewählten kollektiven Kommunikationsprimitive des Message Passing Interface (MPI) evaluiert, einem etablierten Kommunikationsstandard im Bereich HPC. Die Ergebnisse zeigen, dass *partielle Aggregation* bei kommunikationsintensiven Anwendungen zu einer drastischen Verbesserung der Laufzeit führen kann. Ein Experiment mit verteiltem Sortieren unterstreicht, warum *partielle Aggregation* zu einer effizienteren Latenzversteckung bei kollektiven Kommunikationsprimitive führt.

Neben der Laufzeitverbesserung ermöglicht unser Ansatz auch eine effektivere Integration asynchroner Programmiermodelle, inbesondere in heterogenen Architekturen mit Beschleunigern. Eine stetig zunehmende Anzahl an PEs bedingt optimierte kollektive Kommunikationsprimitive sowie effiziente Latenzversteckung, um eine effektive Auslastung aller PEs und damit Leistungsskalierbarkeit erreichen zu können.

Table of Contents

1 Introduction

Simulation and modeling is an established branch in the scientific methodology of many disciplines. At their core, scientific simulations encode mathematical models in software through algorithms. These algorithms often process large data volumes on advanced computing systems to understand and solve complex problems in science and engineering. As an example, Computational Fluid Dynamics (CFD) is used to investigate and solve problems of fluid flow in arbitrary geometries through numerical analysis and data structures. Today, no airplane, no car, and no train enters the manufacturing process without prior CFD simulation [80]. Another example of scientific simulation is Numerical Weather Prediction (NWP) to obtain weather forecasts, but also to understand extreme weather events and long-term climate models [157].

As scientists are interested in solving larger problems to obtain a better understanding of their models, demands for memory capacity and computational speed grow continuously. Although processor technology has been exponentially improved over the last decades, requirements of large-scale scientific workloads often exceed computational resources on a single desktop machine. To solve these problems in a reasonable amount of time, High Performance Computing (HPC) systems interconnect a large number of processors with high-bandwidth networks for parallel computation.

The question of how to perform parallel computation on such machines, has lead to many different parallel programming models with explicit and implicit concurrency management. Generally, an application is decomposed into units of computation, called tasks. Tasks are mapped to Processing Elements (PEs), describing arbitrary hardware components to execute a stream of instructions, *e.g.* processors or cores. Simultaneous execution of tasks is key to achieve performance, i.e. to minimize total runtime for the entire problem.

Besides the number of simultaneous tasks, i.e. the degree of parallelism, communication between tasks to coordinate data sharing has a strong impact on performance behavior. Understanding the frequency and model of communication, the granularity of synchronization, and how these primitives are supported in parallel architectures, is vital in performance reasoning.

Considering the ongoing growth of HPC systems in terms of PEs, communication between tasks becomes the main performance bottleneck in parallel computation, because the average distance between PEs increases [84]. The following sections emphasize this statement with an overview about recent trends in HPC system architectures, and what aspects of communication limit parallel performance scalability. A summary about established programming models to address these challenges through explicit and implicit concurrency management follows. Based on these insights we identify open research problems to introduce the problem statement of this thesis.

1.1 Hardware Trends in High Performance Computing

Throughout the last decade, HPC system design has changed significantly due to challenges in microprocessor architectures. Traditional performance scaling approaches of processor chips were driven by *Moore's Law* [147] and *Dennard Scaling* [49]. Moore's Law is the observation to double the number of transistors in an integrated circuit every 18–24 months. *Dennard Scaling* describes that power density is constant in a given area of silicon. Both laws enabled an exponential growth of transistors and a continuous rise in clock frequency from one processor generation to the next without a significant increase in power consumption. As a result, computational performance of single-threaded programs steadily increased with comparable small efforts on the application side. However, a combination of economic and mainly physical limits lead to stagnating clock frequencies with significant consequences for hardware and software design [84]. Instead of driving clock speeds, modern processors facilitate a large number of compute cores to further scale performance. We anecdotally emphasize this observation as nobody could buy a modern smartphone without at least 4 cores today[1].

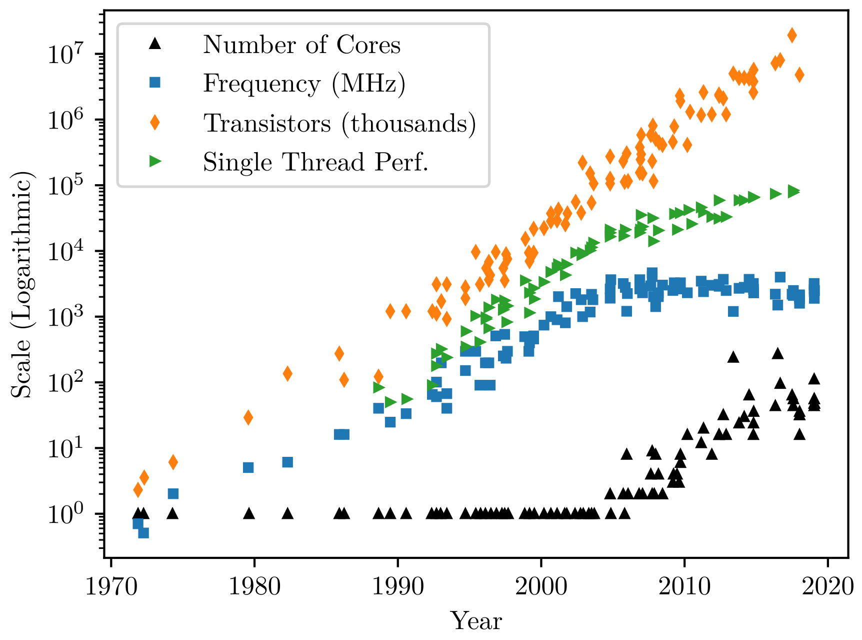


Figure 1.1: 42 years of microprocessor data.
Original data up to the year 2010 by M. Horowitz, F. Labonte et al.
New plot and data collected for 2010–2020 by K. Rupp

Fig. 1.1 visualizes, how performance in microprocessor technology has been improved over time. The x-axis plots a linear time scale, starting from the early 1970s. The y-axis follows a logarithmic scale to show the exponential improvements of most important characteristics in multiprocessors. The still increasing number of transistors (diamonds) reveals that Moore's Law will continue although this is controversially debated among computer scientists [84]. However, the breakdown of *Dennard Scaling* around 2006 established a power wall and lead

[1]https://en.wikipedia.org/wiki/Comparison_of_smartphones (accessed March 2020)

to flattened clock frequencies (blue squares). We can still observe small increases with single-threaded performance in recent years (right aligned triangles), but these improvements were achieved with effective power management and dynamic frequency scaling. The main driver for a still exponential increase results from a strong growth in the number of compute cores on a single chip (black upper triangles). Given a limited energy budget, computer scientists expect next generation HPC systems to be equipped with many low frequency processors [23, 84].

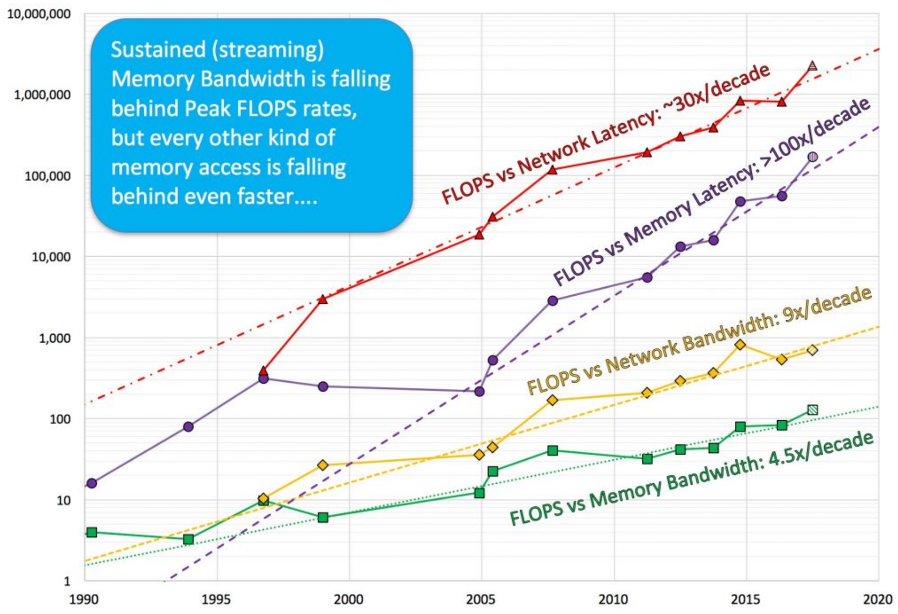


Figure 1.2: Relative performance of floating point arithmetic, memory and network bandwidth over the last 25 years [141].

A consequence of the increasing core count is substantial bandwidth tapering in hierarchical memory subsystems. Fig. 1.2 visualizes relative performance improvements of floating point arithmetic, measured in FLOPS[2], and data movement technologies. The x-axis plots a linear time scale, while the y-axis follows a logarithmic scale to compare exponential performance improvements in computation and data movement technologies. It is evident from this chart that the relative cost of computation to data movement has been further skewed in favor of computation. Although there is still significant progress in memory and network technologies, the available bandwidth cannot satisfy a rapidly increasing core count, leading to substantial imbalance between computational and memory throughput [78, 84].

The impacts on HPC system design due to an increasing machine imbalance are already visible. The majority of last generation HPC clusters interconnected a high number of *homogeneous* multi-core compute nodes. Although efficient performance scaling has been achieved in these systems, many experts emphasize a high energy consumption due to expensive data movement across the memory hierarchy and large networks [121, 176]. On modern processors, each core performs tens of arithmetic operations per cycle, but a single core-to-core communication accumulates to at least tens of cycles of latency [146]. Therefore, maximizing data locality and minimizing off-chip communication are essential

[2]Floating point operations per second.

pillars to push performance beyond current levels. With *exascale systems*[3] as the next milestone in HPC, hardware architects design distributed memory nodes featuring massive shared memory parallelism [23].

As an example, we compare the previous and current HPC system generations of the LLNL Computing Facility, called Sequoia and Sierra to illustrate this. The Sierra supercomputer, launched in 2019, delivers ten times the peak performance of its predecessor Sequoia while cutting the number of total compute nodes from 98 000 to 4600 [193]. A single compute node features two IBM Power 9 Central Processing Units (CPUs), coupled with four Nvidia Graphical Processing Units (GPUs) in a coherent shared address space. As the majority of top ranked supercomputers feature a similar design with combinations of CPUs and accelerators, this trend is expected to continue [181].

1.2 Communication as the Main Performance Bottleneck

Efficient parallel programming models, either through explicit or implicit concurrency management, are necessary to scale application performance on HPC systems. This means that any performance critical algorithm and implementation needs to be parallelized efficiently. However, understanding and harnessing many-core machines places great challenges on algorithm designers due to complex interactions between PEs in hierarchical memory subsystems. Increasing parallelism often exposes more communication per task, or more frequent synchronization of processors.

The impact of communication costs on performance depends on an algorithm's *operational intensity*. Operational intensity defines the number of instructions per memory transfer, i.e. the ratio between computation and communication. The higher the operational intensity, the lower the impact of communication bottlenecks [198]. Hence, parallel algorithm design resolves around maximizing operational intensity. However, many applications have an inherently low operational intensity, either due to unstructured memory access patterns or due to low computation costs. Efficient communication mechanisms are crucial in these cases to achieve performance. Common examples in scientific computing are sparse linear algebra kernels, stencil computations or n-body particle simulations.

Computer scientists distinguish two fundamental communication paradigms: *Shared* memory and *distributed* memory. Shared memory communication is commonly used in Symmetric Multiprocessor (SMP) architectures, where PEs have *symmetric* access to a single shared address space. Therefore, data is implicitly shared through native memory accesses. While shared memory is often perceived as a natural abstraction particularly for inexperienced programmers, it does not scale to a large number of PEs in HPC systems.

Instead, most HPC systems interconnect many SMP nodes, each with a private memory, in high-bandwidth networks. Because global memory is physically distributed, computer scientists refer to distributed memory machines. The most commonly used programming model for distributed memory machines is message passing. Message passing compels programmers to communicate shared data explicitly through sending/receiving messages, usually in software.

[3]Systems featuring a peak performance of $\approx 10^{18}$ floating point operations per second.

Whether communication happens in shared or distributed memory, the main goal in parallel algorithms is to maximize communication throughput. Optimally, communication throughput is bounded by memory or network bandwidths. *Bandwidth* defines the maximum rate of data transfer across a given path, which depends on the *channel bandwidth* and *injection bandwidth*. *Channel bandwidth* is the maximum data volume a communication channel can carry at a given time, e.g. a link between two hosts in a network. On the other hand, *injection bandwidth* defines the maximum data rate a communication endpoint can send/receive at a given time. Because message passing protocols are commonly managed in software, some PEs are *occupied* in message processing, which consumes CPU cycles and prevents progress of computation. If such *occupancy* is incurred for every data word in a message, or for each communication event, this limits effective communication bandwidth. HPC researchers characterize this as communication overhead.

If an application is executed on a large number of SMP nodes, communication overhead amplifies negative performance impacts, as synchronization delays between any two PEs can propagate to other communications. This particularly occurs in message passing applications following a bulk-synchronous lockstep model. In each lockstep, PEs process independent tasks in parallel, communicate respective output and synchronize before continuing with the next lockstep [192]. Although communication and synchronization patterns depend on task interactions, researchers have identified common communication patterns in HPC applications, e.g. broadcast, reduce, or all-to-all communications. Because these communication primitives involve interaction between all PEs, they are called *collective* communications. Collective communication interfaces are important building blocks in shared and *distributed memory* algorithms. As collective communication patterns contribute a significant fraction of overall communication costs in parallel programs, HPC researchers study and steadily improve collective communication algorithms to minimize communication overheads [24, 99].

1.3 Problem Statement

Taking the aspects mentioned above into account, we identify three important factors to obtain scalable performance efficiency.

- The degree of parallelism exploited by algorithms.
- the *occupancy* of compute cores, and,
- the effectiveness of *latency hiding*.

The degree of parallelism characterizes the number of simultaneous tasks and, therefore, strongly depends on parallel algorithm design. The other two terms, *latency hiding* and *occupancy* depend on supported communication mechanisms in parallel programming models.

Latency hiding describes the ability to *hide* communication overhead by overlapping communication with computation. Therefore, effective latency hiding *occupies* idle PEs with computation, which would otherwise be forced to wait until pending communication is complete.

In shared memory architectures (e.g. SMPs), latency hiding is commonly managed in hardware through caches and instruction prefetching. However, programmers still need to manage data locality, as cache coherency protocols and off-chip communication can cause severe performance overhead. Programming abstractions with explicit communication (e.g. message passing) make latency hiding even more difficult. It usually needs to be supported in software through asynchronous communication, which is integrated into properly designed algorithms.

To better manage the growing complexity of HPC machines due to an increasing number of PEs, HPC researchers have proposed a vast amount of data-centric interfaces to express task parallelism. Based on data dependencies, sophisticated runtime systems handle scheduling of work and coordination between PEs via asynchronous data movement and synchronization. Whether data movement happens in shared or distributed memory is abstracted from applications. Therefore, programming abstractions with implicit concurrency decouple control and data flow, enabling automatic load balancing and proper latency hiding [1, 16, 139, 201]. However, specifying parallelism only through data dependencies often results in a limited view of global communication patterns, especially with respect to collective communications. This may increase communication costs especially in distributed memory machines, where data is shuffled across a large number of PEs.

For this reason, programmers commonly rely on hybrid programming models with both explicit and implicit concurrency [6, 93]. The main challenge results from integrating different interfaces, which often expose conflicting semantics due to orthogonal programming models. For example, message passing favors coarse-grained parallelism to maximize effective communication bandwidth, while multi-threaded concurrency in a shared memory model advocates fine-grained parallelism to improve computation throughput.

Researchers address semantic mismatches in hybrid programming models from various perspectives. The Message Passing Interface (MPI) [150], which is an established standard for distributed memory applications in scientific computing, has received much attention to improve interoperability with shared memory programming abstractions. Many papers present implementation-specific optimizations to improve multi-threaded performance. Other papers extend the set of point-to-point communication primitives to improve interaction with fine-grained task parallelism. However, only few of these works consider collective communication primitives. While collective communications are optimized for network-level parallelism their interfaces abstract the inherent parallelism, limiting the integration of latency hiding techniques.

Even if communication is asynchronous, message buffers usually cannot be accessed before the collective operation is complete. To mitigate communication costs, it is often useful to perform computations with partially completed message transmissions, and aggregate them into a single result. A common example is global reduction to express parallel summations, which is among most important collective communication patterns. Scalable reduction algorithms rely on properties of binary operations to interleave computation and communication. However, partial aggregation is not only useful in global reduction but in many other collective communication patterns. Examples are neighborhood communications [104] or distributed sort algorithms [129], which we analyze in our work.

Our approach breaks up the monolithic interface of collective communication primitives. A collective communication pattern is described as a Directed Acyclic Graph (DAG) where a set of PEs, represented as nodes, resolve data dependencies through communication along the edges. We introduce *partial aggregation* to improve latency hiding in collective communication. Based on mathematical rules of binary operations and homomorphism, we expose abstracted data parallelism in collective communication primitives to the user.

We aim at improving the performance of HPC applications due to improved latency hiding capabilities and better interoperability in hybrid programming models. With our approach, users obtain fine-grained parallelism without loosing the level of abstraction provided by collective primitives. To obtain a systematic approach, we phrase the following research question (RQ):

> RQ*: How does the integration of partial aggregation improve collective communications?*

We decompose the overall research problem into smaller subquestions (SQs):

SQ1 Which applications are sensible to the performance of collective communications?

SQ2 Which deficiencies in collective communication primitives limit parallel efficiency?

SQ3 How can partial aggregation expose a high degree of parallelism in collective communications?

SQ4 How does partial aggregation improve performance efficiency in collective communications?

We first review established programming models with explicit and implicit concurrency management to provide an understanding of parallel computation in HPC architectures. Based on these concepts, we analyze interfaces and algorithms for collective communication. We use a simple communication cost model to quantify parallelism and latency hiding potential in collective communications to answer *SQ1*.

To answer *SQ2*, we discuss common latency hiding techniques, and how they are combined with low-level communication protocols. Researchers have proposed asynchronous programming abstractions to structure parallel algorithms, while sophisticated runtime systems handle latency hiding and scheduling of work. However, their interoperability with collective communication primitives is limited due to monolithic interfaces, which do not expose inherent parallelism.

To address these limitations, we propose partial aggregation for collective communications, which leads to *SQ3*. Concepts of function decomposition enable to break up the monolithic interface for collective communication. We discuss several optimizations, following the idea to explicitly expose parallelism in collective communication patterns. To demonstrate the applicability of partial aggregation concepts, we evaluate a subset of collective communication primitives in the most recent MPI-3 standard. Relying on MPI protocols ensures that the presented results can be compared to existing approaches and that scientific applications can immediately benefit from our contributions.

This leads to *SQ4*, where we first showcase a prototypical implementation of partial aggregation in collective communication primitives. To evaluate the performance impact, a detailed case study with microbenchmarks and common scientific applications reveals significant speedups in communication-intensive use cases.

1.4 Contributions

The main contributions in this thesis are summarized as follows:

- We compare important important parallel programming models to structure parallel computation. Understanding fundamental concepts from a theoretical perspective is necessary to establish a systematic approach for our approach. To answer the proposed research problem, we borrow constructs of function parallelism, which is common in data-centric programming models, e.g. Map-Reduce.
- We analyze state-of-the-art message passing abstractions, which support collective communication primitives in distributed memory machines. Our findings show that monolithic interfaces prevent efficient latency hiding in collective communication patterns. However, collective communication often imposes implicit synchronization of participating processors due to data dependencies in the communication pattern. We use a simple communication cost model to quantify and understand these bottlenecks.
- We discuss the complexity and design of collective communication algorithms in more detail to understand hidden data parallelism and how implicit synchronization limits parallelism. To integrate latency hiding techniques in collective communication, we propose partial aggregation concepts. Partial aggregation effectively enables to compute partial results, while collective communication is in progress. Idle PEs can proceed with computation, which would otherwise be forced to wait until collective communication completes. To guarantee a correct output after combining partial results, we apply concepts of function decomposition.
- We demonstrate with an efficient prototypical implementation how partial aggregation can be integrated into existing collective communication primitives. We rely on the predominant MPI standard in scientific computing. Because existing collective communication semantics are not modified, our implementations can be easily integrated into arbitrary application codes. To achieve practical performance and portability, the implementation relies on modern C++ concurrency features. A primary design goal is to ease the interaction with other parallel programming abstractions, in particular with multi-threaded runtimes in shared memory.
- We demonstrate how partial aggregation improves performance in data intensive use cases. Benchmarked are a stencil application and a distributed sort algorithm, which are ubiquitous in scientific computing. Both applications have been the subject of numerous research papers to maximize performance efficiency. With partial aggregation, performance degradation is reduced, while a simpler interface to explicitly express asynchrony for better communication-computation overlap is provided. In conclusion, our approach contributes partial aggregation concepts to improve parallel efficiency of collective communication patterns in hybrid programming models.

Authors Preliminary Work

Below we list our *own* publications as lead author prior to this thesis and their relation to the presented work.

- R. Kowalewski, P. Jungblut, and K. Fürlinger, “Engineering a Distributed Histogram Sort,” in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, Sep. 2019

  Summary: This publication presents a novel distributed sort algorithm in a hybrid message passing and Partitioned Global Address Space (PGAS) model. Presented results discuss performance bottlenecks in collective communication, in particular with large message transfers. Contributions of this dissertation significantly reduce these bottlenecks, as demonstrated in Ch. 7.

  Own contribution: Concepts in this dissertation significantly reduce synchronization bottlenecks in collective communication patterns. Therefore, distributed sort serves as an evaluation use case in this dissertation. The author of this dissertation designed the sort algorithm and contributed the major fraction of the implementation.

  Other Contributors: P. Jungblut contributed a minor part of the implementation and supported the practical evaluation. Likewise, K. Fürlinger joined for supporting practical evaluation.

- R. Kowalewski, T. Fuchs, K. Fürlinger, and T. Guggemos, “Utilizing Heterogeneous Memory Hierarchies in the PGAS Model,” in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, IEEE, Mar. 2018

  Summary: This publication evaluates latency and bandwidth trade-offs of scientific applications in heterogeneous memory models in PGAS. We present semi-formal data placement strategies from common memory access patterns to specific memory types. Insights of these publication can be utilized in user-defined partial aggregations, as elaborated in Ch. 5.

  Own contribution: The author of this dissertation proposed the major part of this paper, including the approach, concepts and implementation to evaluate practical performance.

  Other contributors: T. Fuchs, K. Fürlinger and T. Guggemos supported the practical evaluation on different HPC systems.

Collaborative Contributions

The following publications are within the scope of the presented work through involvement in co-authorship.

- J. Schuchart, R. Kowalewski, and K. Fuerlinger, “Recent experiences in using MPI-3 RMA in the DASH PGAS runtime,” in *Proceedings of Workshops of HPC Asia*, ser. HPC Asia ’18, ACM, Association for Computing Machinery, Jan. 31, 2018

  Summary: This publication reveals issues and challenges in the MPI+PGAS model. We particularly focus on performance assessment of one-sided communication in most recent HPC systems.

  Own contribution: The author of this dissertation contributed collective communication algorithms (all-to-all, all-gather) based on one-sided communication primitives.

These algorithms were part of the benchmark suite in the paper and have been integrated into this dissertation.

Other Contributors: J. Schuchart proposed the initial idea and methodology. K. Fürlinger joined for supporting the experimental evaluation.

- F. Mößbauer, R. Kowalewski, T. Fuchs, and K. Fürlinger, "A Portable Multidimensional Coarray for C++," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, IEEE, Mar. 2018

  Summary: This publication adapts the co-array abstraction, originally proposed in Fortran language, into a C++ PGAS model.

  Own contribution: The author of this dissertation contributed a memory management concept to support the co-array abstraction in heterogeneous memory environments. The memory management concept is also published as a major contribution of this dissertation [126].

  Other contributors: F. Mößbauer proposed the initial idea of this publication. T. Fuchs supported and K. Fürlinger supported in the experimental evaluation.

- K. Fürlinger, R. Kowalewski, T. Fuchs, and B. Lehmann, "Investigating the performance and productivity of DASH using the Cowichan problems," in *Proceedings of Workshops of HPC Asia on - HPC Asia '18*, ACM, ACM Press, 2018

  Summary: This publication evaluates the DASH framework, which is a PGAS abstraction in distributed memory, for various scientific kernels included in the Cowichan benchmark suite.

  Own contribution: The Cowichan benchmark suite includes a distributed sort. The implementation in the paper is based on contributions of this dissertation [129].

  Other contributions: K. Fürlinger proposed the initial idea and methodology of this publication. T. Fuchs and B. Lehmann supported in the experimental evaluation.

- K. Fuerlinger, T. Fuchs, and R. Kowalewski, "DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Ieee, IEEE, Dec. 2016

  Summary: This publication presents concepts of DASH as a hybrid MPI+PGAS programming abstraction for scientific applications. We present fundamental data structures and algorithms along with an experimental performance evaluation.

  Own contribution: The author of this dissertation ported some of the evaluated benchmarks in the paper to the DASH programming abstraction. Further academic work lead to major contributions of this dissertation [126, 129].

  Other contributions: K. Fürlinger proposed initial ideas and overall design of this dissertation. T. Fuchs contributed own concepts and supported in experimental evaluation.

- K. Fuerlinger, J. Gracia, A. Knüpfer, D. Hünich, T. Fuchs, P. Jungblut, R. Kowalewski, and J. Schuchart, "DASH - Distributed Data Structures and Parallel Algorithms in a Global Address Space," in *Software for Exascale Computing - SPPEXA 2016-2019*, ser. Lecture Notes in Computational Science and Engineering, H.-J. Bungartz, S. Reiz, B. Uekermann, P. Neumann, and W. E. Nagel, Eds., Cham: Springer International Publishing, 2020. DOI: 10.1007/978-3-030-47956-5

  Summary: This publication presents research results of all contributors involved in the DASH project which has been funded by the *Deutsche Forschungsgemeinschaft* (DFG).

  Own contribution: The author contributed a chapter about distributed sort algorithms in the PGAS model which relies on major contributions of this dissertation [129].

  Other contributions: Each co-author contributed one chapter about research results which have been achieved in the scope of the DASH project.

Beyond the Scope of this Dissertation

The following publications are not directly associated, but were written alongside efforts in the presented work.

- R. Kowalewski and K. Fürlinger, "Nasty-MPI: Debugging Synchronization Errors in MPI-3 One-Sided Applications," in *Euro-Par 2016: Parallel Processing*, ser. Lecture Notes in Computer Science, P.-F. Dutot and D. Trystram, Eds., vol. 9833, Cham: Springer International Publishing, 2016, pp. 51–62. DOI: 10.1007/978-3-319-43659-3_4

  Summary: In this publication we design and evaluate a dynamic linkage library to validate correctness of MPI-3 RMA communications. Based on a heuristic approach we exploit corner cases in the MPI-3 standard to manifest latent synchronization errors at runtime.

- R. Kowalewski and K. Fürlinger, "Debugging Latent Synchronization Errors in MPI-3 One-Sided Communication," in *Tools for High Performance Computing 2016*, C. Niethammer, J. Gracia, T. Hilbrich, A. Knüpfer, M. M. Resch, and W. E. Nagel, Eds., Cham: Springer International Publishing, 2017, pp. 83–96. DOI: 10.1007/978-3-319-56702-0_5

  Summary: This publications extends concepts and capabilities of aforementioned paper [127]. Results are published in a special issue proceeding on correctness tools for HPC.

- P. Jungblut, R. Kowalewski, and K. Fürlinger, "Source-to-Source Instrumentation for Profiling Runtime Behavior of C++ Containers," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, IEEE, Jun. 2018

1.5 Thesis Structure

The structure follows from decomposing the research question, as elaborated in Sec. 1.3. Ch. 2 summarizes fundamental concepts of communication complexity and compares established parallel programming models to organize parallel computation. We discuss differences between Bulk Synchronous Parallelism (BSP) and Map-Reduce as the most popular abstractions in HPC and machine learning, respectively. Although the BSP model is more powerful than Map-Reduce, its influence on application design in HPC lead to monolithic interfaces for collective communication primitives, limiting data parallelism.

Ch. 3 summarizes related work and describes state-of-the-art programming models, which support collective communication in distributed memory machines. To understand the significance of latency hiding, we quantify communication overhead in message passing protocols. Based on presented models, we review latency hiding techniques to mitigate associated communication overhead. Although HPC experts have studied latency hiding strategies in message passing protocols for years, our findings show that interfaces for collective communication patterns either prevent or complicate latency hiding.

We propose a solution in Ch. 5 to address these shortcomings. To understand the degree of parallelism in collective communication patterns, we first discuss design and complexity of collective communication algorithms. In the next step, we apply concepts of function composition and propose partial aggregation for collective communication primitives. Partial aggregation enables to operate on partial results, while collective communication is still in progress. Based on concepts of homomorphism and decomposable functions, we show that the combination of partial results still yields a correct final result. In contrast to current approaches, users obtain the full degree of parallelism without diminishing the abstraction level of collective primitives. Another advantage of partial aggregation is that it allows to relax strong guarantees on message transmission, which cannot be achieved with current collective communication primitives.

In Ch. 6, we present a prototypical implementation and integrate partial aggregation into collective communication primitives. We rely on communication protocols of the most recent Message Passing Interface (MPI) specification, which ensures that our results are comparable to related work. To achieve practical performance, we utilize recently proposed C++ concurrency concepts. A major design goal is to ease interaction with established programming models, which can take advantage of partial aggregation concepts in existing application codes.

Ch. 7 conducts a detailed performance evaluation. We demonstrate for applications which communicate mostly small data volumes over the network, the additional runtime overhead of partial aggregation does not result in any significant performance degradation. However, if message sizes increase, the ability to operate on partial results increases parallelism and, thus, improves performance. In a distributed sort benchmark with large data volumes, we reveal how partial aggregation techniques lead to significant speedups in a personal *all-to-all* exchange.

With our findings at hand, we conclude in Ch. 8 that partial aggregation benefits not only communication in distributed memory. Collective communication become increasingly relevant in accelerator architectures. The concept of partial aggregation provides another possibility to integrate accelerator devices into collective communications, which is a widely discussed topic among HPC experts.

2 Preliminaries

In contrast to sequential executions, parallel executions add the dimension of concurrency. Concurrency is the ability to solve independent tasks of a single program out-of-order or in partial order, without affecting the final outcome [134]. Concurrency allows simultaneous execution of tasks on PEs of parallel systems, with the main objective to reduce total execution time compared to a sequential execution [84].

In scientific computing, a common approach for concurrency is to decompose a large computational domain into smaller independent tasks. Communication and synchronization are essential to model interactions between tasks but limit possible parallelism. Therefore, minimizing communication and synchronization is crucial to achieve performance [73].

This chapter summarizes concepts of concurrency with a particular focus on communication mechanisms and their implications on parallel algorithm design. A comparison of parallel programming models establishes a fundamental understanding of communication complexity, emphasizing the importance of collective communication patterns.

2.1 Multiprocessor Architectures

HPC systems assemble nodes of multiprocessor architectures, ranging from few to hundreds of PEs on a single node, interconnected through a network. To process tasks in parallel, PEs coordinate through communication of shared data. Although communication interfaces differ among specific hardware architectures, computer scientists distinguish between two essential paradigms: *Shared memory* and *distributed memory.*

Experts in academia and industry have proposed standards for programming shared and distributed memory. Either of them has its strengths and weaknesses, depending on supported communication mechanisms and application characteristics. To maximize benefits, efficiently parallelized applications rely on a hybrid programming model. Below we summarize key characteristics of multiprocessor architectures, along with common programming abstractions in HPC.

2.1.1 Shared Memory Architectures

Shared memory architectures assemble SMPs with relatively small number of PEs. All PEs have *equal* access to a single *shared* address space. Communicating shared data is implicit through native read and write accesses in shared memory. Coordinating concurrent read and write accesses to shared data requires explicit synchronization mechanisms.

If all PEs can access shared memory with uniform latency and bandwidth costs, computer scientists refer to Uniform Memory Access (UMA) architectures. This centralized protocol does not scale with a large number of PEs, as increasing bandwidth demands would incur high latency.

For this reason, modern multicore processors follow a Non-Uniform Memory Access (NUMA) design. While the global view to a shared address space is preserved, available memory is hierarchically partitioned into multiple memory domains. Memory domains are interconnected in on-chip communication networks, e.g. rings or meshes. The drawback of this design are different communication costs, depending on whether accessed data words reside in *local* or *remote* memory domains. Observing different communication costs is known as NUMA effects. Minimizing NUMA effects is crucial in parallel algorithms to achieve performance efficiency [84].

Shared memory is often perceived as the most natural programming abstraction for programmers. Communication and cache coherence protocols are efficiently managed in hardware. The communication-agnostic model yields high communication throughput, especially with small data items and irregular memory accesses. It further simplifies the integration of thread-level parallelism, as threads can interact in shared memory with arbitrarily complex communication patterns. In scientific computing, OpenMP defines a standardized programming interface for parallel work sharing on shared memory multiprocessors [154].

2.1.2 Distributed Memory Architectures

Distributed memory architectures are often called *shared nothing* architectures. Each PE has access to a private memory address space and is interconnected in a high bandwidth network to other PEs. This allows simpler hardware design compared to cache coherent shared memory architectures. However, it requires explicit communication mechanisms, which are implemented in software [84].

The most-used programming paradigm for distributed memory machines is message passing. In contrast to shared memory, synchronization is *implicit* through sending and receiving messages. Both sides agree on the communication of shared data. For this reason, message passing is often called two-sided communication. The explicit model of communication enables efficient reasoning about algorithmic complexity. Therefore, it often leads to a better software design, as opposed to shared memory architectures [73]. The disadvantage is that complex communication patterns are difficult to implement, as communication peers are sometimes unknown prior to runtime. To support these cases, message passing protocols require polling mechanisms to probe for incoming messages from the network. Polling often exposes non-negligible software overhead, reducing possible message rate.

Although message passing is commonly used in shared memory, it integrates well in shared memory. If communication peers have access to a coherent shared memory, sending a message can be implemented as a fast memory copy, Ch. 3 summarizes various optimization techniques, which exploit shared memory to improve message rate.

It is even possible to realize a shared memory model on top of message passing protocols. Modern Network Interconnects (NICs) are equipped with Remote Direct Memory Access (RDMA) capabilities [110]. RDMA enables data access from a network host to another, without involvement of the message passing software stack. The advent of RDMA capabilities in HPC networks lead to one-sided communication models. Simple PUT/GET primitives enable a sender to initiate a message exchange, without requiring the receiver to post a matching receive operation. In contrast to traditional two-sided communication, one-sided communication semantically decouples data movement and synchronization. This lead to the rise of data-centric programming abstractions in distributed memory, most notably the PGAS model [10, 116].

An example for a PGASs programming abstraction is DASH, which provides a collection of array-like data structures in distributed memory machines [62]. PEs can access any data item in these data structures, even if the respective elements reside on another network host. Because *remote* memory accesses are much more expensive than *local* shared memory accesses, PGAS exposes an explicit notion of locality to minimize network traffic. Accesses to *local* data are available through shared memory semantics, while *remote* memory accesses transparently map to one-sided communication calls. The PGAS model is a reasonable alternative in highly irregular communication patterns and small message sizes, where the software overhead of message passing degrades overall throughput [19].

Independent of the communication mechanism, it is often useful to analyze communication complexity among a set of PEs. A realistic model for distributed memory machines is the Hockney model [91]. The time to transmit a message of size m machine words is approximated as follows:

$$\alpha + m\beta \tag{2.1}$$

The parameter α models the startup overhead per message and β the time to communicate a single machine word, i.e. the reciprocal of a system's network bandwidth. While this model is precise enough to analyze various communication patterns, it is too detailed to analyze large scientific application codes. HPC researchers have proposed established parallel programming models, which we describe in the next section.

2.2 Parallel Programming Models

In contrast to traditional multiprogramming or transaction-oriented computing, where independent tasks are executed in parallel, parallel computing decomposes applications into tasks, which are executed in a coordinated fashion to solve large scientific problems. Understanding the interactions between cooperative tasks is fundamental to determine performance scalability.

Performance scalability refers not only to minimize execution time on a specific system. To drive scientific progress, it is vital that executed algorithms in applications efficiently scale with future system sizes. Relevant performance aspects include the amount of parallelism, task sizes, the frequency of communication, and nature of synchronization. To reason about these aspects, HPC experts rely on parallel programming models to hide low-level details specific to a particular architecture. We compare the most important parallel programming models in scientific literature:

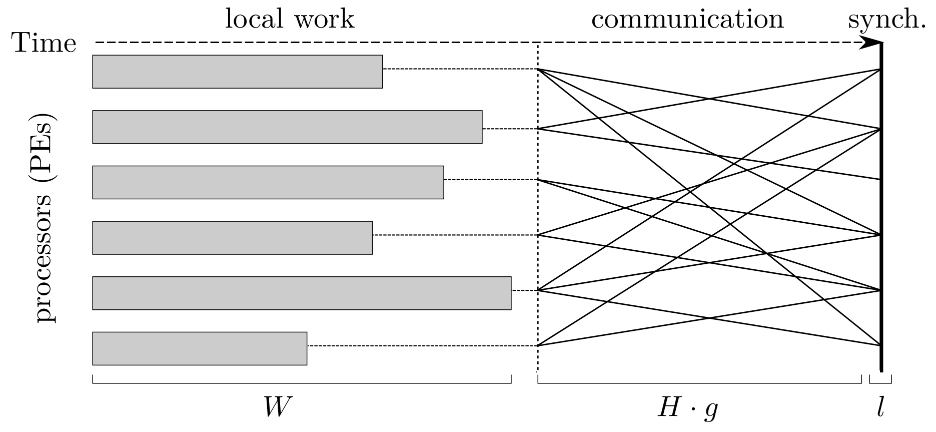


Figure 2.1: A single superstep in the BSP model.

- Parallel Random Access Machine (PRAM)
- Bulk Synchronous Parallelism (BSP)
- MapReduce

2.2.1 Parallel Random Access Machine

The Parallel Random Access Machine (PRAM) model was proposed already in 1978 by extending the traditional RAM, or *Von Neumann* model, to parallel computers [59]. It consists of p sequential PEs, each having access to a global shared memory for communication. It is assumed that all PEs run with a synchronized clock, and any PE can access any memory location in a single step. As this model ignores parallelization costs, it is an ideal model to quantify computational complexity in parallel algorithms.

Although this model has been well accepted in computer science, especially for algorithm design, it is impractical for real-world machines. Mainly, because PEs are assumed to run synchronously and communication in global memory is essentially free, i.e. latency and bandwidth costs are ignored.

2.2.2 Bulk Synchronous Parallelism

The BSP model reflects the design of mainstream parallel architectures better [192]. A BSP machine consists of p PEs, each equipped with private memory, and a fully connected network to route messages between PEs. Since PEs run asynchronously, a synchronization facility is necessary.

BSP algorithms proceed in supersteps, as visualized in Fig. 2.1. The figure shows timelines for 6 PEs in a single superstep, where time progresses from left to right. One consists of three phases. In the first phase, PEs receive some input and perform asynchronous computation (work). In the second phase, the produced output is communicated in an arbitrary complex communication pattern between all PEs. Computation and communication can be interleaved, as PEs initiate message transmission as soon as local work has completed. Because the first two phases are both asynchronous, all PEs need to synchronize in the third phase, before continuing with the next superstep. The fact that PEs can communicate with each other provides explicit control of data distribution.

To characterize a physical system, BSP defines the bandwidth inefficiency g of the underlying network and a synchronization periodicity l. In each superstep, a single PE can send at most h messages and receive at most h messages, each of a fixed size in bytes. This is called the h-relation. Let h_s be the maximum input and output, and w_s the maximum amount of work among all PEs in superstep s. An algorithm running in S supersteps has costs $W = \sum_{s=1}^{S} w_s$ and $H = \sum_{s=1}^{S} h_s$. Accordingly, an estimated execution time T_{bsp} on a physical system is defined as [192]:

$$T_{bsp} = W + H \cdot g + S \cdot l \tag{2.2}$$

The synchronization periodicity l is typically much larger than the bandwidth inefficiency g. Hence, an efficient BSP algorithm minimizes the number of supersteps S, while still optimizing computation and communication costs. Because g and l vary across different systems, algorithm design focuses on minimizing W, H and S.

BSP is a widely accepted communication model in HPC as it matches well with the semantics of message passing and the structure of scientific applications. At the same time, it encourages programmers to design algorithms in a bulk synchronous manner, possibly introducing too many supersteps. Synchronizing a large number of PEs is costly and wastes computation time. To saturate computational capacity on modern many-core architectures, asynchronous algorithms aim at eliminating these synchronization steps.

2.2.3 MapReduce

Despite decades of experience in performance engineering for scientific computing the global industry players have developed their own ecosystems to fit their needs for processing huge data volumes. Google popularized MapReduce [47] which is now a major programming model paradigm for machine learning applications. Open source implementations like Apache Hadoop [8] and, more recent, Apache Spark [202] receive much attention due to the good integration into the Java Virtual Machine (JVM) ecosystem.

The basic model assumes a parallel machine consisting of p PEs, each equipped with a private memory. In addition, all PEs have access to a globally shared memory in a high bandwidth network. Based on concepts of function parallelism, computation is expressed as a sequence of map, shuffle and reduce steps, each operating on a set $X = \{x_0, x_1, \dots, x_n\}$ of values:

- A *map* step applies a function F to each value x_i to produce a finite set of key-value pairs (k, v). To allow parallel computation, F must only depend on x_i.
- A *shuffle* step collects all key-value pairs to collapse them into a set of lists $L_k = \{k; v_1, v_2, \dots\}$. That is, each list L_k groups values by some key k, as assigned in the map step.

- A *reduce* step applies a function R to each list L_k to produce a set of values $Y = \{y_1, y_2, ...\}$. Function R is allowed to be sequential on L_k, however must be independent of any other list $L_{k'}$, where $k' \neq k$.

Generally, the output of a single map-shuffle-reduce step can be used as input in a subsequent round. Because the shuffle step requires communication among possibly all PEs, an efficient algorithm minimizes the number of rounds.

Given the popularity of MapReduce, it is interesting to understand differences with the BSP model. Pioneering work establishes strong connections between both models from a theoretical perspective [70, 155]. Both MapReduce and BSP design parallel algorithms in a rather coarse-grained fashion to interleave communication and computation. Both models can be used to design algorithms on low-end parallel systems, connected via some means of point-to-point communication. And most important, both models rely on intermediate locksteps to synchronize PEs.

Likewise, there are notable differences. After completing a single map/reduce task, the worker on which a particular task was running clears its private memory. Any shared data for subsequent rounds needs to be written to global memory. In contrast, BSP preserves local memory for reuse in subsequent supersteps.

The probably most unusual fact for HPC programmers is that MapReduce does not have an explicit notion of a *place*. The BSP model assumes a fully connected network, where data can be directly exchanged between any pair of PEs. While MapReduce routes data from map to reduce tasks, there is no information which PE will process which portion of the input data. This contradicts with the concept of data locality to load balance work in BSP. Instead, MapReduce requires a single *master* process, which distributes the dataset to map tasks in the very first round. In many cases, an effective hashing algorithm will find a *good* distribution to balance the work load among all PEs. However, the limited control about data placement probably lead some researchers to the conclusion that MapReduce does not support indexed searches [50]. In later work, this statement has been corrected through an efficient multi-search algorithm in a balanced search tree [70].

Although MapReduce provides a simple programming model, limitations in its expressiveness complicate load balancing and communication efficient algorithms. While this is feasible for loosely coupled applications, scientific applications often exhibit static and regular interactions between tasks. In these cases, MapReduce is too restricted to scale on large-scale parallel systems.

Similarly, machine learning applications, where MapReduce is the de-facto programming model, face performance challenges on very large datasets. To improve scalability on massively parallel systems, communication-intensive problems can be better solved with explicit concurrency management, i.e. message passing or multi-threading. [21, 142].

Concluding this section, MapReduce is a powerful abstraction for data-intensive problems, where locality is not the primary concern. Ch. 5 revisits fundamental concepts of MapReduce and functional parallelism to design partial aggregation in collective communication patterns.

2.3 Principles of Parallel Algorithm Design

Scientific applications decompose a computational domain into smaller tasks to process them in parallel. Irrespective of the system architecture, where a particular application is executed, the main challenge is to reduce the interactions between tasks. The following properties determine the interactions between tasks:

- Data Decomposition and Locality
- Task Dependencies and Load Balancing
- Asynchronous Communication and Latency Hiding
- Collective Communication Patterns

2.3.1 Data Decomposition and Locality

The most effective approach to minimize interaction between tasks is to maximize data locality using appropriate data decomposition and mapping techniques. The number of tasks after domain decomposition should be large enough to occupy available PEs, i.e. to utilize the degree of concurrency. Besides the degree of concurrency, another limiting factor are interactions between tasks due to introduced task dependencies.

As an example, we consider Partial Differential Equation (PDE) solvers, ranging from simple Jacobi iterations to complex multigrid and adaptive mesh refinement methods. These applications are implemented using iterative finite-differences by sweeping over points in a spatial grid. Each point in the grid is updated with weighted contributions from nearest neighbors in time and space. The result represents coefficients of the PDE for a specific point. This computation pattern is called a *stencil* operation, defining a neighborhood relationship between points in the grid [22].

Decomposing the spatial grid into smaller subregions introduces dependencies between adjacent regions. While points within subregions can be processed in parallel on different PEs, points along the boundaries need to be exchanged before continuing with the next sweep (iteration). The communication volume is proportional to the surface of overall domain cuts [80]. Hence, proper decomposition ensures *temporal* locality, i.e. maximizing consecutive references to the same data. *Spatial* locality can be achieved by minimizing the frequency of interactions. Instead of exchanging each point along shared boundaries in a separate message, multiple points should be combined into larger pieces. This reduces latency costs and improves bandwidth utilization.

After domain decomposition, tasks are assigned to PEs, also called mapping [140]. An effective mapping ensures that communication overhead between tasks is minimized. It further maximizes occupancy of available PEs. Revisiting the example of stencil applications, this means that adjacent subregions should be assigned to PEs which are close to each other. Assuming that PEs are interconnected through a network, there should be a minimal number of physical links between two adjacent neighbors.

Obtaining an optimal decomposition and mapping is not always possible for two reasons. The first reason results from highly dynamic scenarios, where interactions between tasks are unknown before runtime. This occurs for example in parallel graph processing [137], n-body simulations [196] or unstructured grids [159]. The second reason are non-uniform computation capabilities of available PEs. Modern HPC systems are equipped with special purpose accelerators, co-located with SMPs on a single chip. Accelerator PEs have a different computational throughput compared to traditional SMP architectures.

2.3.2 Task Dependencies and Load Balancing

Applications with irregular communication patterns are often implemented using dynamic decomposition techniques. A common example is divide-and-conquer, where a problem is recursively decomposed into smaller tasks. Because tasks can spawn new tasks, task dependencies are not explicitly known a priori. However, interactions between tasks can be modeled in a DAG. Nodes in DAG represent tasks (computation) and edges specify task dependencies (communication). The fact that sizes are not guaranteed to be uniform additionally complicates efficient mapping schemes.

Based on the concept of DAGs, HPC researchers have proposed data-centric programming abstractions, where parallelism is implicitly managed in sophisticated runtime systems. These runtime systems aim at effective mapping schemes to balance computational load on available PEs, and minimizing communication overhead at the same time. In contrast to explicit concurrency, parallelism is expressed in terms of input and output dependencies between tasks. Frequently used programming abstractions, utilizing task parallelism in a heterogeneous shared memory model, are OmpSs [54], Kokkos [36] and Thrust [20]. OpenMP, which is still the most-used programming abstraction for shared memory in HPC, has been revised towards a task-centric approach [154, 160]. The availability of RDMA and one-sided communication interfaces has further motivated researchers to abstract task parallelism in distributed memory machines [1, 173].

The crucial question is, if overhead introduced by underlying runtime systems sacrifices performance efficiency. This is potentially important, when communication happens frequently in distributed memory across the network. To preserve scalability, scientists follow a hybrid design. The computational domain is initially decomposed into rather coarse-grained tasks, distributing them on available distributed memory nodes. In a follow-up step, additional tasks are spawned for parallel execution in shared memory.

2.3.3 Asynchronous Communication and Latency Hiding

As described above, resolving task dependencies requires communication overhead, which in turn limits achievable performance. This overhead often results in idle time, when a task waits for a specific event, or if message processing consumes CPU cycles which do not advance computation. A major optimization is to hide these overheads, either by overlapping computation with communication or by overlapping multiple communications. Literature often refers to latency hiding to describe these techniques, although *latency* does not include message processing overhead in CPUs in a strict sense [73].

Obtaining efficient communication-computation overlap needs to be considered in parallel algorithm design. Communication has to be initiated as early as possible, and synchronized as late as possible, which is called *early binding* [51]. While this is feasible in spatially and temporally static communication patterns (e.g. stencil applications), dynamic task dependencies are more challenging. For this reason, task-based programming abstractions often advocate many fine-grained tasks in parallel algorithms, such that a process can work on tasks which are ready for execution, while other tasks are still waiting or blocked. While this seems plausible, many fine-grained tasks also increase associated overhead due to runtime scheduling and task management.

Besides properly designed algorithms, latency hiding requires asynchronous communication mechanism. In shared memory architectures, this is not a severe problem as prefetching, caching and pipelining is efficiently managed in hardware [84]. However, in message passing protocols, this often requires additional software efforts, which Ch. 3 discusses in more detail. These overheads can be substantial in large-scale systems.

Instead of overlapping communication and computation, overlapping multiple communications is equally important. Researchers have proposed various pipelining and tiling techniques to improve asynchrony in communication-intensive problems [45, 94, 106]. We discuss pipelining of asynchronous communication multiple times throughout this work.

2.3.4 Collective Communication Patterns

Communication patterns in scientific applications are often static and regular and often involve a large number of PEs. These patterns have been identified in scientific literature as collective communication patterns, covering most important communication patterns both in shared and distributed memory machines. For example, the data exchange in stencil applications between two adjacent tasks is abstracted with optimized neighborhood collectives in MPI. Even in rather dynamic applications, e.g. n-body particle simulations, PEs repeatedly perform a redistribution of particles among each other. In the worst case, each task needs to communicate with all other tasks, forming a complete graph.

Instead of manually implementing these patterns with simple point-to-point communication operations, collective communication algorithms have been developed to minimize latency and to maximize bandwidth utilization [38, 99]. As an example, MPI specifies an interface for collective communication. Underlying algorithms have been optimized for various network topologies and HPC platforms.

Collective primitives represent a large fraction of communication overhead in scientific applications [133]. Ch. 5 summarizes algorithmic building blocks, which have been proposed over recent years. Based on the presented concepts, we introduce partial aggregation as an efficient optimization technique to increase parallelism and latency hiding in collective communication patterns.

2.4 Summary

Parallel algorithm design has been a field of mostly theoretical interest for a long time. Based on abstract machine models of real world machines, computer scientists formulate lower and upper bounds on asymptotic complexity of algorithms. These abstract machine models provide a trade-off between theoretical simplicity and adequate representation of performance characteristics in real world machines [41]. However, due to increasing hardware complexity, these models become less useful for performance assessment. Scaling parallel algorithms with an increasing core count systems requires a sophisticated co-design approach to combine theoretical analysis with algorithmic engineering methods [23, 169].

Algorithms, achieving practical performance, combine the concepts summarized in this chapter into hybrid programming models. While hybrid programming models compel programmers to manage multiple levels of concurrency, it significantly improves performance. Nevertheless, global communication and synchronization patterns due to a bulk-synchronous lockstep model contribute a large fraction of runtime overhead. In the following section, we discuss message protocols and associated overhead in more detail, which emphasizes the significance of efficient collective communication algorithms. Because at least some communication overhead is unavoidable, we discuss latency hiding techniques in hybrid programming models. While latency hiding techniques are crucial to scale performance, we show what semantic mismatches in collective communication primitives limit applicability of latency hiding.

3 State of the Art and Related Work

Parallel computation requires communication between PEs to resolve data dependencies in algorithms. Because communication does not explicitly advance computation, HPC experts refer to communication *overhead*. Therefore, reducing the impact of communication overhead is crucial to achieve performance in large-scale systems.

The main goal of parallel algorithm design is communication avoidance through maximizing data locality. Parallel programming models, as discussed in Sec. 2.2, support algorithmic analysis to structure parallel executions, however, hide various sources of communication *latency* with constant factors in asymptotic complexities. With the increasing scale of parallel architectures, the impact of these constant factors becomes more prevalent.

This chapter discusses different aspects of latency, which manifest in high-performance parallel systems for two primary reasons:

1. Software overhead in message passing protocols. We explain this based on semantics in the Message Passing Interface (MPI) as the predominant programming model in scientific computing. Because at least a fraction of incurred overheads is unavoidable, latency hiding through asynchronous communication is necessary to occupy available PEs.
2. Interface deficiencies in collective communication primitives which complicate the interaction with multi-threaded parallelism in hybrid programming models. Although hybrid programming models are already common practice, efficient support for multi-threaded collectives remains an open issue.

3.1 Message Passing Interface (MPI)

MPI is an established standard for programming HPC applications in distributed memory machines. The design goals of MPI are performance, portability and efficiency on different HPC systems, without restricting users to a specific communication pattern. Application programmers can choose between various open source implementations, including MPICH [76], Open MPI (OMPI)[65] or MVAPICH[158]. The implementations primarily differ in support for specific hardware characteristics such as Infiniband or iWarp NICs [110]. Commercial implementations from HPC vendors are additionally optimized for architectural features in their platforms [109, 112].

Concurrency in MPI programs is explicit by specifying the number of *MPI processes*. An MPI process abstracts the concept of a communication endpoint and can be an arbitrary unit of execution. In the majority of MPI implementations, an MPI process is an operating system process with a private address space and communication happens through message passing primitives in the distributed memory model [150, 153].

The scope and context of communication is encapsulated in groups and communicators. A group of size p is an ordered set of ranks, uniquely indexed in the range $i \in \{0..p-1\}$ to identify MPI processes in communication primitives. Upon initializing a MPI application, the MPI runtime maps PEs to MPI processes and encapsulates them into a predefined global universe, the `MPI_COMM_WORLD` communicator. Consequently, each MPI process has a predefined rank in `MPI_COMM_WORLD`.

MPI users can manipulate communicators to split, union or intersect distinct groups of ranks. Hence, each MPI process in `MPI_COMM_WORLD` can be a member of multiple user-defined groups, having a different rank with respect to each group. As an example, a common approach in HPC applications is to split the initial `MPI_COMM_WORLD` communicator into separate shared memory communicators. MPI processes which belong to the same shared memory communicator are able to communicate with shared memory semantics, i.e. native read and write access to the shared address space instead of exchanging messages.

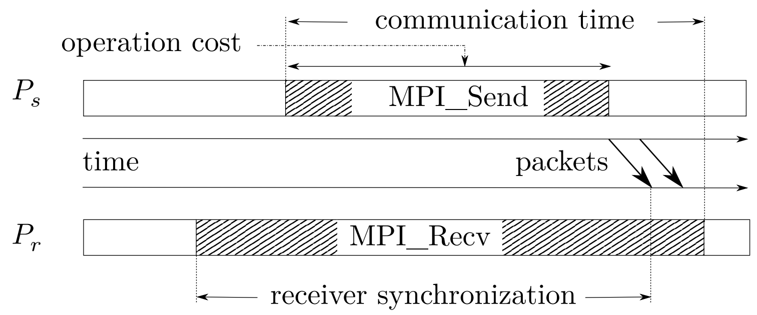
Communication between MPI processes is performed with communication primitives. Below, we describe *point-to-point* and *collective* communication primitives which provide essential concepts for this work.

3.1.1 Point-to-point Communication

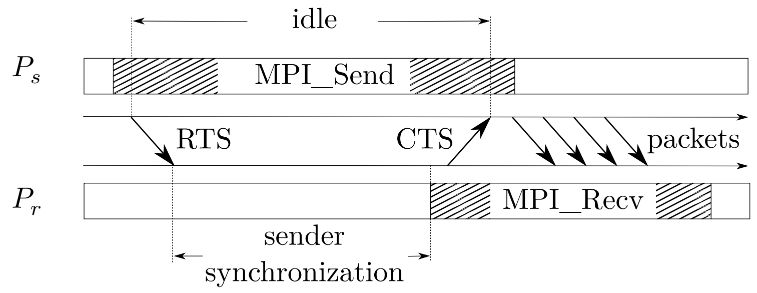
Primitives for point-to-point communication, also called two-sided communication, specify mechanisms for message passing between two MPI ranks in a communicator, i.e. a sender and receiver. A message holds a data buffer and an envelope. The envelope is used on the receiving side for *message matching*, before transmitted data is moved into the receive buffer. Message matching implicitly synchronizes sender and receiver to guarantee in-order message transmission. More specifically, if a sender transmits two messages in succession to one destination, and both match the same receive operation, the respective receive cannot match the second message if the first message is still in transit [150]. Message matching can significantly impact performance, which we discuss more detailed in Sec. 3.2.1.

Point-to-point communication can be either *blocking* or *non-blocking*. Blocking communication guarantees consistent memory buffers of all arguments upon return from the MPI call. Considering the semantics of message matching, this means that in case of blocking sends, the MPI library either literally blocks until message transmission is completed, or allocates auxiliary memory to buffer all arguments, including envelope and data, before returning control to the user.

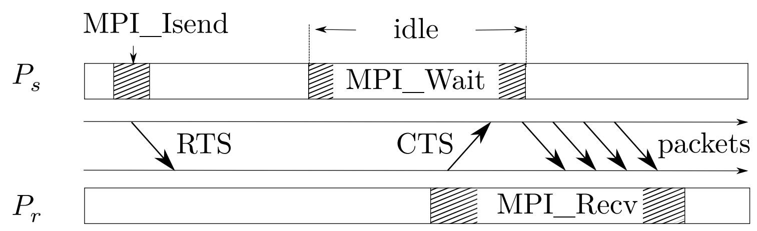
In contrast, *non-blocking* communication relinquishes these guarantees by separating the initiation and completion of a communication operation. After initiating a non-blocking communication there are no consistency guarantees of message buffers. To enforce memory consistency, the respective operation needs to be completed using respective MPI primitives. Therefore, non-blocking communication enables explicit support for asynchronous message transmission. However, MPI implementations are not required to guarantee asynchronous communication.



(a) Short messages: Eager protocol.



(b) Long messages: Rendevouz protocol, blocking send.



(c) Long messages: Rendevouz protocol, non-blocking send.

Figure 3.1: Point-to-point communication protocols for short and long messages.

Point-to-point communications are fundamental primitives, serving as building blocks for synchronization and communication in parallel algorithms. Therefore, researchers have designed performance efficient communication protocols to implement point-to-point primitives. Two prominent examples are [75, 130, 163, 199]:

Eager The sender *immediately* transmits messages to the receiver. The receiving side has to allocate memory buffers in advance to match incoming messages. This protocol is used for small messages and exhibits low startup overhead.

Rendevouz Sender and receiver synchronize through an initial handshake, before the message is actually transmitted. This protocol is commonly used with large message sizes, or if memory consumption is a primary concern.

Fig. 3.1 visualizes three cases of point-to-point communication between a sender P_s and a receiver P_r. Corresponding timelines reside on the upper and lower halves of each diagram, respectively. Time progresses from left to right. In Fig. 3.1a, P_s and P_r do not synchronize before message transmission. Instead, P_s copies the message into a dedicated buffer for *small* message transmission on the receiver side, from which P_r reads transmitted data upon successful message matching. In contrast, Figs. 3.1b and 3.1c represent message transmission in the rendevouz protocol, which requires an initial handshake between sender and receiver through a ready-to-send (RTS)/clear-to-send (CTS) message pair. In Fig. 3.1b, this causes P_s to block in the send routine. When the CTS acknowledgement has been received, P_s starts message transmission. In case of non-blocking rendevouz communication, the send routine immediately returns after issuing the initial RTS message. Later, if P_s requires successful completion, it needs to wait in a waiting routine. In the ideal case, both the CTS acknowledgement and message transmission are already done, causing zero idle time. Otherwise, P_s is idle in the waiting routine. This clearly illustrates the main benefit of non-blocking communication. The time window after issuing the non-blocking send and its subsequent synchronization is available for other work. However, finding a correct timing interval to *perfectly* overlap non-blocking communication and computation is challenging as discussed in Sec. 3.2.4.

The default threshold to switch between eager and rendevouz protocols depends on the respective MPI implementation, and can be explicitly tuned in most cases. To minimize latency overhead in rendevouz protocols, MPI libraries apply different optimization strategies. For example, if the underlying network features RDMA capabilities (e.g. Infiniband), OMPI utilizes CPU bypass mechanisms in the NIC to hide latency. While the sender is waiting for the receiver to acknowledge the handshake, it already *registers* the send buffer, which is a costly operation. Pipelining the handshake and buffer registration reduces latency overhead [199].

Another common optimization for rendevouz protocols aims at fast message transmission in shared memory, if sender and receiver run on PEs in a shared memory node. Most message passing libraries pre-allocate an explicit data exchange zone in a shared memory segment, acting as an intermediate buffer for message transmission. Copying to/from the exchange zone happens in a pipelined fashion to further reduce latency [33]. There exist also *zero-copy* mechanisms through native Direct Memory Access (DMA) support in hardware, which allows a direct copy from send to receive buffers, e.g. KNEM [113]. However, these mechanisms often require special kernel extensions in the operating system. Moreover, while saving a single copy reduces both memory traffic and cache pollution, reports demonstrate performance benefits only for large message sizes [107].

3.1.2 Collective Communication

Point-to-point primitives can express an arbitrarily complex communication pattern in a given communicator. However, performance of global communications depends on different factors, e.g. communicator size, the physical network topology, aggregated communication volume, etc. Even if these parameters are known to a user, designing a system-specific algorithm may not be optimal on other platforms. To simplify optimization of global communication, the MPI standard defines a set of collective primitives. Collective primitives provide an interface for important communication patterns in HPC applications. The scope of collective primitives is defined through a given MPI communicator, i.e. all ranks in the communicator are required to participate in a collective operation. Similar to point-to-point communication, collective primitives are specified in blocking and non-blocking versions.

We classify collective communication primitives into three types:

- Synchronization
- Collective Computation
- Collective Communication

Synchronization primitives are a special case of global communication without any message data. They include only a *barrier* operation to synchronize ranks in a given communicator. A *barrier* is considered complete, if all ranks in the communicator have entered into the operation.

Collective *computation* is realized through *reduction*, where MPI processes perform computations on a distributed dataset, e.g. parallel summations. According to a recent survey, reductions are the most used primitives in scientific applications, accounting for $\approx 60\%$ of the overall time spent in processing MPI operations [39]. Parallel reduction is also an essential operation in machine learning algorithms, which increasingly rely on MPI to scale on distributed memory clusters [21].

Collective *communication* shuffles elements in a distributed dataset in various patterns. We classify the set of collective communication primitives into *dense* and *sparse* collectives. Dense collectives involve all ranks in a given communicator and cover various one-to-many (e.g. broadcast), many-to-one (e.g. gather), and many-to-many (e.g. all-to-all) communication patterns.

In contrast, sparse collectives are optimized for communication patterns, where the involved MPI processes communicate only with a relatively small set of communication peers. Common examples are neighborhood relationships in regular and irregular meshes [22]. Even more challenging are graph traversals, where the set of communication peers is dynamic and unknown at runtime [137]. In fact, as these use cases are ubiquitous in scientific computing, they motivated the standardization of so-called neighborhood collectives in MPI-3 [104, 150]. To express neighborhood relationships, users construct a virtual topology to arrange PEs in cartesian regular or more generalized graph layouts.

Furthermore, collective communications are specified in *regular* and *irregular* flavors. In regular collectives, involved ranks send (receive) a message of the same size. In irregular collectives, each rank sends (receives) a message of different size. The MPI specification reflects this as irregular collectives require to specify the message size for each rank in a vector of size p, as opposed to a single value in regular collectives. In addition to the additional memory overhead, irregular collectives are more difficult to optimize with respect to latency and network congestion. [187].

Algorithms for collective communication primitives are intensely studied and optimized for different network topologies in both shared and distributed memory architectures [38, 82, 99, 103]. There are two approaches to design collective communication algorithms: Hardware-based and unicast-based. Hardware-based collectives exploit capabilities in the NIC to collectively communicate data among all PEs. An example is the Blue Gene/L supercomputer, which supports tree-based broadcast and reductions in the network [66].

On the other hand, unicast-based algorithms are modeled as a series of point-to-point communications among the involved MPI processes. Each stage in a series is a permutation of send-receive pairs. The union of all stages models a complete collective communication. Efficiency is achieved through a minimal number of stages, and each stage is optimized with respect to communication latency and bandwidth utilization. We summarize unicast-based collective communication algorithms in Ch. 4.

Although collective communication can be replaced with simple point-to-point communications, expressiveness, productivity and performance efficiency outweigh hand-coded implementations. Collective operations naturally integrate into common parallel programming models, such as BSP and MapReduce, and further enable hardware support in specific HPC systems. Most important, a standardized interface of collective communications simplifies performance predictability for HPC users and hardware architects, which supports in designing new HPC systems [72].

3.2 Characterizing Latency in Message Passing

Minimizing latency is a major goal of communication optimization. As latency cannot be fully avoided, HPC scientists employ latency hiding through asynchronous communication. To understand the details, we characterize the following common aspects of latency:

- Message matching.
- Operating system interference.
- Multi-threaded resource contention.
- Independent progress.

3.2.1 Message Matching

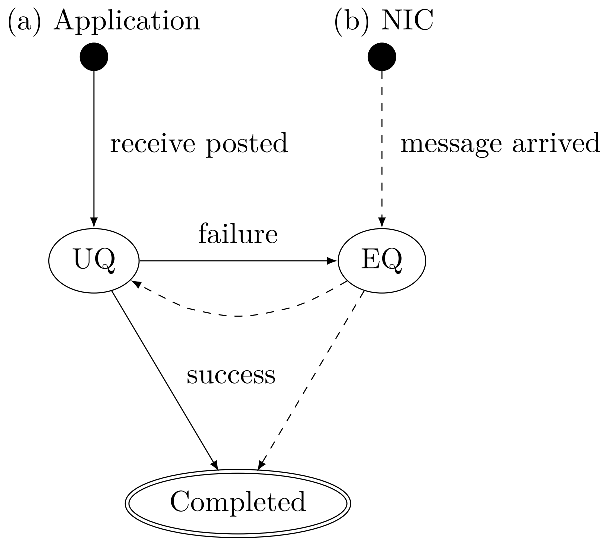
In many cases, it is obviously impossible for a single MPI process to immediately consume all incoming messages. To guarantee serial message matching (cf. Sec. 3.1.1), MPI libraries utilize message queues to handle out-of-sync messages [69, 204]. The usage of message queues is not specific to MPI but is an often used concept to reason about resource usage in message processing. Message queueing operations have crucial impacts on performance and account for up to 60% percent of the overall communication latency [11, 31]. Hence, researchers have proposed various algorithms for message matching to improve communication efficiency [68, 205]. Although these algorithms differ in some details, we describe a basic scheme.

MPI libraries manage two separate queues, one to enqueue expected messages (EQ), and another one to keep unexpected messages (UQ). We visualize both queues in Fig. 3.2. Each queue deals with a specific case:

(a) The application issues a receive operation. The MPI runtime will first scan the UQ for incoming messages from the network prior to the respective receive operation. If matching succeeds, the receive operation is complete. Otherwise, the receive operation is added to the EQ to match a future incoming message from the network.

(b) The MPI runtime is observing an incoming message from the network via the NIC. It begins by scanning the EQ for a matching receive. If this matching succeeds, the request is complete. Otherwise, the send operation is added to the UQ, until the application will issue a matching receive.

Resource usage for message queues is expected to grow in HPC applications. Firstly, because of exponentially increasing data volumes in scientific applications. Secondly, due to a rising number of PEs in HPC systems. To quantify the performance impact of message matching, prior work conducted case studies on real-world applications. Experiments reveal that buffering unexpected send operations degrades performance due to costly data copying on the receiver side. Keeping this in mind, it is natural that performance is proportional to the number of unexpected messages, i.e. the size of the unexpected queue (UQ) [30, 58].



EQ: Expected Queue   UQ: Unexpected Queue

Figure 3.2: A simple message matching scheme.

3.2.2 Operating System Interference

Operating system (OS) interference, often referred as noise or jitter, includes asynchronous interrupts of application codes by the system software. These interrupts occur for various reasons, such as periodic timers, special hardware events, or OS scheduler interventions to replace the currently running thread with a different thread. Understanding the impact of noise is an important research topic. This is not only true for tuning operating and I/O systems but also for overall system design [57]. Previous works assume that the impact of noise generally grows with scale, sometimes even linear with system size [152, 189, 194].

System balance, i.e. the ratio of network bandwidth to computational performance, has additional impact. In particular, applications exhibiting a high arithmetic intensity are sensitive to OS noise [162]. However, in communication-bound applications, we assume that the effect of system balance is rather negligible compared to load imbalance and irregular memory access patterns. In our work, we emphasize that noise includes only factors outside the application influences. As an example, cache misses due to inefficient memory accesses do not classify as noise, while unintended context switches in the OS scheduler do.

Of particular interest is the effect of noise on communication operations. Many HPC codes follow on a bulk-synchronous design with intermediate locksteps, where participating processes coordinate their progress with collective operations. Because all processes must participate in collective communications, delays of one or a few processes can propagate to all processes across the system. Hence, maintaining synchronity between processes is important. In the ideal case, computations between two locksteps take the same time on all processes [17].

A theoretical analysis of collective communication algorithm suggests that, depending on the distribution, noise can in the worse case (exponential distribution) scale linear with the number of processes [3]. Although we think the assumptions in the underlying model only partially reflects real-world use cases. A common misunderstanding in the context of the MPI standard is that collective operations are considered as *synchronizing* operations [189]. However, according to the MPI-3.1 standard, Sec. 5.1 [150]:

> *[...] a collective communication operation may, or may not, have the effect of synchronizing all calling processes.*

Such simplified assumptions result in imprecise understanding for performance tuners. Hoefler et al. consider the specified rule in their LoGOPS-model to obtain a more realistic model depending on the algorithm and network parameters [101]. A detailed analysis of synchronization overheads in point-to-point communications suggests that, although blocking communications can absorb OS noise in some cases, non-blocking communication is more robust in general. To better understand the impact on collective communications, a simulation on up to a million MPI processes was conducted for different HPC platforms. Results are in line with previous studies, revealing that maximum slowdown of large-scale collectives scales linearly with the injected noise. In case of a comparable low number of MPI processes (≤ 512), all systems were able to eliminate negative noise impacts. However, after increasing the number of MPI processes beyond a system-specific threshold, OS noise substantially reduced performance scalability. This is particular true for small message sizes in collective communication patterns. With larger message sizes, the noise impact is negligible compared to message transmission overhead. Benchmarks with real-world applications confirm these results in that collective communications are particularly sensitive to noise.

Solutions to improve robustness against noise are widely discussed in literature:

- In tree-based networks, which is common in HPC systems, it is beneficial to place noisy nodes near the topology's root to minimize impact on frequent communication paths, which happens close to the leaves in most cases [57].
- Another solution is co-scheduling, where the operating system considers application-specific semantics to minimize noise impact due to system tasks. An example are collective communications which require all processes to participate in communication at the same time, e.g. *all-reduce* or *all-to-all*. Whenever such an operation occurs, the scheduler prioritizes all application tasks over system tasks to maximize communication progress [114].However, many applications exhibit non-uniform computation times and sparse communication patterns which complicates parallel awareness for the operating system.

A recent paper studies the impact of the Linux scheduler, which is the most used OS on current HPC systems [136]. Conflicting scheduling decisions in the Linux kernel results in

Table 3.1: MPI thread safety levels.

Thread Level	#Threads	Concurrency	Synchronization
Single	1	—	—
Funneled	N	1 (Master)	Application
Serialized	N	1	Application
Multiple	N	N	Library

significant performance loss. For example, many scheduling algorithms minimize contention for shared caches among threads, while others focus on power management and temperature. Kernel developers have changed their strategy towards simpler scheduling policies and relying more on efficient parallelism in applications. For the same reasons, researchers have proposed lightweight micro-kernels with more effective interfaces for data-intensive workloads [197]. However, given the dominance of Linux on HPC systems, it remains unclear whether and how micro-kernels are integrated into future HPC system software.

Most studies mentioned above confirm that the most effective strategy to absorb unpredictable noise is asynchronous communication. Applications need to overlap both intra- and inter-node communication with computation tasks. In particular with large-scale collectives where progress depends on all involved MPI processes.

3.2.3 Multi-threaded Resource Contention

The rise of hybrid programming models lead to a shift towards thread-level runtimes with fast context switches in user-space (cf. Sec. 3.4). Although MPI supports thread-level parallelism, obtaining performance efficiency is still difficult [12]. Before elaborating the main issues, we summarize supported levels of thread safety [150] in Tbl. 3.1.

The first columns lists specified thread safety levels in increasing order, with *single* as the lowest and *multiple* as the highest requirement, respectively. The second column indicates whether applications may have multiple threads of execution in associated levels. The third column indicates whether multiple threads may concurrently issue MPI calls. The last column indicates responsibility to synchronize concurrent executions, either the application or the library itself.

Thread-level *single* indicates that each MPI process has only one thread of execution, i.e. the application is single-threaded and does not require any thread safety guarantees. Thread-level *funneled* implies a possibly multi-threaded application, however, only the master thread is able to issue any MPI calls. The *serialized* thread-level slightly relaxes this guarantee in that any thread may issue MPI calls, if the application ensures a serialized execution. Although the thread-levels *funneled* and *serialized* are semantically different, at least the open source MPI libraries do not distinguish between them technically [65, 76, 158]. The *multiple* level is the most demanding for MPI libraries as concurrent accesses to inherently serial resources need to be guarded for potential data races. For example, not all network drivers provide thread-safe interfaces, which are intended to achieve better performance [110]. While researchers have proposed various techniques to improve thread support, MPI libraries still rely on coarse-grained critical sections (mutexes) at the time of writing this thesis. These mutexes funnel parallel into serial executions, i.e. only one thread of execution is allowed at a time. Recent evaluations report an up to four-fold reduction in overall message throughput, comparing MPI applications with thread level *multiple* to a single-threaded pure MPI execution [7]. Performance degradation is particularly significant in case of many small messages and continuously improves with larger messages.

Another problem with multi-threaded contention is fairness across multiple threads in NUMA architectures, which are common in HPC systems. Due to NUMA effects,memory access costs depend on the physical location of virtual addresses. If multiple threads are competing for mutex ownership, OS schedulers prioritize threads running close to the previous owner. This minimizes latency during ownership passing, however, penalizes other threads which cannot progress pending communication calls. This issue has been observed in several studies [7, 12] and causes severe problems in particular with collective communications (e.g. barriers), if progress depends on participation of all MPI processes. Proposals to mitigate these issues can be summarized into three approaches:

- Contention reduction,
- contention avoidance, and
- more efficient contention management.

Contention reduction relies on fine-grained locks and atomic operations, instead of heavy-weight mutexes [53, 77, 90]. Although these studies demonstrate reduced contention, they cannot completely avoid it. Similar to mutexes, lock acquisitions may exhibit non-deterministic delays, e.g. due to inter-socket latency, which degrades asynchronous benefits of non-blocking MPI calls.

A more advanced approach focuses on *contention avoidance* along the application's critical path. Based on an offload model (cf. Sec. 3.2.4), processing MPI communication requests is delegated to communication threads [56, 132, 191]. Although this model is simple, it comes with disadvantages. Communication threads compete for resources with application threads, especially if the number of threads exceeds the number of available PEs, which is called thread oversubscription. Furthermore, the offload mechanism requires additional memory buffers for bookkeeping pending and completed requests.

The third approach aims at more efficient *contention management* [5, 7, 46]. Instead of allocating dedicated communication threads, application threads collaborate through token passing to achieve fair progress among all threads. Adaptive load balancing strategies prioritize threads issuing new communication operations over threads, which may block due to polling or waiting for pending operations. An important metric for efficiency is the number of *dangling* requests. In MPI, issuing a non-blocking communication returns a request handle to enforce completion through polling or waiting. A dangling request is a request handle whose associated communication call is already complete, however, has not been explicitly freed by the issuer. To saturate the network, threads need to detect and replace completed requests with new requests as early as possible.

Given the huge amount of applications relying on MPI, it is difficult to integrate these strategies into mainstream MPI libraries. A vague specification in the MPI standard how to interoperate with multi-threaded libraries in shared memory complicates advanced synchronization techniques and load balancing between threads [7].

3.2.4 Independent Progress

The MPI standard mandates a progress rule on non-blocking peer communications. However, as of today not all MPI libraries implement the same strategy. The most imperative approach for users is that MPI libraries progress outstanding (non-blocking) communication calls *independent* of the application. The other extreme is that MPI libraries require the user to issue periodic MPI calls to progress non-blocking communication.

Below, we summarize most common techniques to implement independent progress:

- Hardware-based progression.
- Manual progression.
- Threaded progression.

3.2.4.1 Hardware-Based Progression

Achieving independent progress depends not only on MPI libraries, but on available hardware. High performance networks, e.g. Infiniband, feature RDMA, allowing direct memory access from a host to another host *without* involvement of the network software stack and CPU. Furthermore, it does not affect caches on the remote host, i.e. accessed memory contents are not loaded into CPU caches. Suppose a MPI process posts a receive request, before the send message arrives from the network. When the send request eventually arrives, all required information for message matching is available in the *expected* queue (cf. Fig. 3.2). With RDMA, communication can be completed without any involvement of the application or host CPU [164]. However, if the MPI library requires periodic MPI calls, completion depends on the correct timing. Prior work provides an excellent overview on how independent progress depends on MPI implementation, hardware features and message sizes [31].

Another related capability in this context is offloaded message processing and matching using external co-processors on the NIC [120, 190]. These approaches bypass the host processor, which can continue with compute tasks in the application. However, these interfaces are still blocking, which disables any overlap. Furthermore, these proposals only focus on simple point-to-point messages and do not consider collective communications.

A recent paper presents sPIN, extending the idea of hardware-assisted message processing [92]. Relying on the fact that a single message transmission involves multiple packets in the network, sPIN exposes an interface for in-network packet processing. Applications register specific header, payload and completion handlers which are executed in hardware and operate on dedicated memory buffers to prevent interferences with CPU caches. This proposal provides various possibilities for communication-computation overlap, however, requires expertise to integrate into applications with more complex communication patterns.

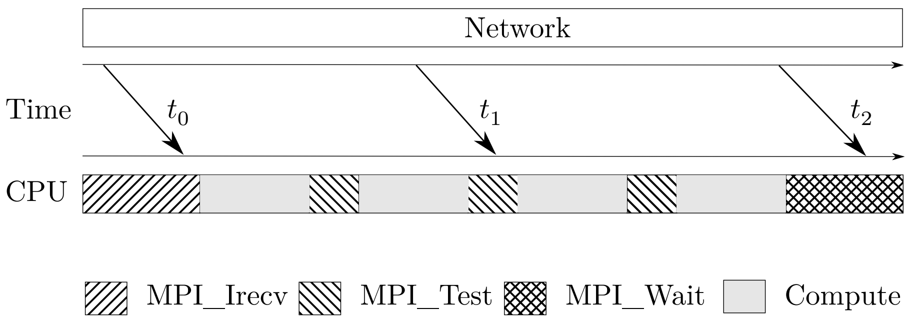


Figure 3.3: Single lockstep in scientific applications with non-blocking polls. Iterative solvers repeatedly execute this lockstep many times until the algorithm terminates.

3.2.4.2 Manual Progression

If applications need asynchronous progress, without knowledge about these specific hardware features, periodic polling in software is still the most reliable approach. The MPI standard defines two primitives to poll on outstanding communication requests:

- `MPI_Wait`: Blocking poll until outstanding requests have been completed.
- `MPI_Test`: Non-blocking poll of an outstanding request.

In iterative solvers, programmers often combine both primitives to *manually* progress non-blocking communications [6]. Fig. 3.3 shows a common pattern in iterative solvers, which is executed by all MPI processes. Time of a single iteration progresses from left to right, and each processor performs three steps. In the first step, all MPI processes initiate non-blocking communication requests, a `MPI_Irecv` in this case. In the second step, each MPI process independently performs compute tasks (grey colored boxes). In the third step, after computation has finished, MPI processes complete communication requests in a blocking `MPI_Wait` to ensure consistent message buffers. These steps are repeated in subsequent iterations, until the application terminates. Additionally, the computation phase is interleaved with non-blocking polls (`MPI_Test`) to progress pending communication requests. However, recent work has shown that finding an optimal polling interval is difficult even with regular communication patterns and fixed message sizes. The opaqueness of MPI requests does not expose relevant details about the underlying communication mechanism in the network. In point-to-point communication, if the message exchange follows a rendevouz protocol, users have to account for an additional handshake. And if a message is split into multiple packets, each packet possibly requires a separate network poll. Finally, the overhead of a non-blocking poll can take up to a few milliseconds [97]. If there is no message arrival to process, either because polling is too late (t_2 in Fig. 3.3) or too early (t_1), performance degrades due to wasted CPU cycles. Recent work has shown that an incorrect timing makes the situation even worse due to unnecessary cache pollution [163].

3.2.4.3 Threaded Progression

A common solution to mitigate the issues with manual progression is threaded progress, where each MPI process spawns an additional communication thread. The communication thread continuously polls the network to process message arrivals immediately. Although this mechanism causes additional overhead, there are multiple reasons for adopting this approach:

- The rising number of PEs in a single node provides more compute resources for *operational* tasks, e.g. message passing progress.
- Because many HPC applications are rather communication-bound, they cannot continuously occupy all cores [6]. Hence, utilizing idle cores for background tasks is reasonable.

Although it is an unspecified feature, many recent MPI implementations provide an explicit flag to enable asynchronous progress. The effect is that each MPI process spawns an additional communication thread. However, this feature requires a multi-threaded MPI library and performance is often poor due to multi-threaded contention (cf. Sec. 3.2.3). These impacts particularly apply in *pure* MPI programs, if each process maps to a single PE per node. Spawning an additional thread doubles the number of execution units, causing significant cache and memory traffic [48, 96].

Reserving one thread for asynchronous progress is more reasonable in hybrid MPI programs, where a single MPI process maps to multiple cores. Previous works have adopted this idea in several variants. A simple approach is to put a thread into a *blocking* receive, which never completes for the entire application runtime. The effect is that this thread continuously polls the network for arriving messages in MPI. However, it requires the *multiple* thread-level, which offers poor performance in most cases [96]. A more promising idea is to initialize MPI in *funneled* or *serialized* modes and offloading all communication operations on a thread-safe message queue in user-space. The progress thread processes communication operations in FIFO-order from this queue to satisfy serial message matching guarantees in MPI with respect to application threads [98, 191]. If queuing operations are efficiently implemented, e.g. with *lockfree* techniques, reduced communication overhead significantly improves performance. However, funneling communications to a single thread may lead to limited bandwidth utilization, especially with large messages.

A problem with threaded progress in general is that progress threads compete with application threads for CPU resources. Depending on the arithmetic intensity of the application and the number of progress, it can result in degraded performance due to increased cache and memory contention. Instead of progress threads, other approaches rely on so-called *ghost* processes, which are isolated from the application. Application and ghost processes communicate via shared memory segments. In contrast to progress threads, this enables multiple MPI processes to share a single progress thread. Similar to progress threads, it can achieve significant performance benefits. However, it adds non-negligible complexity in message matching for two-sided communication [178, 179].

In summary, progressing non-blocking communication is challenging in MPI due to a vague specification in the MPI standard. At the same time, however, efficient mechanisms become more relevant in MPI applications to minimize communication overhead and to maximize overlap potential with increasing node-level parallelism.

3.3 Assessing Latency Hiding Potential

Above we have identified different sources of latency overhead, which results either from requirements for message processing or additional runtimes costs inherent to distributed systems. Thus, hiding latency through communication-computation overlap is crucial. Maximizing overlap potential requires co-design of parallel algorithms and efficient communication layers. While there is no general advice to obtain efficient communication, scientific literature suggest two essential pillars [51, 98]. The first is *early binding*, which means to initiate communication as early as possible. The second is *late synchronization* to provide runtime systems enough time for communication tasks.

Before discussing established techniques to obtain communication-computation overlap, we introduce a semi-formal communication model to quantify latency and bandwidth.

3.3.1 LogP communication model

Communication cost models play an important role in algorithm design. They assist in runtime prediction and performance analysis, without considering architecture-specific details. A wisely chosen set of parameters is relatively small without loosing applicability in real-world applications.

This section summarizes established communication cost models in scientific computing.

3.3.1.1 Hockney Model

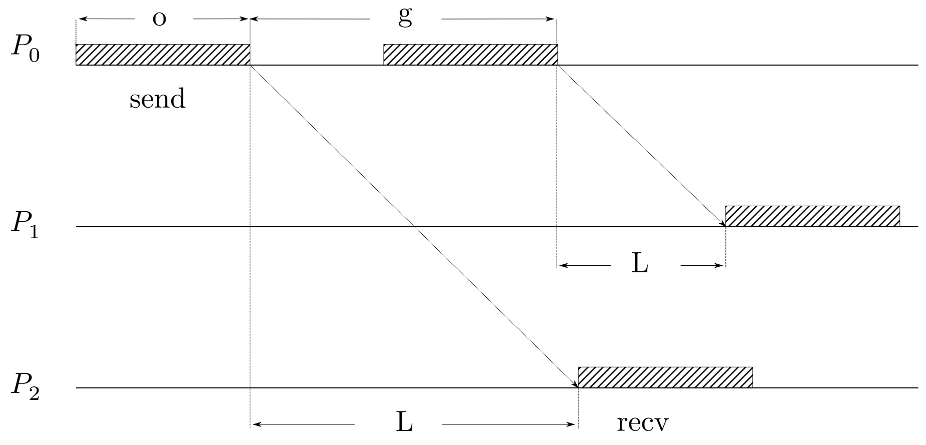
Eqn. (2.1) introduced the Hockney model to quantify communication costs. Its simplicity provides an intuitive understanding about communication complexity and it has been adopted in several projects to assess performance of collective algorithms [73, 167, 184]. However, the simplicity can lead to imprecise performance assumptions due to variable contributions from processors and the network [165]. For example, the Hockney model cannot quantify the overlap potential in case of multiple pipelined message transmission. Another limitation is that network congestion cannot be accurately modeled [38].

3.3.1.2 LogP Model Family

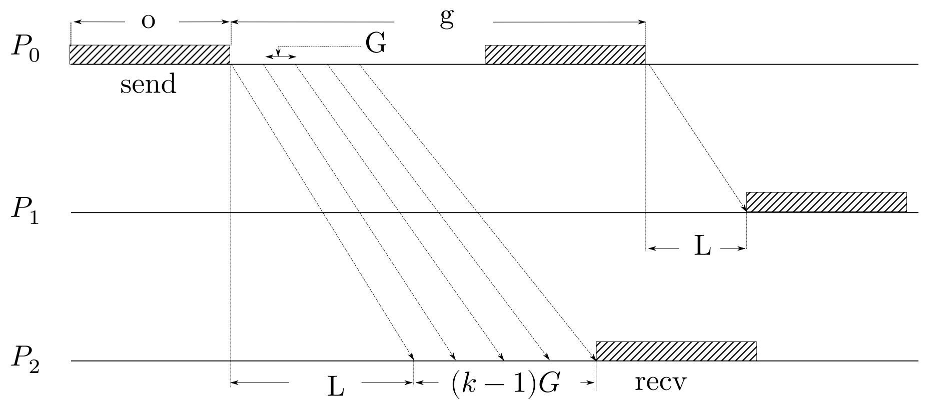
In contrast to the Hockney model, the LogP model and its extensions have demonstrated reliable predictions for modern network interconnects [42]. The original LogP model defines the time T of a single point-to-point communication as follows:

$$T = L + 2o \tag{3.1}$$

In this equation, L is the transport latency for a single byte across the network. Additional software overhead for message handling in the communication endpoints, e.g. message matching or buffer copies, are modeled in variable o.



(a) LogP model.



(b) LogGP model.

Figure 3.4: LogP and LogGP models.

Another assumption is that larger messages are split into smaller ones of size ω, the machine word size. A processor is allowed to issue subsequent messages only every g cycles. Fig. 3.4a visualizes this model with 3 processors, where P_0 sends two small (i.e., $s \leq \omega$) messages to P_1 and P_2, respectively. Time progresses from left to right and is measured in CPU cycles. The send cost for P_0 amounts to o cycles. After a message transmission of L cycles, the receiver pays another o cycles to handle message transmission.

For simplicity, we assume that $w = 1$. Transmission costs for m messages, each of size s bytes, are estimated as follows:

$$T = L + 2o + (\max\{g, o\})(m-1)s \tag{3.2}$$

Consequently, the available bandwidth is limited to $\lfloor L/g \rfloor$ simultaneous message transmissions between two endpoints. This model further assumes a congestion free network, i.e. the network has a capacity to carry at least $p^2 \times L/g$ messages.

An extension to this model, to better capture large message transmissions, has been introduced in the LogGP model [4], which adopts the same notation as the LogP model.

Table 3.2: LogGP parameters.

L	Maximum latency between two endpoints.
o	CPU overhead for a single message.
g	minimum delay between two messages ($1/g \equiv$ message rate).
G	Gap per byte ($1/G \equiv$ bandwidth).
s	Message length in bytes.
p	Number of processors (ranks).

However, instead of splitting large messages, G charges an overhead per byte during message transmission, as illustrated in Fig. 3.4b.

Transmission costs for m large messages, each of size s, are estimated in the LogGP model as follows:

$$T = L + 2o + m(s-1)G + (m-1)g \tag{3.3}$$

Hence, this model distinguishes between two forms of bandwidth: $1/G$ for long messages and $1/g$ for short messages. This approach has demonstrated reliable prediction accuracy in real world machines as it can capture two important optimizations:

- Coalescing multiple small messages to the same destination into fewer ones to take advantage of fast long message transmission.
- Pipelining concurrent message transfers to hide these overheads [123, 183]. The sending and receiving processors are busy only for o cycles. The remaining time, in particular L and g, is available for communication-computation overlap.

Tbl. 3.2 lists all parameters for reference in subsequent sections. Further extensions to this model additionally consider synchronization overhead between sender and receiver [102, 111], depending on the underlying communication protocol (cf. Sec. 3.1.1), and network congestion [148]. Although these approaches further increase accuracy, this comes at a cost of higher complexity in the overall model and we do not consider them in this work.

The LogP model family has been adopted in many research projects to design new parallel algorithms, performance prediction of existing algorithms or prove an algorithm's optimality in distributed memory machines [55, 108, 119]. However, adopting the LogGP model for reliable performance predictions requires careful parameter assessment of a particular machine. Incorrect parameters easily lead to wrong assumptions about performance scalability and bottlenecks. Proper experimental strategies to avoid these pitfalls have been studied [18, 43, 95], which we consider in this thesis for the evaluation in Ch. 7.

3.3.2 Example: Fast Fourier Transform

In this section we review techniques to maximize overlap in HPC applications, based on pipelining theory [34, 45]. These techniques are particularly useful in iterative solvers with regular communication patterns, which represent an important subset of scientific applications. We further assume that all communication is non-blocking, unless stated otherwise. Pipelining techniques have been extensively studied in microprocessor architectures to achieve instruction-level parallelism [84]. HPC scientists have adopted these techniques to improve concurrency in message passing algorithms, which we classify into two categories, called *tiling* and *sliding windows* [45].

We illustrate these techniques with a Fast Fourier Transform (FFT) algorithm, which is an intensely studied algorithm in scientific computing. A well-known implementation is the Cooley-Tukey, radix-2 algorithm with decimation in frequency for computing an n-point FFT [40, 166]. This algorithm has an asymptotic computational complexity of $\mathcal{O}(n \log n)$. On average, computing a single FFT complex value takes 2 floating point multiplications and 3 floating point additions. Combining all floating point operations into a single term gives a serial running time of:

$$T_s = t_w n \log n \tag{3.4}$$

In this equation, t_w is the time to compute one complex value of the FFT algorithm. The parallel algorithm linearly partitions the input vectors of size n elements to p PEs. Given both n and p are a power of 2 (i.e. $n = 2^d$ and $p = 2^j$), each PE owns a contiguous block of size $b = n/p$.

The algorithm operates in two phases. During the first $\log p$ iterations, computation includes values residing on other MPI processes, i.e. elements need to be communicated in a ALL-TO-ALL exchange. The second phase operates only on local values. Suppose the communication time in cycles for a vector of size b is approximated with function $T_c(b)$, the overall running time for p PEs can be approximated as follows:

$$T_p = \log p T_c(b) + \log n (t_w b) \tag{3.5}$$

The first term approximates communication, the second term computation time, respectively. Eqn. (3.5) does not consider any overlap of communication and computation. Assuming that communication and computation can be overlapped, we split up computation in both phases, yielding the following term:

$$\begin{aligned} T_p &= \log p T_c(b) + \log p t_w b + (\log n - \log p) t_w b \qquad (3.6) \\ &= \log p (T_c(b) + t_w b) + (\log n - \log p) t_w b \end{aligned}$$

For the remainder of this section we ignore the second phase in Eqn. (3.6), where computation operates only on local values, and focus only on first $\log p$ rounds (global phase). A simple implementation of a single round is given in Alg. 1. We assume a computation of b mono-dimensional FFTs on vectors of size m, i.e. $b = m \times m$. It is further assumed that n elements are already evenly partitioned among p PEs.

Algorithm 1: 2D FFT global communication phase, without overlap.

```
for $x \leftarrow 0$ to $m$ do
    1d_fft($x$) // x dimension
end
Alltoall($b$) // transpose blocks from x to y
for $y \leftarrow 0$ to $m$ do
    1d_fft($y$) // y dimension
end
```

In Alg. 1, each processor computes first on m local elements along the first dimension. After a global communication phase, another computation step follows to operate on m values along the second dimension. Because communication is blocking, there is no overlap which corresponds to the expected runtime in Eqn. (3.5).

Switching from blocking to non-blocking communication eliminates the serial bottleneck but requires more sophisticated strategies to obtain performance efficiency. We discuss two common pipelining techniques below:

- Tiling
- Sliding windows

3.3.2.1 Tiling

The all-to-all communication pattern is considered as the least scalable one. A first step to hide communication cost is to utilize available parallelism in the FFT algorithm itself. Computing a FFT on a vector of size m does not require complete transmission of this vector, before starting computation. Instead, we split each message of size m into tiles of size k. This divides computation of problem size m into M independent blocks of size m/k. Arranging these blocks in a pipeline with non-blocking communication allows to overlap message transmission and computation. Alg. 2 shows a modified version of Alg. 1, after incorporating the tiling technique.

Algorithm 2: 2D FFT global communication phase, after tiling.

```
for $i \leftarrow 0$ to $m/k$ do
    for $x \leftarrow ik$ to $(i+1)k$ do
        1d_fft($x$) // x dimension
    end
    Ialltoall($i$, $k$) // transpose block $i$ of size $k$
    if $i > 0$ then
        Wait($i-1$)
    end
end
Wait($m/k$) // last block
for $y \leftarrow 0$ to $m$ do
    1d_fft($y$) // y dimension
end
```

Each iteration $i \in \{0..M\}$ is processed in three steps:

1. Compute block i.
2. Transpose block i using non-blocking ALL-TO-ALL.
3. Synchronize communication buffers of block $i-1$.

Note that there is a cost for initializing and finalizing the pipeline, which leads to a drain time for the last block. However, the first and second step of the remaining blocks are available for overlap along the x-dimension. Taking this into account we approximate the overall running time with the following term:

$$t_w k + T_c(k) + (M-1)\ \max\{t_w k, T_c(k)\}. \tag{3.7}$$

We clarify that Eqn. (3.7) provides a lower bound, as T_c and t_w cannot be perfectly overlapped. As discussed in Sec. 3.3.1, a fraction of communication time requires some CPU cycles to perform message matching, buffer copies, etc. During this time, no overlap is possible, which we have to consider in a more precise model. Furthermore, the communication time $T_c(k)$ depends on the underlying ALL-TO-ALL algorithm implemented in the communication runtime. For example, if we assume that involved PEs are interconnected through a hypercube topology, the communication time $T_c(k)$ can be approximated as follows:

$$T_c^{hyp}(k) = \log_2 p(L + 2o + \frac{1}{2}mkG - \mathcal{O}(k)) \tag{3.8}$$

After conducting a careful parameter assessment of LogP parameters for a particular system, we can use linear calculus to optimize the tiling factor k. A more detailed overview about tiling techniques for the discussed FFT algorithm is given in prior work [35].

3.3.2.2 Sliding Windows

Another established approach are sliding window techniques, allowing multiple concurrent non-blocking communications. Instead of a synchronous wait for block $i-1$ in iteration i, a sliding window manages w outstanding communication requests in flight. Iteration i synchronously waits for block $i-w$, giving each outstanding request more time to finish. A well chosen parameter w utilizes available injection bandwidth to further improve overlap potential. The following algorithm extends Alg. 2 with a sliding window technique.

The expected communication in Eqn. (3.7) still holds. However, the final draining step is more expensive on average, as w outstanding communication requests need to complete. Tiling and sliding window techniques introduce a trade-off between overlap and memory requirements. A high tiling factor k limits speedup due to a more expensive initialization and finalization of the pipeline. Hence, the tiling factor k needs to be large enough to maximize communication-computation overlap without causing too high draining costs. Similar considerations apply to the sliding window size. A small window may not fully utilize available injection bandwidth, while increasing w requires more memory for communication

Algorithm 3: 2D FFT global communication phase, after tiling.

```
for $i \leftarrow 0$ to $m/k$ do
    for $x \leftarrow ik$ to $(i+1)k$ do
        1d_fft($x$) // x dimension
    end
    Ialltoall($i$, $k$) // transpose block $i$ of size $k$
    if $i > w$ then
        Wait($i - w$)
    end
end
Waitall($(m/k) - w$, $w$) // last $w$ blocks
for $y \leftarrow 0$ to $m$ do
    1d_fft($y$) // y dimension
end
```

buffers in outstanding requests. Both parameters need to be carefully tuned for a particular platform and require a detailed understanding about computation and communication costs.

3.3.3 Remarks

The case study with a FFT algorithm demonstrates applicability of simple cost models to assess potential for communication-computation overlap. Balancing communication and computation costs is crucial to achieve effective latency hiding. However, although manual transformation techniques are ubiquitous in HPC codes, they require a detailed understanding about application bottlenecks and a particular platform to tune tiling factors and pipelining sizes. This is feasible in dense iterative solvers, where communication and computation are regular. For common kernels, e.g. FFT, highly optimized libraries even abstract this complexity with an easily accessible interface [61].

However, if communication is sparse, irregular, or exhibits varying message sizes, manual transformation techniques are difficult to accomplish. Non-uniform communication and computation costs often lead to idle PEs in a bulk-synchronous programming model. To obtain more efficient load balancing strategies, scientific applications employ a hybrid design of distributed and shared memory programming models. The following section summarizes challenges in hybrid programming abstractions, and recent research efforts to improve present issues.

3.4 Hybrid Programming Abstractions

Since the manifestation of multi-core architectures, HPC applications commonly follow a hybrid programming model. In shared memory, threads interact through native loads and stores. On the other hand, message passing is used to handle interactions in distributed memory through the network.

In contrast to message passing, thread-level parallelism provides a more fine-granular programming model and better interacts with coherent shared memory semantics, where data is shared by default. The trade-off is to hierarchically organize parallelism into multiple layers of distributed and shared memory programming models. Advantages of hybrid programming abstractions have been studied in several papers and can be summarized as follows [168]:

Load imbalance Applications with non-uniform data distributions often require dynamic load balancing mechanism to minimize contention and bandwidth bottlenecks [83]. A shared memory abstractions simplifies implementation of these strategies and can significantly reduce message passing overhead within a node.

Memory consumption Hybrid message passing programs perform multi-dimensional data decomposition to balance work load among multiple levels of parallelism. The more domains with private address spaces (i.e. message passing layers), the higher the aggregated communication surface and memory requirements for halo exchanges. This particularly applies to regular grids where halo boundaries are communicated between neighboring domains. Overhead due to internal data structures and replications for efficiency reasons in runtime libraries needs to be additionally considered.

Algorithmic Improvements Beside reduced memory overhead and a simpler programming interface, a hybrid model can utilizes different algorithms on different levels. For example, in conjugate gradient solvers, different preconditioners are implemented for shared and distributed memory, respectively [159]. Other optimizations include dedicated cores to achieve independent progress with non-blocking communication or I/O (cf. Sec. 3.2.4).

There is no general rule to maximize aforementioned benefits. Whether a hybrid programming model is faster than a pure MPI approach depends on application characteristics and the applied domain decomposition. Representative benchmarks are necessary to determine performance gains for a particular use case. However, a properly designed hybrid of message passing and shared memory parallelism can result in a notable performance speedup [6, 143, 177].

Recent studies on the usage of MPI in supercomputers confirm that hybrid codes represent the largest fraction among scientific applications [24, 133]. Over the last few years, accelerators are gaining increasing popularity in scientific application codes, with CUDA and OpenCL as the driving forces. HPC experts expect that combinations of different frameworks, e.g. MPI+OpenMP+CUDA, will be common in HPC applications. Hence, effective interactions between these programming models are crucial to maximize performance benefits. Below, we discuss two challenges:

- Multi-threaded communication models.
- Parallelism in point-to-point primitives.
- Deficiencies in collective primitives.

3.4.1 Multi-threaded Communication Models

HPC developers follow either of two implementation patterns in a multi-threaded message passing environment:

Single-sender Communication of all spawned threads in a MPI process is funneled into a single master thread, which handles MPI communication and synchronization with other processes.

Many-sender Each thread independently issues MPI calls to communicate its pieces to threads running in other MPI processes.

In the *single-sender* pattern, all tasks within a parallel region first synchronize in a thread barrier, before the master thread can initiate communication. The effect is that communication cannot start before the last thread enters the synchronization point. Additionally, if subsequent computation steps depend on remote data, e.g. a neighborhood exchange, idle synchronization time expands until the master thread has completed pending communication requests. This designs violates the *early binding* rule (cf. Sec. 3.3). Furthermore, it complicates overlap strategies, as idle synchronization time during message processing blocks all but the master thread.

In the *many-sender* pattern, each thread independently issues MPI calls to communicate its pieces. This reduces idle synchronization time of the single-sender pattern, as threads initiate communication when data becomes ready. However, it causes additional overhead due to multi-threaded synchronization in MPI libraries. Since multi-threaded support is still not optimal in state-of-the-art libraries (cf. Sec. 3.2.3), synchronization overhead multiplies with the number of threads. It is even more complicated on the receiving side. In addition to thread synchronization, multiple messages need to be successfully matched. From previous studies, we know that message matching overhead scales linearly with the queue length (cf. Sec. 3.2.1).

To leverage massive node-level parallelism with fast context switches, applications require more efficient point-to-point and collective communication interfaces. Below, we review recent research efforts to address these issues.

3.4.2 Parallelism in Point-to-Point Primitives

To overcome the *private copy* limitation in shared memory, HMPI introduces the concept of ownership passing. Instead of copying data from send to receive buffers, the ownership of a buffer is transferred from the source to the destination process [60]. Although this approach significantly improves performance with intra-node communication, it does not address the single- or multi-sender problems, if data is moved across the network.

A related capability are shared memory *windows* which have been standardized in MPI-3 [93]. It enables MPI processes residing on the same node to allocate a shared memory region for direct load/store accesses. Similar to HMPI, it improves intra-node bandwidth. The interfaces for multi-threaded inter-node communication remain unaffected.

To take advantage of multiple threads in the many-sender pattern, while preserving the strengths of minimal message processing overhead in the single-sender pattern, two proposals have been recently published.

In the so-called *endpoints* proposal, multiple threads running in the same MPI process are assigned logical ranks. Endpoints do not change the communication model, however, enable per-thread addressability [52]. Although benchmarks have demonstrated improved performance, endpoints require non-negligible code changes to allow interoperability between MPI and threading libraries.

A more promising approach are *Finepoints*, which is discussed for adoption as *partitioned point-to-point communication* in the upcoming MPI-4 major release [74, 151]. The overall design is a hybrid of the single- and many-sender patterns. Similar to persistent point-to-point communication, communication peers (i.e. sender and receiver) first agree on an upcoming message exchange. Besides common parameters for message matching, both sender and receiver specify the number of partitions in the send and receive buffers, respectively. When a task has completed, the thread notifies the MPI library about partition readiness for message transmission. Whether transmission immediately happens depends on the MPI library. To guarantee consistency, applications explicitly synchronize ongoing communication requests with conventional wait/test semantics. Essentially, *Finepoints* allow contributions from multiple tasks to a single message on the sender side. On the receiver side, it minimizes message matching overhead because multiple partitions are semantically transferred as a single message. Furthermore, instead of waiting for the whole message, *Finepoints* provide non-blocking probes for arrival of individual partitions in a single message (request). Utilizing this interface leverages additional overlap potential, in particular if many tasks cooperatively act on the receive buffer. MPI libraries are encouraged to balance the latency and bandwidth trade-off through message aggregation.

The concept of *Finepoints* addresses weaknesses of both the single- and many-sender patterns with three design goals:

- Data movement happens as soon as data becomes ready (early binding).
- Communication overhead is minimal because individual partitions are semantically part of a single message, which is beneficial especially on the receiver side.
- Improved interoperability with thread-level parallelism, where multiple tasks cooperatively operate on a single buffer in shared memory.

Because the communication model is not fundamentally different from traditional point-to-point operations, adoption of legacy codes in scientific applications is reasonable with minimal modifications.

3.4.3 Deficiencies in Collective Primitives

The single- and many-sender patterns apply to collective primitives as well. Negative effects on performance even multiply, because collectives primitives involve a possibly large number of message exchanges to fulfill the communication pattern. We identify two additional concerns, partial completion and canonical buffer displacement, which are elaborated below:

- Partial completion.
- Canonical Buffer Displacement.

3.4.3.1 Partial Completion

Collective primitives are commonly implemented as a series of point-to-point communications. However, the opaque interface of MPI requests does not exhibit any information on the state of pending or completed stages. Therefore, MPI processes cannot compute on partially completed communication buffers. Taking the various sources of synchronization overhead into account, this can waste a significant portion of overlap potential.

As an example, in many stencil solvers, each neighborhood relationship can be independently processed from the other neighbors. However, due to the restrictive interface of collective primitives, the application cannot consume for specific neighbors. Instead, each MPI process needs to wait until the collective communication is finished, before continuing with computation. This leads to implicit synchronization, which is prone to noise propagation, and further limits available parallelism.

The rational of partial completion is comparable to partitioned communication in *Finepoints*. Instead of independent contributions from multiple threads to one message, we look into independent messages in a single collective communication. In fact, *Finepoints* complement with partial completion in collectives, as each message can be further decomposed into smaller partitions, obtaining additional performance efficiency.

Partial completion requires a mechanism to act on individual messages, while progressing a collective communication operation. Related research has studied possible approaches based on the significantly revised MPI Tools interface for the upcoming MPI-4 major release [87, 151]. The MPI-4 Tools interface enables to register callbacks within MPI libraries, which are triggered in implementation-defined events. Unfortunately, the most recent MPI-4 draft does not specify a list of concrete events following the MPI Tools interface. Hence, it cannot be guaranteed that a specific MPI library on a particular platforms supports required events for partial completion during collective communication.

A recently published paper proposes to standardize a set of specific events related to partial completion in collectives [37]. In this approach, it is up to the threading runtime (e.g. OpenMP) to register relevant callbacks to operate on receive buffers, while collective communication is still in progress. As of today, it remains unclear whether these suggestions will be accepted for future MPI standardization. We argue that improving overlap is not the primary goal of the MPI Tools interface. The main target is to yield necessary information at runtime to better understand possible performance bottlenecks.

3.4.3.2 Canonical Buffer Displacement

Another restriction with collective primitives is buffer placement. In message passing, each message is copied from a source to a destination buffer. The consequence for collective primitives is that each participating MPI rank in the respective communicator needs to specify the displacement for send and receive buffers, respectively. While this does not cause additional overhead in regular collectives with uniform message sizes, it is a more severe issue in case of irregular collectives with non-uniform message sizes. Irregular collectives require additional memory, proportional to the number of ranks, only to specify the displacement for each communication peer. If message sizes are unknown prior to runtime, many applications first communicate message sizes to obtain displacements, followed by the actual data exchange [94]. Although there are strategies to transform irregular to regular problems through pipelining, the limitation for buffer placement persists [187]. Guaranteeing canonical order of received messages sometimes requires additional memory copies after completing all communication steps, e.g. in dissemination algorithms [32].

In Ch. 5, we show that in many cases it is sufficient to allocate required memory buffers, without any restrictions on the displacement of arriving messages.

3.5 Summary

This chapter characterizes various sources of communication overhead in a message passing programming model. To quantify these overheads from a theoretical perspective, researcher have proposed various communication cost models.

In our approach, we rely on the LogGP model, as it captures relevant parameters to analyze scientific applications. The LogGP model has been repeatedly applied to derive optimal algorithms for collective communications [32, 103]. Collective communications are among most used primitives in HPC applications, as they support in designing efficient parallel algorithms and are heavily optimized for different network topologies. Although communication algorithms are continuously improved, many HPC applications are communication-bound. Examples include stencil kernels, as shown with a FFT case study, but also combinatorial algorithms. In a recent paper, we have studied the distributed sort problem, where communication costs scale proportional to the number of PEs [129]. In communication-bound algorithms, data movement is inherently the main performance bottleneck.

Therefore, latency hiding is a crucial pillar to obtain scalable performance efficiency. However, interactions between orthogonal programming models in shared and distributed memory remains challenging. Several approaches improve interfaces for multi-threaded message passing to either reduce or eliminate synchronization costs (cf. Sec. 3.4.2). While these efforts are a step in the right direction, none of them aims at collective primitives. We have discussed two deficiencies in collective communication primitives, which limit latency hiding:

- No partial completion, i.e. consuming data as early as possible (early binding).
- Canonical buffer displacement, which requires messages in a contiguous range to be in-order.

Both issues compel HPC users to implement communication in terms of low-level protocols, e.g. send-receive pairs. The increasing complexity contradicts the philosophy of abstraction, as collective communication primitives promise performance and expressiveness [72]. With this at hand, we have addresses the following research questions:

SQ1 What applications are sensible to the performance of collective communications?

SQ2 What deficiencies in collective communication primitives limit the degree of parallelism?

Based on above mentioned observations, Ch. 5 proposes partial aggregation to break the monolithic interface of collective communication primitives. Instead of improving the efficiency of low-level message passing protocols, we expose algorithmic parallelism in collective communications to significantly improve latency hiding potential.

4 Algorithms for Collective Communication

Collective communication primitives abstract parallel communication patterns in a set of uniquely indexed ranks. The level of abstraction does not only benefit programmability. It simplifies performance portability and avoids common errors in hand-tuned communication algorithms using point-to-point communication [89].

Non-blocking interfaces additionally allow the programmer to overlap collective communication with computation. Early binding and manual transformation techniques have become ubiquitous optimizations to obtain performance efficiency [98, 131]. The advantage of non-blocking collectives compared to blocking is the ability to absorb implicit synchronization. Although collective primitives are not required to be synchronizing, data dependencies in underlying algorithms and message passing protocols often force it. To better understand data and synchronization dependencies, we summarize fundamental building blocks for collective communication.

4.1 Overview

Algorithms for collective communication can be classified into two approaches:

Topology-aware algorithms These algorithms take the network topology into account and try to minimize the number of communication steps [14, 29]. Various kinds of personalized all-to-all communications are of particular interest, as conflicts and congestion significantly degrade performance. Although these algorithms are highly efficient, they are not portable, because other HPC platforms may rely on different negtwork topologies.

Generic algorithms These algorithms assume a fully interconnected network, based on the assumption that a single message transmission between any pair of PEs takes roughly the same time. Similar to topology-aware approaches, the goal is to minimize both the number of communication rounds and to fully exploit available network bandwidth, without causing network congestion. However, there are no assumptions about the underlying network topology, which significantly increases portability.

Another property of collective communication is whether it is *personalized* or *non-personalized*:

Non-personalized collectives Each rank send the *same* message to their destinations. An example in the MPI standard is `MPI_Bcast` (broadcast) operation, where a single message is propogated from the root to all other ranks.

Personalized collectives Each rank sends a *distinct* message to its destinations. An example in the MPI standard is `MPI_Scatter`, where the root distributes a vector of p data items to p ranks.

We consider only *generic* algorithms to model (non-)personalized collectives, as we want to benefit HPC codes on all platforms from our concepts. Communication costs are approximated using the LogGP communication model. To keep the model simple, we assume that the overhead per message is larger than the message gap (i.e. $o > g$), which is reasonable on recent HPC systems [103]. The list of generic communication algorithms includes the following.

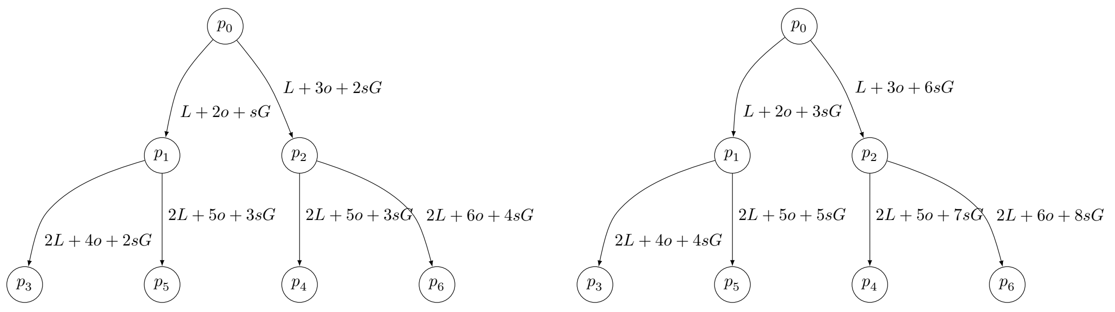
- Tree-based algorithms
- Linear pipelines and rings
- Circulant graphs

4.2 Tree-based Algorithms

Tree-based algorithms are commonly used in rooted collectives. A common example is broadcast, where a given root rank propagates a message to all other ranks in a given communicator. Similarly, collective reduction accumulates data across all ranks and the result is stored on a given root rank.

A generic abstraction to model rooted collectives are trees, where we distinguish between *regular* and *irregular* trees.

4.2.1 Regular Trees



(a) Non-personalized. (b) Personalized.

Figure 4.1: Binary tree broadcast algorithm with 7 ranks.

.

Regular trees hierarchically arrange the set of uniquely indexed ranks such that each node has k children. With the notion of k-ary trees, we can model multiport message passing systems. Multiport means, that each ranks can send k distinct messages to k ranks, and simultaneously receive k messages from k other ranks.

Depending on the semantics of the collective operation, the tree is traversed either top-down (e.g. *broadcast*) or bottom-up (e.g. *gather*). We illustrate the algorithm with a personalized and non-personalized broadcast, which are standardized in MPI as `MPI_Bcast` and `MPI_Scatter`, respectively.

4.2.1.1 Non-personalized Broadcast

Given a k-ary regular tree, a broadcast operation is implemented as follows [13].

- The *root* rank sends k copies of the message, one after the other, to its k children, in order from left to right.
- Ranks other than the root first wait for message arrival from the parent. Upon receiving a message, this message is forwarded to at most k children, in order from left to right.

Fig. 4.1a visualizes a non-personalized binary tree broadcast ($k = 2$) with 7 ranks. The circles are labelled as ranks, while the edges are labelled with communication costs. Rank p_0 is the root and sends messages of size s to the other peers. Approximating the communication costs for a general k-ary tree works as follows. For clarity, we first explain the binary tree case.

We start from the root with rank pair (p_0, p_2):

- p_0 sends the last byte to p_2 at time $t_0 = k(o + (s-1)G)$.
- p_2 receives the last byte at time time $t_1 = t_0 + L + o$.

The same procedure applies to pair (p_2, p_6). If p_6 receives the last byte, the algorithm terminates. Generalizing the algorithm to a k-ary tree is straightforward. As the number of message startups is at most $\lfloor \log_k p \rfloor$, the rightmost child in a non-personalized tree receives the last byte at time:

$$T_{kt} = \lfloor \log_k p \rfloor (L + k(o + (s-1)G) + o) \tag{4.1}$$

4.2.1.2 Personalized Broadcast

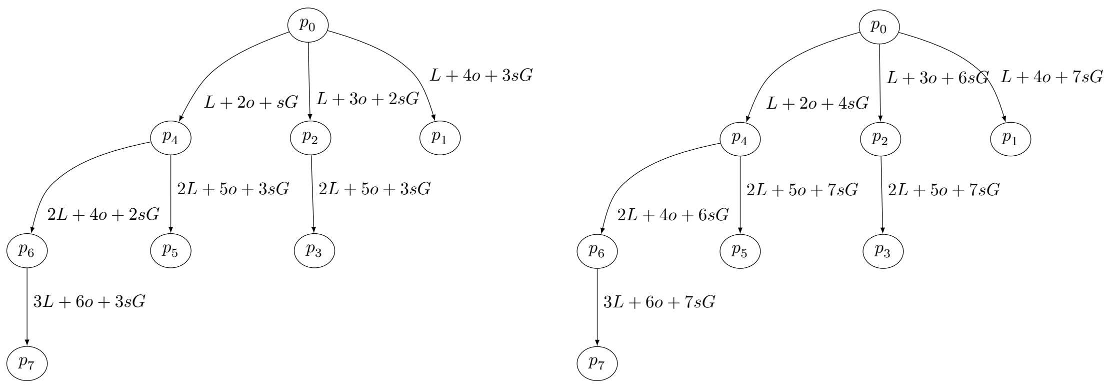
For the personalized variant (Fig. 4.1b), which we denote with $\dot{T}_{kt}$, only the bandwidth term changes. Similar to the non-personalized version, it suffices to accumulate communication along the rightmost path, as shown in Fig. 4.1b. The difference is that the root sends this time $s(p/2)$ bytes to its children. Ranks on the next level send $s(p/4)$ bytes to their children, etc. In general, the number of bytes to send for non-leaf ranks is effectively halved in each level. This leads to approximated communication costs as follows:

$$\dot{T}_{kt} = \lfloor \log_k p \rfloor (L + o(k+1)) + (s-1)G \sum_{i=0}^{\lfloor \log_k p \rfloor} (\lfloor \log_k p \rfloor - i) k^{i+1} \tag{4.2}$$

4.2.2 Irregular Trees

While regular trees are optimal for small messages due to low latency, i.e. logarithmic in the number of ranks, they are not asymptotically optimal. Irregular tree shapes can further improve lower bounds. As an example, an optimal non-personalized broadcast tree can be constructed based on Fibonacci trees [119]. A similar approach has been applied to optimize a personalized broadcast (*scatter*) [4]. However, none of these algorithms has demonstrated practical use as they are difficult to implement.

A more simple class of trees are k-nomial trees, which are widely used in computer science. We illustrate the concept with binomial trees, i.e. $k = 2$. A study of state-of-the-art open source MPI libraries has shown that binomial trees are the most used tree structure for implementing collective communication [65, 75]. Fig. 4.2 visualizes a binomial tree broadcast from p_0 as the root to all other ranks. The advantage of binomial trees is that they achieve a better occupation of the available injection bandwidth. To distribute the message as fast as possible across the network, the root of the left subtree (p_4) receives the first segment. In the second round, rank p_4 forwards the segment to its k children, while rank p_0 concurrently serves the next subtree. Compared to binary trees, where ranks in level j are idle after finishing round j, the binomial tree utilizes all nodes until the algorithm has finished.



(a) Non-personalized. (b) Personalized.

Figure 4.2: Binomial tree broadcast algorithm with 8 ranks.

To approximate the overall runtime cost of a non-personalized binomial tree broadcast, we count the number of segments along the leftmost path in Fig. 4.2a.

$$T_{bt} = \lceil \log_2 p \rceil (L + 2o + sG) \tag{4.3}$$

Accordingly, the personalized version is approximated as follows:

$$\dot{T}_{bt} = \lceil \log_2 p \rceil (2o + L) + sG(P - 1) \tag{4.4}$$

Both equations show that binomial trees exhibit the same latency term as regular trees. However, they achieve an asymptotically optimal bandwidth term, both in personalized and non-personalized versions. Note that asymptotic complexity does not imply practical performance. The reason is that the root of a particular binomial tree has to transmit messages to all children in its subtrees. If message sizes increase, this leads to a significant bottleneck, and more advanced algorithms need to be considered [38].

4.3 Pipelines and Rings

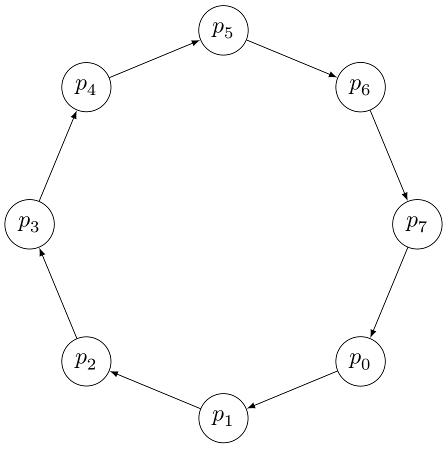


Figure 4.3: Linear array/ring topology with 8 ranks.

There are two problems with binomial trees. First, the tree root needs to send a message to $\lceil \log p \rceil$ other ranks. And second, if we transmit the whole message at once, the bandwidth term grows proportional to the number of ranks. This is reasonable with small messages. For large messages, if $s \gg p$, the root rank becomes a significant bottleneck.

A possible fix to the first problem is to arrange all ranks in a linear array topology, as shown in Fig. 4.3. Although it is simple, it reduces the number of ranks connected to the root. The second problem can be solved through pipelining, which is already discussed with tiling in Sec. 3.3 to improve overlap. Instead of transmitting a single message, we partition the message into S blocks of size s/S bytes.

Implementing a broadcast in a linear array with pipelining is simple. We start with the root node being p_0.

- p_0 sends S blocks to p_1, one after the other.
- p_1 forwards each received block to p_2, while receiving the next block from p_0.
- Generally, p_i sends block j to p_{i+1}, while receiving block $j+1$ from p_{i-1}.[4]

The last rank receives the first block after $p-1$ rounds, and the last block after another $S-1$ rounds. It follows that the total communication cost can be approximated as follows:

$$T_{lp} = (p + S - 2)(L + 2o + \frac{s}{S}G) \tag{4.5}$$

[4]The leftmost (rightmost) rank only sends (receives) blocks.

An interesting question is to determine the optimal block size to balance the latency and bandwidth terms. Applying linear calculus to Eqn. (4.5) leads to the optimal threshold S_{opt}:

$$S_{opt} = \sqrt{\frac{p - 2sG}{L + 2o}}$$

Eqn. (4.5) shows that pipelining has an optimal bandwidth term, however, a suboptimal latency term. A common trade-off is to integrate pipelining into binary trees, as described in Fig. 4.1. The root PE sends a block to two children, in order from left to right. For very large messages, this improves the overall bandwidth term to a logarithmic factor of p:

$$T_{btp} = (\lfloor \log p \rfloor)(L + o) + 2(\lfloor \log p \rfloor + S - 1)(o + \frac{s}{S}G) \tag{4.6}$$

Pipelining in binary trees is often utilized in practice, e.g. in split-binary or fractional tree non-personalized broadcasting [165, 171]. For personalized broadcasts, these approaches perform worse compared to traditional k-nomial trees and linear pipelines.

An important observation with tree-based algorithms is the duality of operations. For example, reversing the direction in a binomial *broadcast* tree provides an efficient *reduce* algorithm. The same holds for *scatter* and *gather* operations.

4.4 Circulant Graphs

Another relevant class are non-rooted collectives, i.e. many-to-many, where p ranks exchange data with $p - 1$ other ranks. In a naive implementation, this results in p^2 communication pairs, causing a congested network particularly with large messages. A simple yet powerful improvement is to combine rooted tree algorithms to implement many-to-many distributions. For example, a *reduce* to any rank p_i, followed by a *broadcast* from p_i to all other ranks, implements a global *all-reduce.*

However, due to the importance of various many-to-many distributions, more scalable algorithms to achieve performance are subject to active research. We briefly summarize established alternatives by focusing on two primitives:

- *all-gather*, and
- *all-to-all.*

Both operations are widely used in data-intensive applications. The *all-gather* primitive is a non-personalized *all-to-all* broadcast, where each PE broadcasts the same message to all other PEs. The personalized counterpart is *all-to-all*, where each rank scatters individual blocks of data to all other PEs. For this reason, *all-to-all* is considered as the least scalable communication algorithm. Nevertheless, it plays an important role in many HPC codes to shuffle huge data volumes, e.g. in FFT (cf. Sec. 3.3.2) or distributed sort algorithms [129].

All-gather

We first consider the *all-gather* primitive, which is not only relevant in applications, but often used to implement other MPI primitives. As an example, MPI libraries often implement non-personalized broadcast (`MPI_Bcast`) as a combination of a *scatter*, followed by an *all-gather* communication. A similar pattern is used in *all-reduce*. This paradigm has shown to be asymptotically optimal in several cases and is often used in large message transfers [15].

Given p PEs, each contributing a block of s bytes. An all-gather collects all blocks on all PEs, such that each PE has p blocks of size $p \times s$ bytes in its receive buffer. The simplest algorithm is a linear ring pipeline, which is already discussed in previous section. In $p-1$ rounds, each PE sends (receives) s bytes to (from) its left neighbor in the ring. The overall complexity is the same as in Eqn. (4.5) ($S=1$). Note that there is no possibility to improve the bandwidth term as each PE needs to receive at least $(p-1)s$ bytes.

To improve latency for small messages, there are two general approaches. The first is a recursive doubling algorithm, following the same communication pattern as a binomial tree reduction. It requires $\lfloor \log p \rfloor$ communication rounds and each PE transmits $(p-1)s$ bytes in total. This is asymptotically optimal both in terms of latency and bandwidth requirements. However, both network traffic and node distance effectively double in every round, which is unfavorable for large message buffers. Moreover, if p is not a power of two, message startup overhead doubles to $2\lfloor \log p \rfloor$ rounds [184].

The second alternative to obtain a logarithmic latency term is a generalization of a dissemination barrier, often called the Bruck algorithm [32, 85]. It takes $\lceil \log p \rceil$ communication rounds for any number of PEs, and each PE transmits $(p-1)s$ bytes. Message blocks are rotated leftwards within a circulant graph. More specifically, in round j, rank i sends data leftwards to rank $(i-2^j) \bmod p$. After performing all communication rounds, each rank has received message blocks from all other rank. However, message blocks are not in order for some ranks. The MPI standard prescribes a canonical order of received message blocks, such that data from rank i is displaced at bucket i in the corresponding receive buffer. Fixing this requires another, potentially expensive, memory copy to complete the *all-gather* operation.

Fig. 4.4 shows a *all-gather* communication graph for 8 ranks using the dissemination algorithm. It requires 3 communication rounds, which are denoted in 3 different line styles, respectively. Each rank sends (receives) at most $\frac{p}{2}s$ bytes per round.

Analyzing open source MPI libraries shows that the recursive doubling and dissemination algorithms are often used for small messages. In contrast, variants of ring algorithms are used in large messages transfers.

All-to-all

The *all-to-all* primitive is the most general and also most challenging communication pattern. Optimal solutions are hard to derive and still subject to active research. The limiting factor is the network's bisection bandwidth, which defines the minimum number of edges that must be removed to partition interconnected nodes into two equally sized partitions. The trade-off between a high bisection bandwidth and low diameter is a major

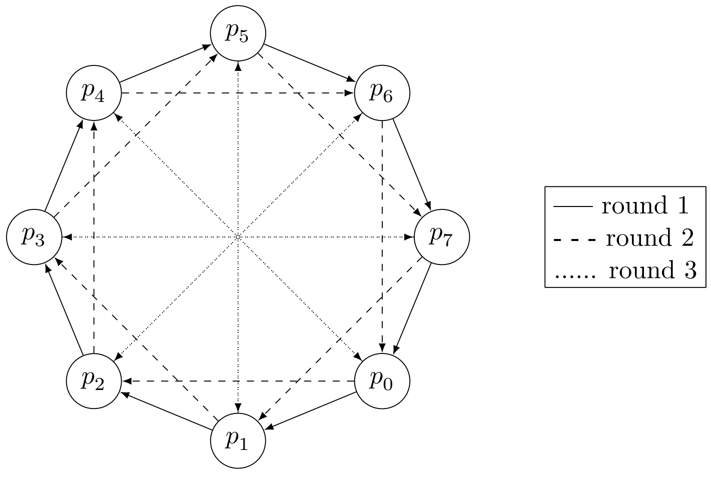


Figure 4.4: Dissemination algorithm with 8 ranks.

challenge in HPC network design [73]. Assuming a fully interconnected network, the bisection bandwidth is $\lfloor \frac{p}{2} \rfloor \times \lceil \frac{p}{2} \rceil$.

Like in *all-gather*, there exist two approaches. For relatively small messages, message combining based on the dissemination algorithm in Fig. 4.4 is commonly used. The radix k facilitates a trade-off between the number of communication rounds per data block (latency), and corresponding bandwidth consumption.

- The total number of message startups is $(k-1)\lceil \log_k p \rceil$.
- In each round, a single PE transmits at most B data blocks, where $B = (k-1)\lceil \frac{p}{k} \rceil$.

Most libraries rely on radix $k = 2$, although a larger radix can be useful in practice. It requires $\lceil \log p \rceil$ communication rounds, and each PE transmits at most $p/2$ data blocks per round. On top of that, the algorithm requires two memory copy operations, before and after the global communication phase, to guarantee canonical order of received message blocks. Taking this into account, the communication cost of *all-to-all* dissemination, also called personalized butterfly (T_{PBF}), can be approximated as follows:

$$T_{PBF} = \lceil \log_k p \rceil (k-1)(L + 2o + \lceil \frac{p}{k} \rceil sG) + \mathcal{O}(ps) \tag{4.7}$$

It is easy to see that Eqn. (4.7) exhibits a suboptimal bandwidth term, as each message block is transmitted a logarithmic number of rounds across the network. For large message sizes, pairwise message exchanges using direct sends are more efficient. Each rank sends a message to all other ranks in $p-1$ communication rounds. The main challenge is to find a schedule of message pairs, which effectively utilizes bisection bandwidth. We summarize three possible approaches.

1. The first approach relies on personalized ring topologies, a simple and often used communication pattern. In round j, rank i receives from rank $p_r = (i-j) \bmod p$, and sends to $p_s = (i+j) \bmod p$. This schedule guarantees that no rank receives more than one message per round, and further uses full duplex capabilities in the NIC.

2. The second approach is an optimization, if the number of PEs is a power of two [175]. In this case, message exchanges follow a hypercube communication pattern. In hypercube *all-to-all*, ranks pair with a distinct partner every round. This algorithm has the property that message exchanges are increasingly contained. The first exchange is performed by message pairs whose ranks yield the same quotient when divided by two. In the second round, ranks yielding the same quotient when divided by four are paired. The divisor increases in powers of two, starting from 1 and up to $p - 1$ rounds. Determining the message partner each round requires only a single *exclusive-or* operation, and thus is easily implementable in hardware.
3. The third approach relies on one-factorization in graph theory [172, 195]. A fully connected network is partitioned into p subgraphs, in which each node has an exact degree of 1. The resulting set of edges connects two nodes in p subgraphs, determining the communication partners in each round, respectively. For even p, the number of communication rounds can be reduced to $p - 1$, which excludes self loops.

4.5 Practical Implementations

Trees, graphs and simple pipelines provide a trade-off between latency, and bandwidth overhead. We briefly summarize, how practical implementations utilize hybrids of these algorithms to improve performance scalability.

4.5.1 Hierarchical Collectives

Optimized implementations distinguish between intra-node and inter-node bandwidth communication. Instead of spanning all ranks in a single tree, one rank per node is selected as coordinator, performing communication on behalf of the other ranks within the node. This approach enables hierarchical collective communication. In a broadcast, the root first broadcasts the message to all other node coordinators. When a coordinator has successfully received a message, it broadcasts the message in shared memory. This significantly reduces network traffic, as fewer ranks contend for available bandwidth. To further improve performance, different algorithms are used per layer. Usually, inter-node communication is performed in a linear pipeline, while intra-node communication is handled in a binomial tree [122, 124].

A similar approach with hybrid algorithms has been examined to further improve the latency bandwidth trade-off for large messages. The general idea is to decompose one large message collective into a sequence of multiple small message collectives [15]. As an example, a large message broadcast is decomposed into a scatter, followed by an all-gather operation. This reduces the bandwidth costs in binomial tree algorithms. The same principle applies to *all-reduce*, which is often implemented as a *reduce-scatter*, followed by an *all-gather*.

This approach generalizes to hierarchically organized ranks as well. As an example, ranks can be arranged in two-dimensional grid with separate communication along each row and column. In this setting, a broadcast can be implemented as follows. In the first step, the root scatters the message along the row dimension. Ranks in the corresponding row communicator broadcast the message along the column communicator. A final all-gather along the rows completes the broadcast.

A more recent paper extends this concept to further optimize collective communication in shared memory, taking NUMA effects into account. Results indicate, that binomial trees and dissemination-based reduction can even outperform OpenMP reduction in parallel loops, the latter probably being the most-used shared memory library in HPC codes [39, 135].

4.5.2 Non-Uniform Message Lengths

Up to this point, the discussed communication algorithms have assumed uniform message sizes, i.e. all ranks contribute equally sized message blocks. However, many problems pose non-uniform data distributions, where communication is either sparse or imbalanced. While proper data decomposition strategies can minimize load imbalance across PEs, skew data distributions still occur at runtime.

The MPI standard specifies irregular primitives for various communication patters (e.g. *scatterv*, *all-to-allv*) to handle these use cases. However, these algorithms are difficult to optimize. Although asymptotic lower bounds for sparse data exchanges are available [103], achieving performance still requires sufficient application knowledge to implement these communication patterns. Another major challenge are data-intensive applications, which shuffle huge data volumes across the network. Message sizes are relatively large and exhibit a skew distribution among involved ranks [187]. An example are n-body simulations, where communication is predominantly irregular. Because processed bodies can migrate between ranks, data needs to be repartitioned to balance the workload after a specified number of simulation steps. These algorithms are often implemented with distributed sort algorithms, or distributed hash tables [129, 170].

Implementing irregular collectives with common tree-based algorithms may lead to bottlenecks close to the root. The reason is that processing times in subtrees (e.g. in a binomial tree) may differ, and these delays propagate across all levels in the tree. Algorithms to mitigate these issues for specific variations of irregular collective communication have been studied [185, 187]. However, these approaches are either too expensive with small messages sizes or, too complex to implement them in general purpose libraries.

The problem of finding an optimal schedule for irregular communication patterns is not addressed this thesis. However, concepts of partial aggregation, which we introduce in subsequent sections, can hide latency bottlenecks through efficient communication-computation overlap.

4.6 Summary

Collective communications are commonly implemented as a series of send/receive pairs between involved ranks, which we call a schedule. This chapter presents algorithmic building blocks to derive efficient communications schedules, based on various shapes of trees, circulant graphs and linear arrays. Presented algorithms can be categorized into two approaches:

- Direct message transmissions, where a message is directly routed from the source to destination rank. Direct transmissions are primarily used for large messages where the data transfer rate dominates communication costs rather than latency.
- Intermediate transmissions, where a message is routed from the source to the destination rank in intermediate steps. This approach is used for small message transmissions, where latency dominates the bandwidth term.

We approximate communication costs of presented algorithms based on the LogGP model. The LogGP model also reflects parallelism in collective communications as it encodes both the number of message startups in a communication schedule (latency term) and the data transfer rate (bandwidth term). In Ch. 5 we show how this parallelism can be exposed to the user. We present our concept of partial aggregation, enabling to operate on message buffers of a pending collective communication.

5 Partial Aggregation in Collective Communication

In this chapter we address interface deficiencies in collective communication primitives. Our approach is based on grouping and aggregation of data sets as fundamental building blocks to improve collective communications. Essentially, computation can operate on message buffers, although the respective collective communication is still in progress. Based on semantics of binary operators, we expose inherent data parallelism to allow efficient latency hiding techniques.

The remainder of this section is organized as follows:

1. We provide a semi-formal model to define requirements for partial aggregation in distributed data sets.
2. We show, how this model can be integrated into collective communication primitives, as specified in the MPI standard.
3. We analyze simple, yet representative, use cases to demonstrate how partial aggregation improves HPC codes.

5.1 Requirements for Partial Aggregation

Restricted programming models based on concepts of data dependencies often exploit algebraic properties to automatically extract concurrency. A common approach is to compute and aggregate partial results in parallel algorithms to mitigate I/O bottlenecks. Literature describes various forms of partial aggregation on distributed data sets, which we categorize into the following:

Decomposition Computation of an usually *large* data set is initially decomposed into independent smaller portions. In a later step, *partial* results are aggregated to obtain a consistent output.

Fusion A sequence of computations on a data set is fused into a single, semantically equivalent, operation.

An intuitive example for *decomposition* are parallel scan operations, which are ubiquitous in many HPC applications. Given an associative but not necessarily commutative function, prefixes for n elements distributed on n ranks can be computed in $\mathcal{O}(\log n)$ time [26]. A rooted tree (e.g. binomial tree) of depth $\log n$ spans all ranks. In each level of the tree, pairs of ranks compute intermediate results in parallel (up-sweep), which are then recursively combined (down-sweep). Hence, the overall algorithm requires $\mathcal{O}(2 \log n)$ steps.

The same pattern be found in collective *(all-)reduce* computation, where distributed data is accumulated using an associative binary function. If this function is additionally commutative, distributed data can be combined in an arbitrary order.

Decomposition also plays an important in the Map-Reduce programming model. Map-Reduce provides a *combiner* interface to pre-aggregate map tasks. After intermediate results are grouped by key in the *map* phase, the *combiner* emits partial results for each key. Partial results usually have a much smaller size than the set of intermediate results, that would have been transmitted without aggregation. Partial aggregations can be used at multiple levels in the network hierarchy to exploit data locality and reduce overall communication overhead.

In contrast, the *fusion* pattern follows the opposite idea. A notable example is to fuse multiple subsequent collective computations into a single operation due to the *law of distributivity* [71]. As an example, a scan operation followed by a reduction can be fused into a single reduction, given that the reduction operator *distributes* over the scan operator. In case of small messages, this reduces overall network traffic. In practice, however, this usually requires a user-defined reduction function. Optimized message passing libraries exploit dedicated hardware features to support *native* reduction operators. Exploiting these features results often in better performance.

In our work, we focus on the *decomposition* pattern in collective operations. The rational is to increase parallelism, allowing more efficient latency hiding. Below, we list mathematical requirements to decompose collective operations:

- Homomorphism
- Decomposable functions

5.1.1 Homomorphism

Distributed memory algorithms, following the BSP model, are expressed as a sequential composition of parallel stages: Computation f and communication g, where g is assumed to be a, possibly non-blocking, collective operation. We model the sequence of parallel stages f and g with function composition ($\circ$):

$$(f \circ g)x \equiv f(g(x))$$

Parallelism of stages f and g is described with the notion of homomorphism, which preserves the operations of elements in algebraic structures [25].

Definition 1 (Homomorphism)**.** *We use $\overline{x}$ to denote a vector of data items, while $\overline{x} \mathbin{+\!\!+} \overline{y}$ denotes concatenation of $\overline{x}$ and $\overline{y}$. Function h is homomorph, iff there exists an operator $\odot$, such that:*

$$h(\overline{x} \mathbin{+\!\!+} \overline{y}) \equiv h(\overline{x}) \odot h(\overline{y})$$

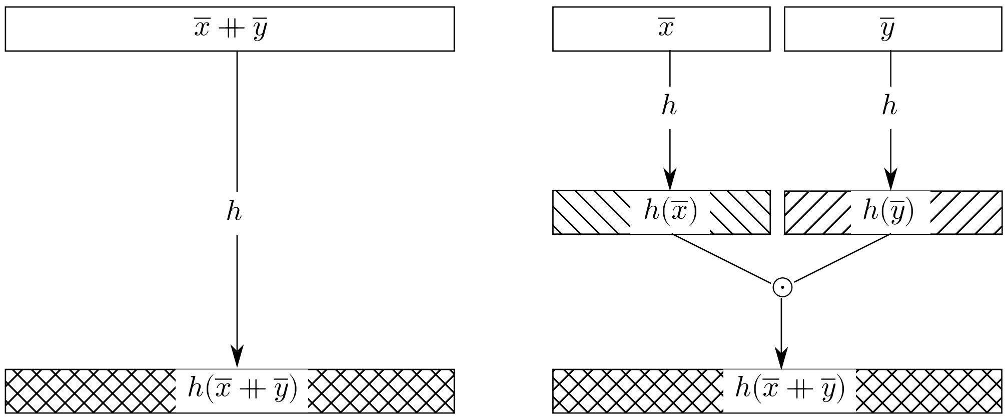

$\overline{x} + \overline{y}$
h
$h(\overline{x} + \overline{y})$
$\overline{x}$
$\overline{y}$
h
h
$h(\overline{x})$
$h(\overline{y})$
$\odot$
$h(\overline{x} + \overline{y})$

Figure 5.1: Computation of homomorphic function h, according to Def. 1.

Left: Sequential algorithm, Right: parallel algorithm.

Intuitively, this means that the result of h on a concatenated vector is the same as applying a combiner $\odot$ to partially computed pieces, visualized in Fig. 5.1. The left hand side shows a sequential computation of function h on a single concatenated vector. The right hand side shows a parallel computation, where the vector is partitioned into two pieces, $h(\overline{x})$ and $h(\overline{y})$. As both pieces are independent operations, they can be processed in parallel, and later combined with the combine operator $\odot$.

Homomorphism is excessively used in functional programming and task-parallel constructs as it supports supports systematic extraction of parallelism in the divide-and-conquer paradigm through introducing auxiliary functions [71];Hu, Iwasaki, and Takechi [105]]. For example, OpenMP relies on homomorphism to implement reductions in parallel shared memory regions [154]. Below, we utilize the homomorphic property to define *decomposable functions*, serving as the basis for partial aggregation.

5.1.2 Decomposable Functions

Decomposable functions extend the notion of homomorphism such that the combine operator $\odot$ is expressed in terms of two auxiliary functions: *Accumulate* and *Merge*. Accumulate yields partial results from intermediate values. If initial accumulation is not necessary, it can also be a simple passthrough function. While partial results are transmitting to their destinations, they are recursively merged until computation is done. Because partial results can arrive in an arbitrary order from the network, the functions *accumulate* and *merge* need to satisfy the concept of decomposable functions.

Definition 2 (Decomposable functions)**.** *A function H is decomposable, if there exist two functions A and M satisfying the following criteria.*

1. *H is the composition of A and M:*

   $\forall\, \overline{x}, \overline{y} : H(\overline{x} \odot \overline{y}) \equiv M(A(\overline{x} \odot \overline{y})) \equiv M(A(\overline{x}) \odot A(\overline{y}))$

2. *A is commutative:*

   $\forall\, \overline{x}, \overline{y} : A(\overline{x} \odot \overline{y}) \equiv A(\overline{y} \odot \overline{x})$

3. M is commutative:

$\forall\, \overline{x}, \overline{y} : M(\overline{x} \odot \overline{y}) \equiv M(\overline{y} \odot \overline{x})$

Intuitively, A and M correspond to the functions *accumulate* and *merge*, respectively. Requirement 1 ensures the semantics of M, so that combining two partial results yields either the final or another partial result. Requirements 2 and 3 ensure that partial results can arrive at any order. Commutativity is crucial in partial aggregation. Several bugs in MapReduce applications have been reported due to non-commutative reduce operations [200].

Besides decomposability, another algebraic property for partial aggregations is often provided.

Definition 3 (Associative-decomposable functions)**.** *A function H is associative-decomposable, if there exist two functions A and M satisfying conditions 1–3 in Def. 2, and the associative property:*

$$\forall\, \overline{x}, \overline{y}, \overline{z} : M(M(\overline{x} \odot \overline{y}) \odot \overline{z}) \equiv M(\overline{x} \odot M(\overline{y} \odot \overline{z}))$$

Informally, it means that the aggregation order of partial results, residing on different ranks, does not affect correctness of the final result (Def. 1). More precisely, if a computation satisfies the concept of *associative-decomposable* functions, we can construct a multi-level aggregation tree where partial results are iteratively combined. In contrast, if a computation step satisfies only *decomposability* (Def. 2), we can still partially aggregate, however, without any intermediate steps.

5.2 Non-Blocking Collective Aggregation Trees

A collective aggregation tree integrates the concept of decomposable functions and collective communication. The main goals are:

- Exposing available parallelism in collective communication.
- Reducing interface deficiencies between collective primitives and task-parallel programming models.

The remainder of this section is organized as follows. We first describe data and synchronization dependencies in collective communication algorithms. While data dependencies are inevitable for correctness, synchronization dependencies can be classified as algorithmic latency.

Then, we extend the interface for collective communication primitives, as specified in the most recent MPI standard. However, we consider only the receiving side of collective communication primitives. We further restrict the available scope to use cases, where a participating rank receives a set of message blocks from all other ranks in the same communicator, i.e. many-to-one or many-to-many patterns.

The overall concept consists of two contributions:

Algorithm 4: Pipelined tree broadcast with non-blocking sends.

```
input: Number of segments $S$
for $k \leftarrow 0$ to $S$ do
    MPI_Recv($k$, parent)
    for $i \leftarrow 0$ to outdegree do
        MPI_Isend($k$, $i$)
    end
    MPI_Waitall() // Locally synchronize non-blocking sends
end
```

1. We propose an interface for *partial completion* in collective communication primitives with minimal modifications to the existing MPI-3 interface. HPC application which rely on non-blocking collectives can immediately benefit from partial completion with minor code changes.
2. We extend the interface to incorporate the concept of *decomposable functions* into the interface for collective primitives. Providing the MPI libraries additional hints enables further optimizations in collective communication algorithms. We discuss the optimization space and possible impacts on application performance.

5.2.1 Dependency Analysis in Collective Communication Algorithms

In a collective communication schedule, each PE has a personal schedule, derived from tree- and graph-based algorithms to fulfill the collective communication pattern (cf. Ch. 4).

In collective communication primitives, message exchanges are often implemented as non-blocking send-recv pairs, respectively [98]. Although communication is non-blocking, *data dependencies* require intermediate synchronization steps. As an example, consider the binomial tree broadcast in Fig. 4.2. All ranks, excluding the root, remain idle until the first message has arrived from the respective parent. Satisfying data dependencies is necessary to ensure correctness of the algorithm. If any rank is delayed due to some kind of noise, this delay can propagate to all its children.

Another source of delay are *synchronization dependencies.* In Fig. 4.2, when rank p_4 receives a message from rank p_0, it immediately forwards this message to its children using non-blocking send operations. At some point, however, rank p_4 has to locally *synchronize* non-blocking communication. This occurs for example, if the binomial tree broadcast is additionally pipelined for large message transfers to balance the load among all ranks (cf. Sec. 4.3). In a pipelined message transmission, the root splits the message into smaller segments, sending them one after the other to its children. Similarly, intermediate nodes forward received segments to their children until reaching the leaves. To prevent network congestion, a rank does not start sending segment $k > 0$ before segment $k-1$ is successfully transmitted to all children.

Alg. 4 implements a pipelined tree broadcast, where a single message is segmented into S segments. When a rank receives a single segment from its respective parent (line 2), it immediately forwards it to its children using non-blocking sends (lines 3–5). All non-blocking operations are locally completed using a blocking `Waitall` primitive, before continuing with the next segment (line 6). If either of the children is delayed, this delay propagates to communications of the next segment. Due to the blocking nature, the synchronization dependency in the algorithm affects both the parent and other children, although these communications are independent.

This pattern is commonly used in most recent releases of MVAPICH and OMPI [65, 158]. A more efficient approach replaces the blocking `Waitall` with an event driven implementation, which integrates callbacks into the MPI progress engine to achieve fine-grained synchronization [138]. More specifically, if a request becomes ready, the registered callback immediately issues another non-blocking operation.

While non-blocking communication is more robust against noise (cf. Sec. 3.2), the combination of data and synchronization dependencies in communication algorithms can still lead to noise propagation. Heterogeneous shared memory architectures make this even more severe, as PEs with different capabilities participate in communication [138]. Hierarchical communication algorithms can mitigate the problem (cf. Sec. 4.5.1). However, communication between levels in the hierarchy is still synchronized, which hinders concurrency [9].

5.2.2 Partial Completion

We propose to allow *partial completion* of individual segments in collective communication primitives, as specified in the MPI-3 standard. A segment is defined as a sequence of data items which a rank contributes to the collective communication pattern. Therefore, each segment is defined with three parameters:

- Rank: The source *rank* within the communicator, where the collective communication is performed.
- Datatype: The datatype of a single data item.
- Count: The number of data items in the segment.

The *datatype* and *count* parameters define the message length in bytes per segment. We specify two requirements for the interface of partial completion in collective operations:

1. Partial completion should allow to immediately consume arrived segments, rather than waiting for all segments in the collective operation. Hence, parallelism in collective primitives is explicitly exposed to the user, which can operate on individual segments.

2. The interface should allow easy adoption into existing HPC codes. Therefore, we avoid modification of existing semantics for collective communication in the MPI-3 standard.

Based on these requirements we propose a `MPIX_Parrived` function call to probe for arrival of individual segments in collective communications. We explain the semantics with a non-blocking `MPI_Igather` (many-to-one) operation, as illustrated in Lst. 5.1. In this example, a root rank collects p segments into a local receive buffer, where each segment originates from a different rank. After initiating communication, the root rank enters a loop to test arrival of any segment $i \in \{0..p\}$ (line 8). Upon successful arrival of a single segment, the root breaks out of the loop to consume received data of segment i. While computation proceeds, the remaining segments will eventually arrive. Note however that arrival is not guaranteed before returning from the blocking `Wait` at the latest.

Listing 5.1: Probing partial completion of individual segments.

```c
MPI_Comm_rank(comm, &rank);
MPI_Igather(sbuf, scount, stype, rbuf, rcount, rtype, root, comm, &req);

/* Useful work */

if (rank == root) {
    for(int i = 0; i < p; i = (i+1) % p;) {
        MPIX_Parrived(&req, i, &flag); // Nonblocking test for segment i
        if (flag) break;
        // break if any segment has arrived
    }
    /* Useful work with segment i */
}

MPI_Wait(req, status); // Wait for pending segments

/* Process remaining segments */
```

We emphasize that this is a simplified example. However, it demonstrates the idea of partial completion in collective communication. Because arrived segments can be immediately consumed the receiver's virtual address space, it significantly improves interoperability in hybrid programming abstractions. Applications, using task-parallel programming models, can submit tasks on a per-segment granularity. This effectively hides communication latency of other segments in the collective communication, which are still in flight.

An advantage of the proposed interface for partial completion is that it does not change behavior of collective communication as specified in the most recent MPI-3 standard. If segments arrive out of order, this still conforms with specified rules of collective communication calls. The arrival of any segment $i > 0$ does not indicate that segments $j \in \{0..i-1\}$ have been successfully delivered. It is the user's responsibility to ensure a specific ordering. For example, a rank can poll on pending segments until a consecutive range of segments becomes available.

An extension to the proposed design is to further refine the granularity of partial completion. If a segment is composed of multiple data items (i.e. `rcount` > 1), it is possible to operate on a per data item granularity which additionally increases available overlap potential. This approach essentially combines benefits of partitioned point-to-point communication [74, 151] with partial completion in collective primitives. However, as partial completion has not been standardized at the time of writing this thesis, we cannot evaluate possible benefits in our work.

5.2.3 Non-Canonical Buffer Placement

Partial completion indicates the arrival of a given segment in a pending collective communication. This already provides a benefit in massively parallel programs, as tasks can consume data as soon as it becomes available. Yet, if an application can operate on partially received segments, the question related to canonical ordering of these segments consequently arises. More precisely, if a consuming task satisfies the concept of a *decomposable function* (Def. 2), a canonical displacement of segments is not required anymore. Instead, the receive buffer can be contiguously filled with arrived segments. This provides two advantages:

- It simplifies interaction with traditional work sharing constructs, which linearly partition the iteration space of data items. An example are parallel loops in OpenMP.
- In effect, it improves spatial and temporal locality due to a contiguous memory layout. Large vectors (receive buffers) are divided into smaller chunks, so that each chunk possibly fits into the last level cache.

As consuming tasks satisfy the concept of *decomposable functions*, the collective communication operation itself can utilize the commutative property. This may result in a more efficient communication pattern in various many-to-one and many-to-many operations. For example, a non-blocking `Gather` is often implemented using an in-order tree algorithm to retain the canonical displacement of segments in the receive buffer. The communication pattern of in-order trees heavily depends on the virtual to physical mapping from ranks to PEs [78, 144]. Specifying the commutative property in the `Gather` collective relaxes the ordering constraint and provides more flexibility to the underlying communication schedule.

Similar effects can be observed in many-to-many distributions, e.g. `Alltoall`. For relatively small to medium-sized messages, message transmission is implemented in terms of a dissemination pattern in a circular graph. The dissemination algorithm proceeds in three phases:

1. Each rank rearranges local segments.
2. Each rank performs $\lceil \log p \rceil$ communication rounds. In each round, segments are shifted along the circular graph.
3. Each rank again rearranges received segments to guarantee a canonical order.

Only phases 1 and 2 are necessary to deliver segments to their receivers. The last phase can be omitted if an application does not require a canonical order of message blocks. Omitting the last phase does not only save a potentially expensive memory copy. It eliminates an implicit synchronization point, as phase 3 cannot start before phase 2 is finished. If the number of participating ranks is large, even a logarithmic factor of communication rounds causes non-negligible overhead. This particularly manifests with growing message sizes, as each each rank transmits $\lceil \frac{p}{2} \rceil$ message blocks per round. At the same time, the number of segments arriving at their final destination doubles each round. This provides potential to consume these segments, while collective communication is still in progress.

With linear exchange algorithms, where ranks transmit local data with a direct message to their destination, partial aggregation is even more natural. The canonical ordering of segments is not a significant limitation in this case, as consuming task can immediately access any offset in the receive buffer. However, the possibility to arrange segments in arbitrary order provides more freedom in the communication itself. Efficient MPI implementations manage multiple outstanding non-blocking communication requests. The runtime continuously progresses pending requests and marks them as complete if a message has been successfully transmitted. As the arrival order of simultaneous communication requests is non-deterministic in large networks, canonical order requirements pose implicit synchronization in the communication schedule.

We propose two extensions to the existing interface of non-blocking collective communication primitives:

a. An additional parameter to specify a commutative, i.e. out-of-order, buffer placement of arrived segments in the receive buffer. This parameter hints MPI libraries, that applications relinquish the prescribed canonical placement.

b. As a consequence, users cannot deduce the source rank and displacement of a specific segment in the receive buffer. Therefore, we add another function call to probe the source and address of received segments, called `MPIX_Parrived_any`. Similar to the `MPIX_Parrived` primitives, this call is non-blocking.

Listing 5.2: Probing partial completion of individual segments.

```
   int canonical_order = 0;

   MPIX_Igather(sbuf, scount, MPI_INT, rbuf, rcount, MPI_INT, root, comm,
       &req, canonical_order /* out-of-order arrivals allowed */ );
 5 /* Useful work */

   void*    seg_address; // the address of arrived segment
   int      seg_source;  // the source rank
   while(true) {
10     // poll until first segment arrives
       MPIX_Parrived_any(&req, &seg_address, &seg_source, &flag);
       if (flag) break;
   }

15 /* Useful work on arrived segment at seg_addr */

   MPI_Wait(req, status); // Wait for pending segments
   /* Process remaining segments */
```

We illustrate the semantics in Lst. 5.2. Involved ranks issue a non-blocking gather to the root. The signature adopts the most recent MPI-3 standard, adding the last parameter. It permits out-of-order displacement of segments in the receive buffer. Similar to Lst. 5.1, the root enters a non-blocking loop to poll until arrival of at least one segment. Since the displacement of specific segments is not guaranteed due to out-of-order arrivals, we pass two additional variables to the probe function, determining the segment's source rank and address, respectively.

The signature of `MPI_Parrived_any` conforms to the existing probe functions in MPI-3. If a request has been successfully completed, it sets the flag to true, as well as the segment's address and source rank (line 11). If a segment has successfully arrived, the receiver is free to operate on its data.

Unlike *partial completion*, this change is non-trivial, as it modifies existing semantics of collective primitives. While for regular collectives, the change involves only an additional parameter, consequences are more significant with irregular collectives. In irregular collectives, the send (receive) counts and displacements are specified for each segment (rank). This results in memory overhead linear to the number of ranks, as 4 vectors are required to specify the collective communication pattern. Specifying a non-canonical order of received segments allows to discard the receive displacements vector, cutting memory overhead by a quarter. While the reduced memory overhead is beneficial in large-scale applications, the main benefit results from an increased overlap potential.

5.3 Use Cases

In this section we analyze common use cases in HPC to show the benefits of partial aggregation in collective communications:

- Collective neighborhood exchange
- Dense matrix-vector multiplication.
- Distributed sort

Compared to manual transformation techniques (cf. Sec. 3.3.2) to achieve effective communication-computation overlap, partial aggregation further improves latency hiding without significant code changes.

5.3.1 Collective Neighborhood Exchange

Collective neighborhood exchanges are ubiquitous in HPC. A common example are stencil patterns which are used to solve PDEs in high-dimensional grids. To understand optimizations through partial aggregation in collective neighborhood exchanges, we briefly summarize the general pattern of stencil computations.

Given a possibly multi-dimensional problem domain of size n which is initially decomposed and distributed to p PEs. The involved PEs are organized in a regular d-dimensional mesh. Accordingly, each PE operates on a local subgrid of size $\approx n/p$. Elements are iteratively updated with weighted contributions from their neighbors, until a certain criterion is met. The stencil shape has a radius k, defining the neighborhood of each element [80].

Fig. 5.2 illustrates the stencil pattern with two-dimensional ($d = 2$) processor grid. We visualize two common stencils, both of radius 1 ($k = 1$):

- 5-point stencil
- 9-point stencil

In the 5-point stencil, the neighborhood of each cell spans 4 neighboring cells, visualized in solid arrows along rows and columns. The 9-point stencil additionally involves 4 corners in the diagonals (dashed lines), accumulating to 8 neighbors, respectively. Note that in 9-point stencil, the corners (gray fill) need to be exchanged with 3 neighbors. Black cells need to be copied to only one neighbor.

If MPI is used as the communication runtime, neighborhood collectives provide enough flexibility to perform an efficient boundary exchange. In fact, stencil applications motivated the inclusion of neighborhood collectives in the MPI-3 standard. Based on the topology

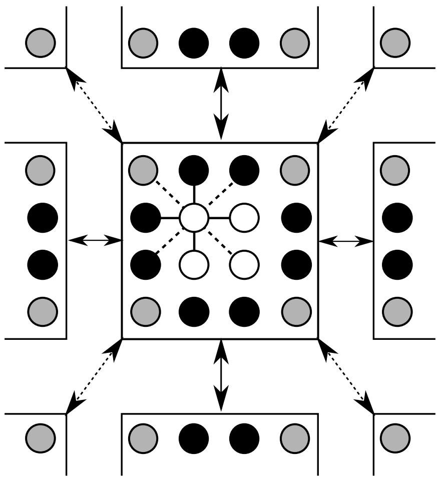


Figure 5.2: 5-point (solid) and 9-point (solid + dashed arrows) radius-1 stencil computations in distributed memory.

interface, the communication schedule can be efficiently tuned for the underlying network. To further improve message transmission, derived datatypes allow *zero-copy* exchanges. This means that communication happens in-place without any temporary communication buffers.

The most recent MPI-3 standard specifies two communication patterns for neighborhood exchanges (including irregular counterparts):

- `MPI_Neighbor_allgather`: Each process sends the same block to its neighbors.
- `MPI_Neighbor_alltoall`: Each process scatters a personalized block of data to each neighbor.

In Fig. 5.2, the neighborhood exchange corresponds to a *neighbor_alltoall* pattern, as each neighboring rank receives a personalized data block of the send buffer, i.e. boundary. Because neighborhood exchanges are significant in HPC codes, efficient algorithms are subject to active research. Notable benefits are achievable in 4-dimensional stencil computations and small message sizes. Isomorphic neighborhoods, where all ranks have the same neighborhood structure, can be additionally tuned with specific algorithms [186]. A more recent paper utilizes message combining techniques to reduce latency for small message exchanges [145]. Using neighborhood collectives is also beneficial in irregular graph problems. It has been shown that non-blocking neighborhood collectives allow efficient latency hiding in a communication-bound 2D BFS algorithm [117].

Implementing a stencil computation pattern with MPI neighborhood collectives is shown in Alg. 5. The code executes with a fixed number of iterations. Each iteration performs an n-dimensional radius-1 stencil. The neighborhood exchange is performed using a non-blocking *all-to-all* collective communication (line 3). Latency hiding is achieved by overlapping the non-blocking collective operation with updates to the *local* cells of a rank's subgrid which do not require any interaction with neighboring cells (line 4). After the local update step, the non-blocking communication call is locally completed (line 5) to subsequently update the boundary regions (line 6).

Algorithm 5: Stencil computations using non-blocking neighborhood collectives.

```
1 prepare communication buffers, MPI datatypes, etc.
2 for $i \leftarrow 0$ to niters do
3     req ← MPI_Ineighbor_alltoall(outer, neighbors)
4     Stencil(inner)
5     MPI_Wait(req)
6     Stencil(outer)
7 end
```

Reviewing Alg. 5 suggests that the best performance is achieved, if the neighborhood exchange (line 3) and stencil updates on the inner cells (line 4) take roughly the same time. However, Sec. 3.2 already discusses several challenges which complicate this approach. Workload imbalance across PEs, complex interactions in hierarchical memory subsystems, and non-deterministic scheduling in operating systems propagate as additional synchronization overhead in the message passing runtime [2].

To mitigate these issues, data needs to be consumed as soon as it becomes available, both on the sender and receiver side. It is important to note that neighboring interactions are always pairwise dependent. Although boundary regions are exchanged collectively in a rank's neighborhood, each boundary depends only on its respective neighbors.

A commonly used approach in existing HPC codes is to replace the collective primitives with a series of non-blocking point-to-point communications by following the single-sender model. While such a schedule generally works for many HPC codes, it does not benefit from existing and future improvements to neighborhood collectives.

Instead, our approach relies on the proposed interface for *partial completion* (cf. Sec. 5.2.2). As an example, in the 9-point stencil computation, each collective communication exchanges 8 segments, one per neighbor. The exact schedule how these segments are exchanged is abstracted within the MPI library. With *partial completion* we can probe for individual segments and consume them upon arrival in the receive buffer. The pseudocode in Alg. 6 shows the modified version of Alg. 5, omitting minor details. Besides the decoupling of implementation details in the communication schedule from the computation, it results in following benefits:

- Not all segments in the neighborhood exchange are of the same size. In the 9-point radius-1 stencil, corners contain only 1 data item, while edges are of size $\approx \frac{n}{p}$.
- HPC applications following a hybrid programming model using the single-sender model, multiple threads cooperatively compute in a rank's subgrid. It is unlikely that all computations take the same time among all threads, e.g. due to load imbalance, scheduling effects in the operating system, etc. Therefore, some threads may be ahead of others and finish their local stencil updates earlier. In the single-sender model, fast threads have to wait for the master thread to complete outstanding communications. With *partial completion*, each thread can probe individual segments and continue with stencil updates to the boundary region.

Algorithm 6: Collective neighborhood exchange with partial arrivals.

```
prepare communication buffers, MPI datatypes, etc.
for $i \leftarrow 0$ to niters do
    req ← MPI_Ineighbor_alltoall(outer, neighbors)
    Stencil(inner)
    while not all blocks arrived do
        flag, $k$ ← loop over neighbors using MPI_Parrived()
        if flag then
            Stencil(outer, $k$) // stencil update for block $k$
        end
    end
    MPI_Wait(req)
end
```

5.3.2 Dense Matrix-Vector Multiplication

Non-blocking collectives allow multiple collective communication operations in flight, enabling effective pipelining techniques in distributed memory. An example for collective communication pipelines is a dense matrix-vector multiplication, which is a fundamental building block in many HPC applications. Examples include finite element solvers [88] or Google's PageRank algorithm [156].

A dense matrix-vector multiply computes $y = Ax$ in parallel, where A is matrix of size $N \times N$, while x and y are vectors of size N. Given p PEs, matrix A is decomposed on a $P \times P$ processor grid. For simplicity, we assume that $P = \sqrt{p}$. Accordingly, x is partitioned into P blocks, and processors $p_{:,j}$ have a copy of the j-th block of vector x.[5] Alg. 7 outlines a possible implementation. In the first step, each process $p_{i,j}$ computes a local matrix-vector product in parallel. Local results are row-wise accumulated into y and, finally, again distributed with the same partitioning as x. We briefly analyze computational parallelism and overlap potential.

A dense matrix-vector multiply has a high degree of parallelism as the row-wise accumulation (Alg. 7, line 2) is both associative and commutative. Therefore, the operation generally satisfies the concept of a associative-decomposable function (cf. Def. 3). The *accumulate* function is implemented in terms of a reduction with the sum operator, while the *merge* function is an identity, i.e. a passthrough, function.

In Alg. 7, the communication steps in lines 2–3 are not strictly synchronizing. However, the broadcast (line 3) cannot start before partial results are completely accumulated. Several approaches suggest to transform a sequence of large collective communications into a pipeline of multiple smaller ones [94, 106]. To overlap pipelined communications, the communicator is duplicated in each pipeline stage. This approach improves parallelism and performance, if the tiling factor is well chosen for the underlying problem size and platform. The overall drawback is significantly increased code complexity. Moreover, serially arranged pipelines still impose synchronization overhead, as a pipeline stage cannot complete before the preceding stages.

[5] The colon notation specifies slices along a single dimension.

Algorithm 7: Parallel matrix-vector multiplication.

```
input  : $A, x, p$
output: $y$, distributed as $x$
1 $p_{i,j}$ computes a local matrix-vector product: $y_i^{(j)} = A_{i,j} x_j$.
2 $p_{i,:}$ reduces $y_i = \sum y_i^{(j)}$ to $p_{i,i}$ (reduction along rows).
3 $p_{i,i}$ broadcasts $y_i$ to $p_{:,i}$ (broadcast along columns).
```

Through the interface of partial completion, we can broadcast individual segments in rank $p_{i,i}$, while row-wise reduction (line 2) is still in progress. Moreover, we expose more parallelism, as the order of segment arrivals is not strictly synchronized, in contrast to serial pipelines, as suggested above. The approach is particularly beneficial with large message sizes. A commonly used implementation for the *reduce* operation with large messages is an initial *reduce-scatter* followed by an *gather*. [167]. We briefly analyze the complexity of this algorithm.

Assuming the number of ranks is a power of two and each rank contributes a vector of size n. The algorithm proceeds in two phases. Either of the two phases takes $R = \log p$ rounds:

a. Reduce-Scatter: In each round, neighboring ranks of distance $d = 2^i$, where $i \in \{0, 1, ..R-1\}$, swap the half of their memory buffers. More precisely, even/odd ranks send the second/first half of their buffer to rank $r + d/r - d$ in round i. Then, each rank accumulates received data into its local buffer by a local reduce, before continuing with the next round. In each round, the message length of scheduled send/receive pairs is halved, while the distance is doubled. This pattern is often described as *vector halving/distance doubling*. At the end of this phase, each rank has a fraction $\frac{n}{p}$ of the globally reduced vector.

b. Gather: In this phase, the scattered blocks are collected to the root rank using a *gather* operation. It essentially implements the inverse pattern of the preceding *scatter* operation.

Assuming that the gather operation is implemented using a binomial tree, the communication cost is approximated in Eqn. (4.4). With partial completion each of received segments in $\log p$ communication rounds can be independently forwarded to the subsequent *broadcast*. Therefore, two non-blocking collective communications are efficiently pipelined without any manual transformation technique.

5.3.3 Distributed Sort

As the last example, we discuss benefits of collective partial aggregation in a distributed sort algorithm. Sorting is among the most import combinatorial problems in computer science and serves as a fundamental building block in HPC applications, e.g. Big Data clustering [161] and sparse linear algebra kernels [44]. We focus on *sample sort* as it has demonstrated scalable performance in practice and, therefore, is widely adopted in HPC codes.

Algorithm 8: Parallel sample sort.

```
input  : $A$, distributed on $p$ PEs
output: $A$, globally sorted
1 samples $\leftarrow p_i$ collects local samples in block $A_i$
2 g_samples $\leftarrow$ MPI_Allgather(samples, $p$)
3 Select $p-1$ pivots from g_samples
4 Partition local block $A_i$ into $p$ pieces
5 $A_i \leftarrow$ MPI_Alltoallv($A_i$,$p$) // Route piece $A_{i,j}$ to rank $p_j$
6 Sort exchanged pieces in block $A_i$
```

Essentially, sample sort generalizes the well-known quicksort using $p-1$ pivots [27], where p is the number of PEs. The algorithm proceeds in 3 phases:

1. Pivots are selected from a sufficiently large sample of the overall input size n.
2. Based on the pivots, each rank partitions local data into p pieces, and route piece i to rank i, following an all-to-all pattern.
3. Each rank again sorts local data to obtain a globally sorted sequence.

Alg. 8 shows a possible implementation of sample sort using MPI. Samples are globally gathered to all participating ranks. Based on these samples, each rank selects $p-1$ splitters to partition local data into p pieces. After finishing the partitioning step, ranks perform an all-to-all exchange to route pieces to their respective destinations. Note that all-to-all exchange is irregular as message sizes of individual pieces may differ depending on the data distribution. In case of large data sets, the all-to-all exchange is the dominant performance bottleneck. Therefore, several approaches attempt at improve all-to-all communication.

The multi-level sample sort algorithm groups all ranks PEs into k smaller groups [67]. Accordingly, ranks agree on $k-1$ pivots, partition local data into k pieces, and route piece i to group i. Each group then recursively invokes sample sort in parallel. Compared to Alg. 8, data is moved a logarithmic number of times. While this approach increases robustness against synchronization bottlenecks in the all-to-all exchange, the workload between different groups may not be balanced. Or the workload is balanced, however, comes at the cost of an expensive pivot selection.

In a recent paper, we modified sample sort with a new pivot selection algorithm [129]. The algorithmic scheme follows the steps in Alg. 8 with two modifications:

a. Each process samples from a locally sorted range. The approach increases the work load compared to a traditional sample sort but improves the splitting and partitioning algorithms through regular sampling.

b. Because received pieces in the *all-to-all* exchange are already sorted, the final sort is replaced with a k-way merge algorithm to obtain a globally sorted sequence.

Although we could improve practical performance compared to other approaches, we have not addressed the all-to-all communication bottleneck. Improving the all-to-all exchange itself is difficult as the communication volume between ranks depends on the data distribution. However, it is possible to overlap the *all-to-all* operation and the k-way merge.

In the current MPI-3 standard, this requires either a pipeline of multiple independent *all-to-all* communications, or to replace the *all-to-all* operation with pairwise send/recv pairs [180].

We attempt at a different approach by improving latency hiding with partial aggregation. Note that the $k - way$ merge operation is both commutative and associative. That means, the order in which received segments are merged does not affect the overall result. Hence, we construct a partial aggregation tree such that the *accumulate* function is a passthrough function because there is no aggregation required. The *merge* function is defined in terms of the $k - way$ merge.

To analyze how partial aggregation improves latency hiding, we assume medium/large message sizes such that bandwidth term dominates the latency term in the communication cost model (i.e. $G * s > L$). In this case, most MPI libraries prefer direct message transmissions, where all ranks transmit their segments to the respective destinations. The total communication cost T_{a2a} for an *all-to-all* algorithm with direct transmissions can be approximated as follows:

$$T_{a2a} = (p-1)(L + 2o + sG)$$

The computational complexity for merging k sorted sequences, each of size s, is $T_{merge} = \mathcal{O}(ks)$ [41]. For latency hiding, we rely on non-blocking point-to-point communications, which can be overlapped with local work. The challenge is to determine an optimal tiling factor k which balances the communication costs of k segments and computation costs of merging k segments, respectively:

$$\begin{aligned} T_{a2a} &= T_{merge} \\ \frac{p-1}{k}(L + 2o + sG) &= \mathcal{O}(ks) \end{aligned} \tag{5.1}$$

Eqn. (5.1) states that the communication costs to send/receive k segments can be overlapped with a merge operation of k other segments, following the same approach as in the FFT analysis (cf. Sec. 3.3.2). Determining an optimal factor k is difficult as it depends not only on the machine specific LogGP parameters but on careful benchmarking of k-way merge algorithm. The performance of a k-way merge in turn depends on the number of compute threads, cache sizes, etc. However, we show in Ch. 7 that a reasonable tiling factor k achieves a significant speedup factor.

5.4 Summary

This chapter has established a formal model of partial aggregation for collective communication. Algorithms for collective communication are generally expressed as a schedule of rounds, where each round represents a set of disjoint communication pairs to transmit individual data segments. Utilizing this fact, we propose the idea of partial aggregation to operate on individual segments. If a segment has reached its destination, computation can immediately operate on this segment.

We define a set of rules based on homomorphism and decomposable functions to guarantee that operating on individual segments leads to correct results. To utilize partial aggregation in practice, we introduce two extensions to the most recent MPI standard:

1. *Partial completion* which does not modify existing collective communication semantics. Therefore, partial completion can be easily used in HPC applications, relying on MPI collectives.
2. *Non-canonical buffer placement* which addresses communication bottlenecks in tree-based communication algorithms. Tree-based algorithms transfer segments from sources to destinations in a minimum, but non-consecutive, number of rounds. If the canonical ordering is not required, partial aggregations allow to operate on non-consecutive segments.

We analyse three representative HPC use cases to demonstrate the benefits of partial aggregation in collective communication primitives. In contrast to related work, our approach exposes data parallelism in collective primitives instead of replacing them with low-level communication calls. Therefore, the presented concepts answer the following research question:

SQ3 How can partial aggregation expose the maximum degree of parallelism in collective communications?

In Ch. 6, we present FunnelMPI (FMPI) to show how presented concepts of partial aggregation can be implemented with collective communication primitives. The FMPI library is then used in Ch. 7 to assess performance benefits of our approach.

6 FunnelMPI: An Optimized Reference Implementation

This section describes FunnelMPI (FMPI), a message passing extension to implement non-blocking collective communication primitives with explicit support for partial aggregation. We propose a high-quality implementation to maximize performance efficiency for asymptotically optimal communication algorithms.

First, we present a lightweight implementation to guarantee independent progress in collective communication primitives. Independent progress, as described in Sec. 3.2.4, is necessary to take advantage of partial aggregation. Second, we propose an intuitive interface to express partial aggregation in collective communication primitives. The interface allows to specify the degree of parallelism through associative-decomposable functions (cf. Def. 3), along with compute tasks to operate on respective message buffers.

The overall design goal of FMPI is smooth integration with asynchronous programming models in HPC. The presented implementation serves as a proof-of-concept to evaluate the performance benefits of partial aggregation in Ch. 7. Hence, this chapter contributes to the solution for research question *SQ4*:

SQ4 How does partial aggregation improve performance efficiency in collective communications?

6.1 Design Space

FMPI follows the design of the libNBC library, which has been introduced as a reference implementation to support non-blocking collectives in MPI-3 [98]. As the libNBC implementation has been adopted in commonly used MPI libraries, e.g. OpenMPI and MPICH, we can build on experiences of prior work to provide a high-quality implementation. Because FMPI relies only on standardized MPI communication protocols, it can be easily integrated into existing HPC codes.

We formulate the following design goals:

- Non-blocking collective primitives initiate collective operations and return immediately with minimal overhead.
- Request handles to identify the state of pending collective operations are needed.
- Testing the state of a request handle must be non-blocking and specific to a single request handle.
- Waiting for a request handle must be blocking, unless the collective operation is already complete.

- Communication contexts limit the scope of collective communication similar to MPI communicators.
- Message tags distinguish communication operations in multiple concurrent collective operations.

6.1.1 Collective Schedules

A collective *schedule* is a personalized execution plan to process a collective operation in a group of p ranks. Schedules are an ordered set of communication *rounds* to model data dependencies. The ordering of communication rounds guarantees that actions in round r cannot finish before any action in round $r' < r$. Each round, in turn, is an unordered set of *actions* to resolve modeled data dependencies. Actions can be realized through point-to-point messages, one-sided put/get operations, or fast `memcpy` operations if source and destination ranks have access to a shared address space. Therefore, an action stores all necessary information to perform communication operations, including the source and destination ranks, message buffers, message tags, etc.

We formalize the concept of a collective schedule as follows:

Definition 4 (Collective Schedule)**.** *A collective schedule consists of $R > 0$ rounds and $A \geq 0$ actions:*

- *Rounds are strictly ordered, such that round r cannot execute before round $r - 1$. Each round groups a set of independent actions.*
- *Actions resolve data dependencies through communication operations to progress a collective schedule.*

Deriving the number of rounds for rank i depends on the specific algorithm (e.g. regular tree), the rank i itself, and the size of a communicator. We illustrate the overall concept with a binary broadcast tree, as visualized in Fig. 4.1. In a non-pipelined broadcast, the collective schedule of inner nodes (e.g. node p_2) consists of two rounds. The first round receives data from parent nodes, while the second round forwards it to respective children.

This concept is generic enough to model data dependencies in arbitrary collective communication patterns, e.g. trees, circulant graphs, rings, and hybrids of these concepts (cf. Ch. 4). Below, we show that optimizations like pipelining and flow control can be applied as well.

6.1.2 Executing a Collective Schedule

The design of FMPI allows multiple concurrent executions of a collective schedule. Similar to MPI, we identify an instance of a collective schedule in a request handle. A request handle encapsulates the communicator, schedule and necessary state to progress communications. To avoid tag collisions with user communicators, they are shadowed upon first use. Additionally, a unique tag is assigned to each request, distinguishing operations in multiple outstanding, i.e. concurrent, collective communication operations.

All collective operations in our library execute on the shadowed communicator instance. Besides conflict avoidance, this enables optimizations within the library. As an example, MPI allows communicator-specific hints for send/receive operations to relax serial message matching semantics. We utilize these hints to improve parallelism in collective communication primitives in our shadow communicators.

Putting all together, a collective schedules executes in three steps:

1. Allocate a request handle including necessary state.
2. Perform actions in the first round of the collective schedule.
3. Progress actions and continue to the next round, until all rounds have been successfully completed.

The issue with a round-based design is that it introduces pseudo-synchronization between ranks, as round r cannot start before round $r-1$ is complete. Sec. 5.2.1 describes the resulting effects in more detail. Such a design is unproblematic, if the average number of rounds per rank and transmitted message sizes are relatively small. However, high-quality implementations, e.g. MVAPICH or OpenMPI, incorporate flow control mechanisms to prevent network congestion with large message sizes.

As an example, segmentation of large messages into S smaller packets increases the number of rounds by a factor of S. To prevent unnecessary synchronization due to segmentation, we propose an event-driven design of collective schedules.

6.1.3 Data Transfer and Flow Control

FMPI facilitates an *event-driven* design through signals and callbacks. A signal notifies about a certain event due to progress of actions, e.g. completing a non-blocking message receive. Signals are associated with a list of callbacks, specifying how to react on the event. To illustrate the principle, consider a pipelined non-personal broadcast, where the root rank segments the message into S blocks. In a binomial tree, the root transmits the first segment, s_0, to $\lceil \log p \rceil$ children using non-blocking sends (cf. Fig. 4.2). When either of these send requests completes, the next available segment s_1 is immediately transmitted, independent of the progression of other send operations.

This callback-driven design further enables flow control mechanisms to pipeline transmission of segments. Sending individual segments of a single message one after the other, and in-order, might not utilize available network or channel bandwidths. Therefore, issuing multiple segments to each child results in a sliding window to concurrently progress actions of inter-dependent rounds. Sliding windows are a ubiquitous control flow concept [182]. In the scope of HPC applications, Sec. 3.3 introduces sliding windows to pipeline multiple concurrent collective communications. In the context of collective schedules, sliding windows manage a correct order of actions to progress a sequence of rounds.

A sliding window has a capacity for N send operations and M receive operations for each message pair, i.e. $N + M$ actions in total. The root PE posts N sends to each child, transmitting the first N segments, respectively. Upon completion of any send request, it immediately transmits the next segment. Accordingly, children issue M receive operations, matching the first M segments. When either of these M outstanding segments arrives, it is immediately forwarded to respective children, if they exist. Moreover, for each arrived segment, PEs post another receive operation to occupy the window slot for the next outstanding segment from the parent.

The rational behind a different capacity for sends and receives in a sliding window is the behavior of message queues in MPI libraries. If a segment enters a specific endpoint through the NIC, before the application has posted a matching receive operation, the segment is considered as *unexpected* (cf.@sec:characterizing-latency). Taking this into account, we set $M > N$ to minimize the *unexpected* message queues (UQ).

Sliding windows increase robustness against noise compared to a simple round-based synchronization scheme. In other MPI libraries, a rank always waits until a segment has reached all its peers in the communication tree, which is prone to noise propagation. With the proposed callback-driven design and sliding window techniques, delays can be absorbed more efficient as each action is synchronized independently of other actions.

6.2 Independent Progress Engine

After presenting essential concepts and data structures to implement collective schedules, we tackle the problem of independent progress. Our approach consists of two components:

1. Dedicated progression threads to perform communication asynchronously.
2. Signal and callback handlers as an interface to exchange control information between the application and the FMPI progress engine.

6.2.1 Communication Threads

Upon initialization of the MPI runtime, each rank spawns an additional thread to progress collective communication operations on behalf of the user threads. As we note in Sec. 3.2.4, spawning one thread per MPI rank can be inefficient, as it occupies available compute cores. However, hybrid applications commonly run only few MPI processes per node, while the majority of cores is occupied by thread-based work sharing. Moreover, as many applications have an inherently low operational intensity, dedicating some compute cores to communication progression does often not impact computational throughput. We elaborate this more detailed in Ch. 7.

This design obviously requires to exchange control information between the progression thread and main threads. This control information is encapsulated in `FMPI_Request` handles, wrapping an `MPI_Request` and additional implementation-specific details, as described in an earlier section. When the application issues a non-blocking collective communication, the progress thread wakes up to prepare the request handle, and immediately returns it to the application. Thus, preparation overhead is reduced to a minimum, allowing the application to immediately advance computation.

FMPI adopts the MPI interface, providing `FMPI_Wait` and `FMPI_Test` as blocking and non-blocking completion primitives for `FMPI_Request` handles. To provide partial completion capabilities, our library provides `FMPI_Parrived` following the same interface, as described in Sec. 5.2.

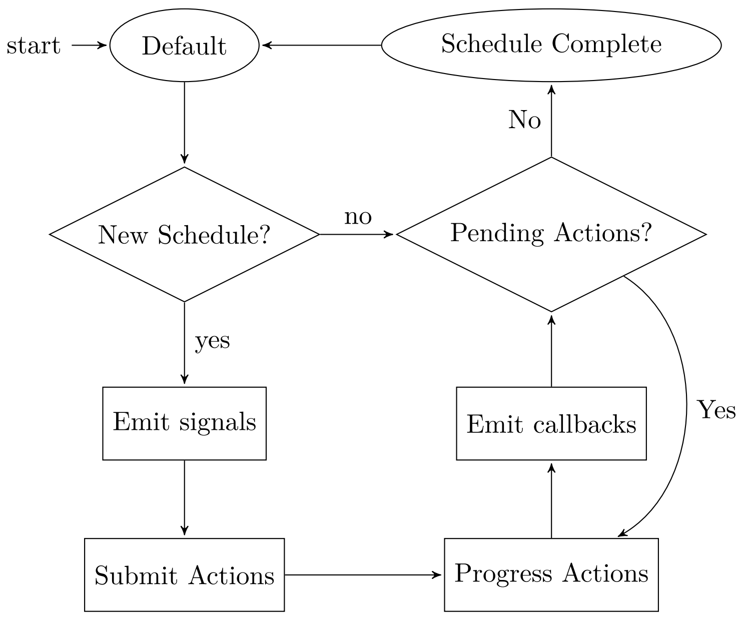


Figure 6.1: State Diagram of FMPI Progress Engine.

To handle multiple outstanding communication requests, progress threads are equipped with a lock-free producer-consumer request queue (RQ) to track outstanding collective communication requests, i.e. *instances* of collective schedules [86]. Each instance has a unique connection to the respective `FMPI_Request` to notify the application about its progress state. The progress thread pulls communication schedules from the RQ and issues associated actions on its internal progress channel (PC). The progress channel is implemented as a first-in-first-out (FIFO) queue and has a fixed capacity, limiting the number of simultaneous actions to be processed. Therefore, a progress channel essentially implements a sliding window mechanism to minimize message queue overhead in MPI libraries. Data items on the RQ and PQ are fixed size objects of 128 bytes and aligned to cache line boundaries, which prevents false sharing effects in caches between user threads and the progress thread.

An interesting question is the placement of progression threads. Taking related work into account, it is most beneficial to place auxiliary progression threads close to the application threads [48, 96]. Therefore, FMPI progression threads are pinned on the same NUMA domain as the respective MPI rank, i.e. the main thread. This reduces NUMA effects, and increases benefits from shared last level caches between user and progress threads.

6.2.2 Signals and Callbacks

To implement event-driven behavior, a progress thread features two additional lists for signals and callbacks specific to each even type, e.g. send or receive. A signal fires, when a communication action enters the progress channel. Thus, signals can be seen as a preparation step in the progress thread, before associated communication operations are issued to the MPI library.

A common use case for signals is to allocate temporary message buffers in tree-based communication algorithms. We utilize this approach to postpone expensive memory allocation until really needed.

Callbacks, on the other hand, fire when an action is ready, i.e. the associated communication request is marked as complete due to progressing pending actions. Hence, callbacks can be used to progress dependent communication operations in collective communications, reclaiming memory of temporary message buffers, etc.

The order in which signals and callbacks are registered to the progress channel, determines the execution order when associated events fire. Deterministic execution allows to register a pipeline of very small callback functions, which can be even modified at runtime.

The state diagram in Fig. 6.1 visualizes the process of the FMPI progress engine. The progress thread initially moves into the default state to wait for new collective schedules issued by the application. When a collective schedule arrives, it is assembled into rounds of actions. After emitting registered signals for actions in the first round, associated communication requests are submitted to the MPI library. In the next step, the communication thread probes the MPI library to progress the submitted communication requests. If either of these requests have been completed, registered callbacks trigger and eventually replace free slots with new outstanding actions to progress the collective schedule. For the final step, there are two alternatives. If the progress channel has still pending requests, the communication remains in the progress loop. Otherwise, the collective schedule is marked as complete, and the progress threads transitions back into the default state.

There are two edge cases to consider:

a. If both the request queue (RQ) and the progress queue (PQ) are empty, the communication thread remains in an idle default state. A different approach would be to continuously poll the task queue, which is utilized in other works for similar purposes [118, 125]. However, waking up progress threads only upon request releases PEs for application threads, if there is no pending collective communication.

b. If the task queue is not empty and the progress channel is fully occupied, tasks remain in the queue until a single slot becomes free. As task queues are implemented in a thread-safe manner, it is possible to spawn multiple communication threads per rank. However, evaluated use cases in Ch. 7 have shown that one communication thread per rank suffices to process the workload, which confirms results in related work [48, 118].

Therefore, the progress engine is designed to progress pending collective schedules as fast as possible, rather than processing many collective schedules in parallel. The approach minimizes contention on the critical path, following similar principles as discussed in related work [191].

6.3 Structured Concurrency in Collective Communications

One of the main goals for partial aggregation in collective communication is to improve interoperability between message passing and thread-level parallelism. Therefore, we provide a task-based interface, which is based on recent C++ concurrency features.

The aim is to mitigate any coarse-grained synchronization, which is still common in established programming. We show in the following sections, how partial aggregation supports fine-grained parallelism in collective communication primitives through the following concepts:

- *Futures* which we define as control objects to pass asynchronous state between HPC applications and the FMPI library.
- Synchronization primitives to integrate partial aggregation on *future* objects.

6.3.1 Futures and Continuations

Besides an asynchronous progress engine to offload expensive collective communication, FMPI provides local control objects to express parallelism and synchronization through data dependencies. Based on data dependencies, latency hiding is implicitly implemented through the event-driven programming interface, as described in Sec. 6.1.3. The most fundamental local control object is a future [81].

A *future* is a control object which semantically decouples the result of a function, i.e. a task, from its execution. The task can be executed in a different context, and the result is accessible from the *future* object.

Supported execution policies adopt and extend standardized functionality in recent C++ standards and template libraries:

Synchronous exection policy This is the most common way to execute a task. The caller schedules a task and immediately waits for successful return. In MPI terminology, this means that the task runs in a blocking fashion.

Asnchronous exection policy The execution of a task is scheduled on a implementation-defined context to execute the task on behalf of the caller. Similar to a non-blocking MPI call, a future is immediately returned and the caller can proceed with other work.

Fire and forget execution policy Similar to asynchronous execution, a task is scheduled on a different context without waiting for any result. However, there is no notification to the caller upon task completion. If a result is not needed, this reduces costly synchronization between different execution contexts.

An alternative to futures are callback functions, which are often used in parallel programming abstractions. Similar to FMPI, tasks are executed in abstracted execution contexts within parallel programming libraries. Whenever a task finishes, the callback is triggered to communicate the result to the caller [87, 138]. The drawback of this approach is that the callback needs to be available in advance, before a task is scheduled.

Instead, a future allows to attach a *continuation*, which provides a similar interface like traditional callbacks. The main benefit is that the callback function can be attached to an already submitted *future* object. This concept makes futures easy to compose and integrate with other programming models. However, the flexibility comes at the cost of an atomic operation to determine the state of the future, i.e. whether the result is available or pending. However, it provides enough flexibility for latency hiding, as all attached continuations but the last one can run asynchronously in FMPI execution contexts.

Listing 6.1: Collective communication with FMPI futures.

```
   // Some work to operate on message buffers
   void do_work(int* recvbuf, std::size_t n);
   void do_other_work(int* recvbuf, std::size_t n);

 5 void do_comm() {
     // common variables
     int* sendbuf;
     int* recvbuf;
     std::size_t n;
10
     // Signature of FMPI collectives is identical with the MPI standard.
     auto future = fmpi::alltoall(sendbuf, n, ...,recvbuf, n, ...);
     future.then(do_work).then(do_other_work).wait();
   }
```

Lst. 6.1 sketches a simple example of the proposed interface. Essentially, the FMPI interface follows the same conventions as the MPI standard, except that `MPI_Request` handles are replaced with FMPI futures. FMPI futures can be moved around between different scopes, and whenever the result is needed, a blocking wait can be performed. To prevent dangling communication requests, a future always blocks when its lifetime ends unless the result has already been consumed.

Lst. 6.1 also shows, how continuations can be attached to a future with the `then` operator (line 13). The execution of continuations synchronizes with a future's readiness. In this example, the `do_work` routine is called upon successfully completing the all-to-all communication. Since the `then` operator returns another future, we can chain multiple continuations into a single sequence. Each continuation in the chain synchronizes with the preceding operation.

The challenge is that one cannot easily determine, in what execution context the continuation will run. This can lead to surprises for experienced HPC programmers, as many optimizations usually require explicit control about the execution context. Therefore, it is crucial to select the correct execution policy described above.

6.3.2 Synchronization Primitives for Partial Aggregation

Futures encapsulate the execution context of collective communication primitives. To perform tasks on individual segments of collective communications, we provide additional synchronization primitives:

- `when_any_partial`: Creates a *future* object that becomes ready when a single segment in a collective communication has been successfully received. To identify the received segment, the result of the future is a tuple, storing the segment index and a pointer to the first byte of the segment data.
- `when_some_partial`: Creates a *future* object that becomes ready when *at least* one segment in a collective communication becomes ready. The return value of the future is a list of segments, each containing the segment index and a pointer to the receive data.
- `while_some_partial`: Similar to `when_some_partial`, it creates a *future* object that triggers when *at least* one segment in the associated collective operation becomes ready. However, the attached continuation is called repeatedly until the collective operation completes.

We illustrate the functionality with a distributed sort algorithm as shown in Alg. 8 [129]. The last step involves is an all-to-all exchange, where processors route individual pieces to their destinations. After performing all-to-all communication, each processor locally merges received pieces to obtain a globally sorted sequence. Since the merge operation satisfies the concept of associative-decomposable functions (cf. Def. 3), it is possible to merge received sequences in any order.

Listing 6.2: Asynchronous k-way merge with partial aggregation.

```
    void parallel_multiway_merge(std::vector<segments>) {
        // fast parallel k-way merge algorithm in shared memory
        // can be implemented in any multi-threaded abstraction (e.g., OpenMP)
    }
 5
    void distributed_sort() {
      // sampling, splitting, and pivot selection...

      auto future = fmpi::alltoall(sendbuf, n, ...,recvbuf, n, ...);
10
      while_some_partial(future).then([](std::vector<segment> segments) {
          // merge received segment in parallel
          parallel_multiway_merge(segments);
      });
15  }
```

Utilizing the interface for partial aggregation enables to overlap the collective all-to-all communication with the local merge, as shown in Lst. 6.2. Whenever at least one segment arrives at the caller's receive buffer, the attached continuation is invoked to merge the list of received segments. We express this using the `while_some_partial` routine (line 11) which triggers the associated continuation upon receiving an arbitrary segment. The continuation, in turn, merges the partially received segments using a fast parallel k-way merge algorithm (line 13). This process repeats until the collective communication has been locally completed.

6.4 Summary

This chapter describes FMPI which is a proof-of-concept to integrate partial aggregation into collective communication primitives. The formulated key requirements are listed as follows:

- Maximum performance benefits through partial aggregation in asynchronous collective communication.
- Smooth integration into existing scientific algorithms.

A prerequisite for the first goal is an asynchronous progress engine. Therefore, we implement a communication offload model to perform collective communication on a dedicated progress thread. Minimum synchronization overhead between user threads and progress threads is achieved using lock-free data structures. To express partial aggregation, we rely on standardized C++ future concepts due to its wide adoption in multi-threaded programming models for shared memory.

Utilizing MPI as the underlying message passing layer and standardized C++ concurrency features ensures that our prototype can be used on essentially on any HPC system. It further ensures that presented results in Ch. 7 can be compared with state-of-the-art MPI implementations which support collective communication primitives.

7 Evaluation

We empirically evaluate our concepts of partial aggregation in Ch. 5, based on a prototypical implementation in FMPI. The evaluation studies the performance benefits of partial aggregation in collective communication with the FMPI library and, therefore, addresses the following research question:

SQ4 How does partial aggregation improve performance efficiency in collective communications?

We systematically answer this question in two steps:

1. We conduct a set of synthetic microbenchmarks to compare the performance of collective communication primitives implemented in FMPI versus an optimized MPI library which is commonly used in practice.
2. We assess possible performance improvements with partial aggregation in collective communications by integrating the FMPI library into existing HPC applications:
    - A large-scale distributed sort, where large data volumes are transferred across the network. We show, how partial aggregation can hide high communication costs in a dense collective all-to-all communication.
    - A heat equation solver, implemented as an iterative Jacobi solver. The application requires neighborhood communication to exchange the boundary conditions in each iteration. We show, how partial aggregation maximizes latency hiding through overlapping local computation with a sparse collective neighborhood all-to-all operation.

7.1 Platform Description

The experiments are conducted on the supercomputer SuperMUC-NG , hosted at the Leibniz Rechenzentrum (LRZ). At the time of writing this thesis, the system is ranked on position 15 at the Top500 list, achieving a peak performance of approximately 26.8 Petaflops[6] [181].

SuperMUC-NG interconnects 6336 compute nodes in a non-blocking 100 Gbit Intel Omni-Path network fabric. Each node provides dual-socket Intel Xeon Platinum 8174 processors, with a clock base frequency of 3.1 GHz and a maximum turbo frequency of 3.1 GHz. The socket of an 8174 Platinum Skylake processor contains 24 cores with 2-way Simultaneous Multi-threading (SMT) enabled per core, which accumulates to 96 hardware threads per

[6] 1 Petaflop $\approx 10^{15}$ floating point operations per second.

Table 7.1: Fact Sheet for a single node in SuperMUC-NG.

CPU	2 × Intel Skylake Xeon Platinum 8174
Cores	48 (96 hardware threads)
Cache	L1 32 kB (instruction + data) L2 1024 kB L3 33 MB (shared per socket)
Memory	96 GB
Network	Intel Omni-Path fabric (100 Gbit Ethernet)

node. 96 GB of host memory is provided for both sockets (48 GB per socket, spread across six memory channels). Processor cores have access to a private L1 and L2 data caches of 32 kB and 1 MB, respectively. Each socket provides a distributed L3 cache of 3 MB across the 2D processor mesh. Tbl. 7.1 summarizes relevant features for reference in the experiments.

For all experiments reported below, we use the Intel 19.0.5 with GCC 9.2 compatibility enabled. All applications are built with an ISA optimized build of Intel MPI 2019. The operating system is SUSE Linux Enterprise Server 12.3.

7.2 Microbenchmarks

To assess the performance of the FMPI communication library, we conduct a series of microbenchmarks, based on a subset of the Ohio State University (OSU) microbenchmark suite[7]. These benchmarks are synthetic, and provide only limited insights for real-world application performance. However, they provide an effective approach to measure performance in extreme cases. The main purpose here is to obtain a performance profile of FMPI, compared to an optimized reference implementation on the SuperMUC-NG system.

The series of microbenchmarks is divided into two parts:

- Multi-threaded ping pong to measure the uni-directional *latency* cost between multiple pairs of senders and receivers in multi-threaded applications.
- Aggregated multi-pair communication throughput to measure the maximum attainable communication *bandwidth*.

7.2.1 Latency

The multi-threaded latency benchmark measures the uni-directional latency, where 2 MPI processes spawn multiple threads to communicate back and forth repeatedly, using blocking send/receive operations. Both MPI processes reside on two different nodes to guarantee that the network fabric is involved. Since FMPI supports only non-blocking communication, a single roundtrip is implemented as a sequence of two non-blocking send/receive calls, followed by a blocking wait to complete pending communication calls.

[7]http://mvapich.cse.ohio-state.edu/benchmarks/

7.2.1.1 Benchmark Setup

During the multi-threaded latency experiments, we have observed large performance variabilities. Therefore, each experiment repeatedly executed at least 20, and up to 200 warmup iterations, until the standard deviation was bounded within 5% of the arithmetic mean. To obtain reliable results, we report the arithmetic mean of another 10 000 iterations for small message sizes ($\leq 4\,\text{kB}$), and 1000 iterations for larger message sizes.

We have studied four configurations, each with an equal number of senders and receivers, ranging from 1 to 8 threads. Each configuration has been executed with different message sizes, ranging from 0 bytes to 32 kB.

7.2.1.2 Results

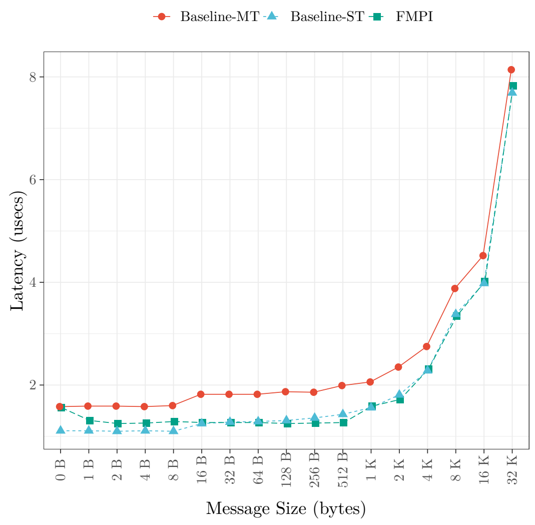
Fig. 7.1 reports measured latency costs for four configurations. In each plot, the x-axis shows an increasing scale of message sizes in bytes. The y-axis visualizes the latency cost in microseconds (usecs).

Fig. 7.1a visualizes the latency overhead of FMPI and the multi-threaded baseline (Baseline-MT), compared to a single-threaded MPI configuration (Baseline-ST). For very small messages ($\leq 16\,\text{B}$), FMPI comes with an additional latency cost of $\approx 0.3\,\mu\text{s}$. This is reasonable due to the offload detour through the communication thread. Each message is first encapsulated as a single action to schedule for the communication thread which executes the action on behalf of the user thread. When the action is complete, the user thread is notified to obtain the ready state.

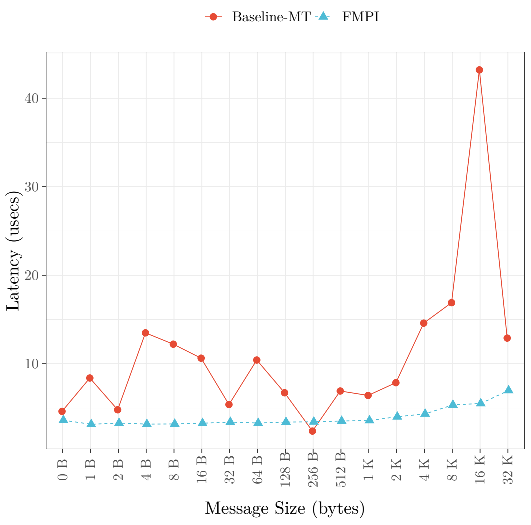
FMPI consistently outperforms the multi-threaded baseline version with little and medium-sized messages. This is expected, because FMPI does not incur synchronization overhead within the MPI runtime. The performance improvements almost vanish with very large messages ($> 16\,\text{kB}$), because large messages are affected more by bandwidth than latency. Another factor is the switch from an *eager* to a *rendevouz* protocol, which is exactly 64 kB in the default settings of the Intel MPI library. We have not modified this parameter in this benchmark.

Figs. 7.1b–7.1d show further performance improvements with FMPI for 2, and up to 8 threads per MPI process, compared to a multi-threaded MPI runtime. While latency costs for small message sizes do not significantly differ, the improvements are significant with large message sizes.

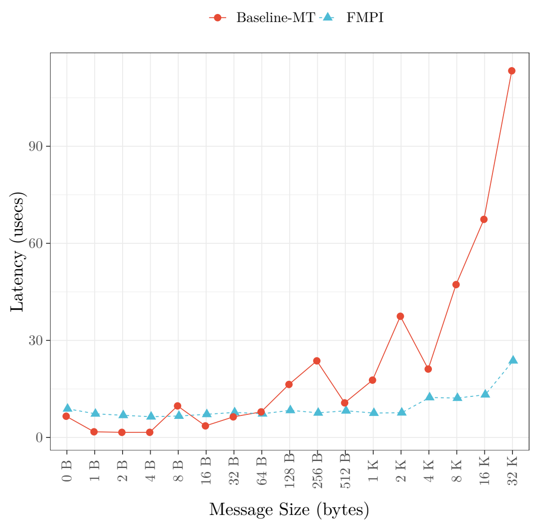
Note that the performance degradation becomes worse with an increasing number of threads and large message sizes. Furthermore, while FMPI achieves relatively stable performance results, the multi-threaded baseline suffers from non-deterministic scheduling effects in the operating system. Because send/receive operations are always blocking, the MPI runtime cannot serve other threads before a pending message has been completely transmitted. More specifically, if a MPI process receives an unexpected request for a blocking send (receive) operation from a particular thread, it needs to wait until the corresponding thread posts a matching receive (send) operation. Although multiplexing blocking sends through message tags is possible in the MPI communication standard, progress often depends on the operating system scheduler. These effects are particularly observable with large message



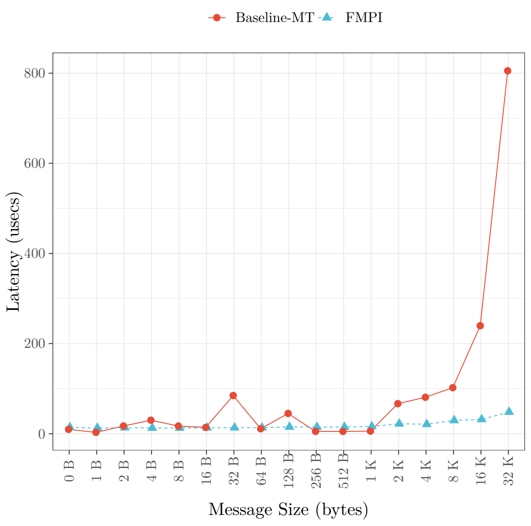
(a) 1 thread.



(b) 2 threads.



(c) 4 threads.



(d) 8 threads.

Figure 7.1: Multi-threaded latency benchmarks between two nodes. Threads on one node send, while threads on the other node receive messages.

sizes, as the MPI library switches from an *eager* to a *rendevouz* protocol, where an initial handshake synchronizes communication peers before each message transmission.

Overall performance results can be attributed to two reasons:

- The offload mechanism dedicates a single thread to process communication, which prevents multi-threaded contention within the communication runtime.
- Communication progress in FMPI is not as sensible to operating system scheduling effects. Because the offload thread is pinned to a dedicated core, preemption does not interfere with communication progress in the communication runtime. As a result, FMPI is more robust in multi-threaded applications compared to the Intel MPI library.

7.2.2 Bandwidth

To assess the overhead on communication bandwidth, we perform a multi-pair bandwidth test which is included in the OSU benchmark suite. This test measures the aggregate communication throughput between multiple pairs of senders and receivers, distributed among a configurable number of nodes.

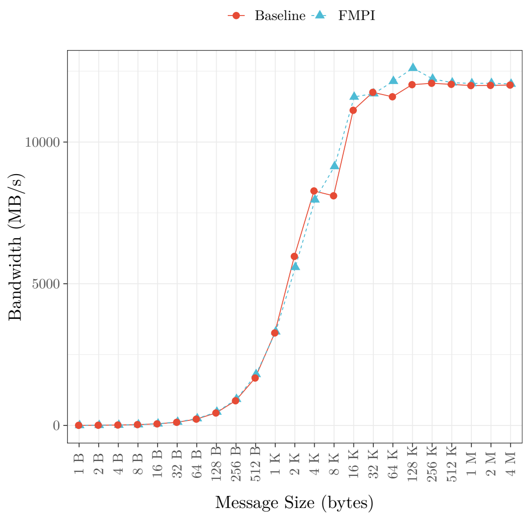
7.2.2.1 Benchmark Setup

As we are interested in attaining the maximum throughput, each sender issues a bulk of non-blocking send operations to the paired receiver, and waits for the receiver to acknowledge successful transmission. Accordingly, the receiving processes issue a bulk of matching non-blocking receives. Successful transmission of all outstanding operations is acknowledged with a single blocking send to the paired sender. The overall procedure is repeated multiple times to report the arithmetic mean in megabytes per second (MB/s). We used two different configurations with 2 and 16 nodes. Due to the dual-socket architecture on SuperMUC-NG, each node runs two MPI processes.

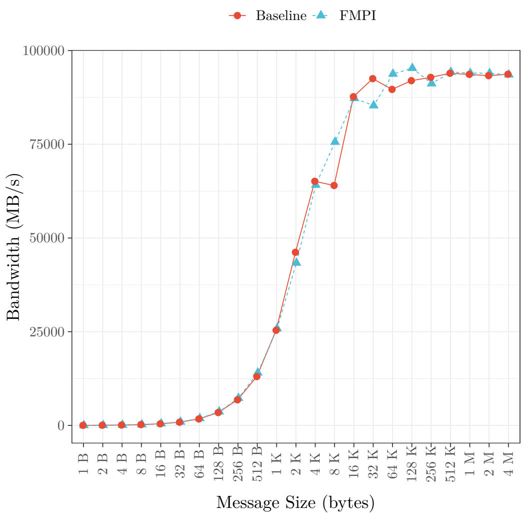
The tuning parameter in this benchmark is the window size, i.e. the number of messages per bulk. We empirically studied the best configuration for the baseline implementation, starting from a window size of 2, and up to 128 messages. The best performance has been achieved with a window size of 64 send/recv requests. With this result in mind, we performed the bandwidth benchmark to evaluate the maximum attainable bandwidth with both FMPI and the baseline implementation. The window size is set to 64 requests in both variants. The communication pairs are constructed by bisecting the range of ranks in `MPI_COMM_WORLD` into two halves. Senders reside in the lower half and receivers in the upper half. This scheme maximizes the distance of packets between senders and receivers, which is a reasonable estimation for communication bandwidth of a particular system.

7.2.2.2 Results

The results are reported in Fig. 7.2. In each plot, the x-axis shows evaluated message sizes in bytes for each send/receive request, while the y-axis visualizes the achieved bandwidth in megabytes per second (MB/s). All plots consistently show that the offloading overhead



(a) 2 pairs (2 nodes).



(b) 16 pairs (16 nodes).

Figure 7.2: Multi-node bandwidth benchmarks, where each node runs two MPI ranks.

in FMPI does not impact overall performance efficiency, both with 2 and 16 nodes. While there is a negligible penalty with very small messages up to 8 bytes, the overall performance characteristics of FMPI match with the Baseline implementation.

We conclude this section with the observation that communication offloading results in reduced latency costs due to multi-threaded synchronization bottlenecks in even efficient MPI libraries. The effect is significant with an increasing number of threads and large message sizes, which are both common characteristics in efficiently parallelized applications.

7.3 All-to-all Collective Communication Benchmark

Performance efficiency in collective communication is a prerequisite to obtain benefits through partial aggregation. In this chapter, we conduct a detailed evaluation of the all-to-all collective in FMPI by comparing it with the Intel MPI implementation on SuperMUC-NG.

Note that FMPI provides the dense all-gather and sparse neighborhood all-to-all primitives as well. The all-gather pattern is very similar to all-to-all as it is only the non-personalized counterpart. Because all-to-all is considered as the most general and also least scalable communication pattern in MPI, the following evaluation provides also interesting insights for other collective communication primitives.

7.3.1 Methodology

The performance study follows the methodology in the OSU collective benchmark suite. In this benchmark, p ranks in a given communicator allocate a contiguous block of n bytes, which accumulates to $p \times n$ bytes in total. After filling the memory buffer with random values, ranks exchange local data according to the collective communication pattern. To provide stable results, each benchmark is repeated for multiple iterations, and we report the arithmetic mean as the latency cost for the respective collective communication. Before the timing starts, 100 warmup iterations are performed.

Besides the overall communication latency to complete a collective communication, it is interesting whether an MPI process can perform computation, while collective communication is in progress. Therefore, we focus only on non-blocking communication calls, which are divided into two steps:

- Initialization time: The time to initiate the non-blocking communication.
- Wait time: The time to locally complete the collective communication.

With optimal latency hiding, the initiating rank can perform computation, while communication asynchronously progresses in the background. If asynchronous progress capabilities are not available, progress is delayed until the previously issued communication request is explicitly completed, effectively turning the non-blocking into a blocking communication call.

To assess the overlap ratio, the benchmark is performed in two parts:

1. We first measure the raw latency t_c for a *blocking* collective communication call.
2. It follows the *non-blocking* benchmark, consisting of the following steps:

   a. Involved ranks initiate the collective communication call.

   b. For a period of $t_w \approx t_c$, all ranks perform some computation to simulate nearly perfect overlap. The computation performs a matrix-multiply of small dimensions, where the buffer size accumulates to $\approx 5\,\text{MB}$.

   c. The non-blocking communication request is locally completed with a blocking wait call.

The values for t_c and t_w are reported along with total execution time T which includes all steps (a–c). Therefore, the overlap ratio r can be calculated as follows:

$$r = \frac{t_c + t_w - T}{\max(t_c, t_w)} \tag{7.1}$$

Eqn. (7.1) states that in a good overlap ratio, T is close to the maximum of t_c and t_w. Otherwise, T is close to the sum $t_w + t_c$.

7.3.2 Implemented Algorithms

Ch. 4 suggests two approaches to implement line all-to-all algorithms:

- The first it to arrange all ranks in a linear ring (cf. Fig. 4.3). Given p processors, each processor executes a schedule of $p - 1$ rounds, where each round is a one-to-one mapping of message pairs.

- An alternative are circulant graphs, where ranks shuffle all segments in intermediate steps to perform an all-to-all pattern. An asymptotically optimal algorithm which implements this approach is the Bruck algorithm [32].

Since linear rings are commonly used with large message sizes, it is crucial to minimize congestion bottlenecks in the network. We implemented three techniques in FMPI to achieve this:

- Ring-Scatter: In round j, rank i sends to $(i+j) \bmod p$ and receives from $(i-j) \bmod p$. Although message transmissions are scattered throughout the network, it is not guaranteed to be congestion-free. We list this algorithm as the *ring* algorithm in the remainder of this section.
- One-Factor: Another algorithm, which we have implemented, is based on one-factorization in graphs [172]. If p is odd, the partner of rank i in round j is $(i - j) \bmod p$. Therefore, one rank is idle in each round. If p is even, this idle processor communicates with rank $p - 1$.
- Hypercube: If p is a power of two, the hypercube pattern guarantees a congestion-free schedule. A message between two ranks i and j travels across l links, where l is hamming distance of their respective binary representation. Sorting these links according to their dimension in increasing order guarantees a unique path for each message pair in the hypercube. Determining the partner requires only a single exclusive-or operation $i \oplus j$ [73].

For the *ring-scatter* and *one-factor* algorithms, we support a sliding window mechanism to pipeline multiple rounds in a single batch. Hence, we issue communication requests for multiple rounds. If the window capacity is fully exhausted, pending communication requests are completed with a single `waitall`.

The communication schedule of the Bruck algorithm is already discussed in Sec. 4.4. While this algorithm is asymptotically optimal, the communication pattern of the Bruck algorithm is inherently non-contiguous. The data elements that are sent in one communication round have been received in previous rounds. Therefore, elements will either have to be communicated directly from/to non-contiguous segments of memory, or must be reorganized locally into contiguous communication buffers, often called packing. In FMPI, we implement this with derived MPI data types to support zero-copy message transmission in hardware [188].

Tbl. 7.2 summarizes relevant properties for each algorithm (first column). The second column is the number of communication rounds to complete the schedule. The third column lists the source (*src*) and destination (*dst*) of a specific rank i in round j, depending on the number of rounds r. If not explicitly specified, source and destinations are the same. The last column defines, whether message buffers are contiguous, which is not the case only in the Bruck algorithm.

7.3.3 Performance Results

All implemented listed above are evaluated. Experiments scale from 2 up to 128 nodes. In each experiment, message sizes start from 1 Byte and increase up to 128 kB.

Table 7.2: Benchmarked all-to-all algorithms.

Algorithm	Rounds (r)	Peers in round j	Contiguous Message Buffers
Ring	$p-1$	src: $(i-j) \bmod r$ dst: $(i+j) \bmod r$	✓
One-Factor	p	$(i-j) \bmod r$	✓
Hypercube	$p-1$	$i \oplus j$	✓
Bruck	$\lceil logp \rceil$	src: $i-(1 \ll j)$ dst: $i+(1 \ll j)$	✗

src = source rank
dst = destination rank

7.3.3.1 Performance

Fig. 7.3 visualizes measured performance results. In each plot, the y-axis shows the total execution time in microseconds for respective message sizes in bytes (x-axis). Note that the total execution is calculated as the arithmetic mean across all processors, after performing at least 1000 (100) warmup iterations for small (large) message sizes, respectively. Different FMPI algorithms are distinguished by line type and color, while the baseline implementation is represented as a black solid line. The sliding window size in the ring-scatter and one-factor algorithms is marked through different shapes.

Below we summarize our performance measurements:

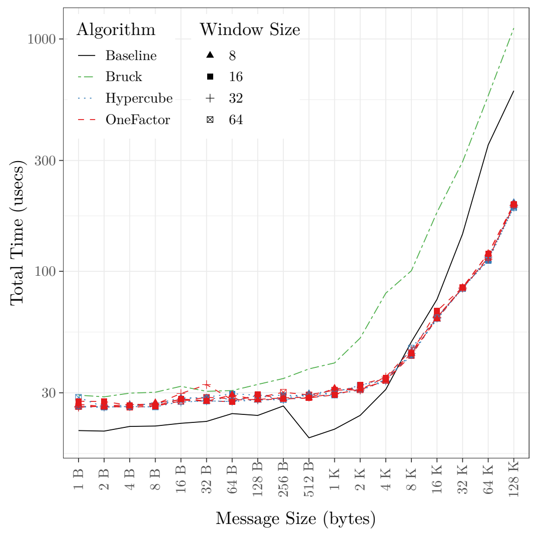
1. The first experiment, visualized in Fig. 7.3a, involves only four nodes (8 ranks). With small problem sizes, the offloading overhead in FMPI degrades performances. However, if message sizes surpass the threshold of 4 kB, direct FMPI algorithms outperform the Baseline with a factor of ≈ 2.

2. Doubling the number of nodes to 8 (16 ranks) shows already notable performance benefits for FMPI, as visualized in Fig. 7.3b. With small messages up to 2 kB, the one-factor algorithm with 8 simultaneous messages per rank is $\approx 20\%$ better than the Baseline. In this experiment, we can already see the advantage of pipelining, as a window size of 8 requests establishes two local synchronization points. The remaining window sizes issue all messages at once and synchronize with a single *wait* to complete outstanding message transmission, which leads to traffic congestion.

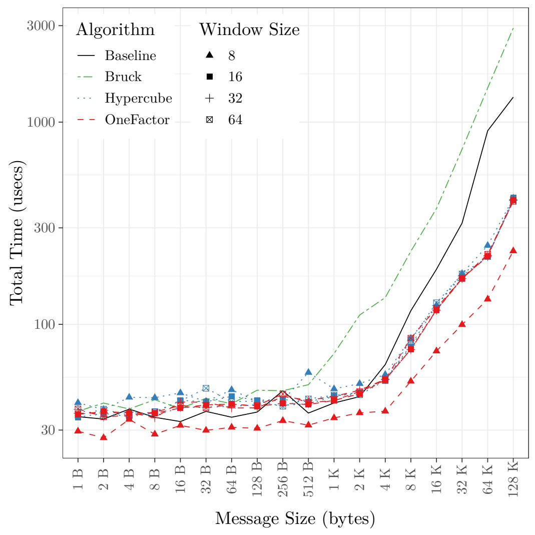
3. In the third experiment (cf. Fig. 7.3c), which involves 64 ranks in total, performance behavior again changes. For very small messages up to 256 B, the Bruck algorithm achieves the best performance due to a logarithmic latency term. The reason, why the Bruck algorithm cannot achieve this efficiency with smaller experiments, results from relatively high message packing and memory rearrangement costs, which are incurred in each message round. Therefore, the Bruck algorithms require a sufficiently large number of ranks to outweigh these overheads. The Bruck algorithm in FMPI is $\approx 20\%$ faster compared to the Baseline. Due to the similar scaling behavior between both lines, we suggest that the Baseline implementation relies on the same algorithm for small messages, however, FMPI is more efficient with its offloading model. If message sizes increase beyond a threshold of 256 B, the one-factor algorithm in FMPI with a window size of 32 dominates performance by an order of magnitude compared to the

other algorithms. With very large messages (> 32 kB), performance is improved up to a factor of 9.3 compared to the Baseline, which we attribute to a very efficient scheduling in the one-factorization pattern.

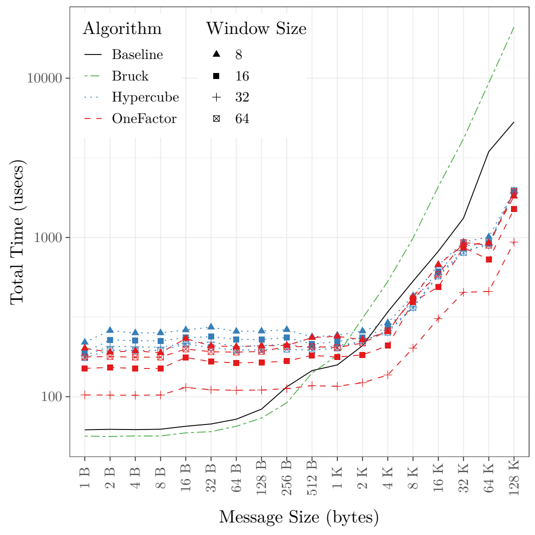
4. The last experiment scales to 128 nodes (cf. Fig. 7.3d) and confirms observed performance trends. For small message sizes, the FMPI Bruck algorithm achieves even higher speedup factors in the range 1.6 to 2. With message sizes beyond 256 B, the one-factor algorithm again shows the best performance with a window size of 64 simultaneous messages per rank. The speedup factor varies in the range 3.5 to 5.



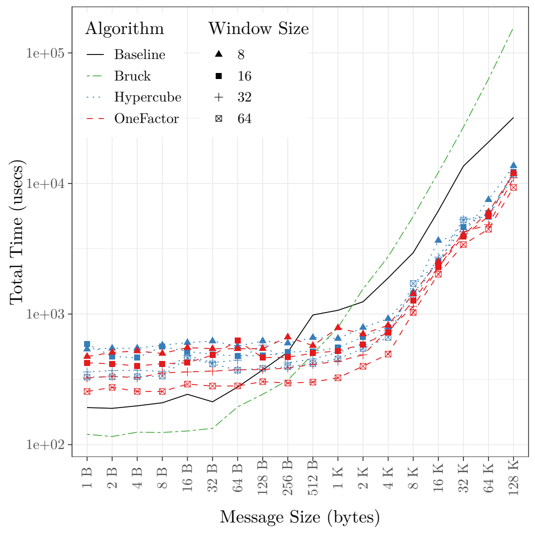
(a) 4 nodes (8 ranks).



(b) 8 nodes (16 ranks).



(c) 32 nodes (64 ranks).



(d) 128 nodes (256 ranks).

Figure 7.3: All-to-all performance: MPI (Baseline) vs. FMPI Algorithms

While the performance speedup of FMPI algorithms compared to an optimized Baseline implementation is impressive, the overall scaling behavior matches with asymptotic complexity of measured communication algorithms.

For very small messages, intermediate routing strategies are more beneficial due to a logarithmic latency term. If message sizes increase, it is better to route messages directly from sources to destination. Surprisingly, the one-factor algorithm achieves the best performance. This was not expected, as we could not find any open source MPI library (MVAPICH, OpenMPI, MPICH) which implements this algorithm for all-to-all communication.

A reasonable expectation was to favor the Hypercube algorithm, because it is often cited as the most efficient scheduling in fat-tree networks [167] which characterizes the network topology of the SuperMUC-NG system.

Another interesting aspect is the sliding window capacity. Performance guidelines often suggest a window size of 1 for large messages [165, 167] to prevent network congestion in rendevouz protocols. However, in our experiments, we can observe that large windows significantly improves performance. This is advantageous particularly in partial aggregation techniques, if computation can proceed with arrived messages in a single batch.

7.3.3.2 Overlap Ratio

The overlap ratio quantifies the relative fraction of overall communication latency, which can be used to proceed with computation. Therefore, the higher the overlap ratio the better available latency hiding capabilities.

Fig. 7.4 presents the observed overlap percentage with 128 nodes, corresponding to the performance scaling study in Fig. 7.3d. On both plots, the x-axis shows increasing message size in bytes, while the y-axis denotes the overlap ratio in percent as calculated in Eqn. (7.1). We add black cross points to visualize the overlap ratio for each measurement.

The left hand side in Fig. 7.4 shows the overlap ratio for the baseline (Intel MPI), while the right hand side visualizes the overlap ratio for FMPI. The fractions of the total execution time which are spent in computation and communication are represented through different colors.

- *Schedule* is the overhead to issue the non-blocking all-to-all communication, i.e. t_c.
- *Computation* represents the fraction where the CPU can perform productive work, i.e. t_w.
- *Synchronization* is the idle waiting time to complete the respective non-blocking all-to-all communication.

We can observe in Fig. 7.4 that the scheduling overhead is similar in both implementations. However, FMPI has a significantly lower synchronization cost which can be utilized to progress available work in the background. The FMPI overlap ratio varies between 81 % to 98 % compared to the baseline implementation does not achieve any notable computation overlap.

7.3.3.3 A Heuristically Tuned All-to-all Implementation

With the performance results at hand, we were able to derive a heuristically tuned version of the all-to-all communication primitive for the SuperMUC-NG system. Based on the number of processors p and the message size n, we implement a decision table in Tbl. 7.3

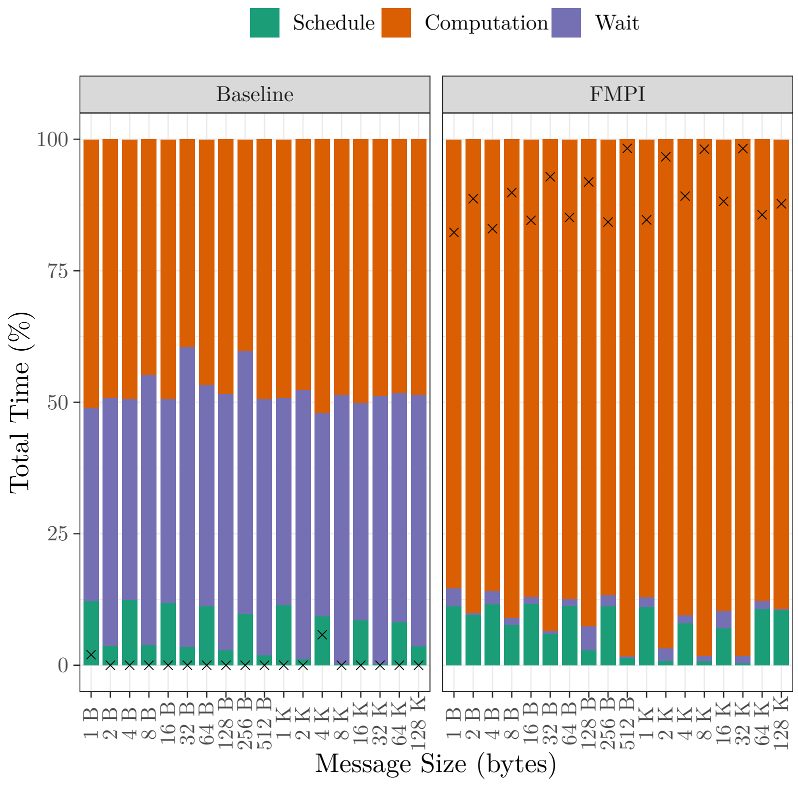


Figure 7.4: All-to-all overlap percentage (128 nodes, 256 ranks).

Table 7.3: Tuned all-to-all parameter decisions.

Processors (p)	Message Size (n)	Algorithm	Window Size
$p \leq 8$	$n \leq 4\,\text{kB}$	Baseline All-to-all	–
$p \leq 8$	$n > 4\,\text{kB}$	One-Factor	p
$8 < p < 64$	–	One-Factor	N[1]
$p \geq 64$	$n < 256\,\text{B}$	Bruck	–
	$n > 256\,\text{B}$	One-Factor	64

[1] N = total number of SMP nodes, which span the communicator.

to select the best communication algorithm at runtime. While Tbl. 7.3 is specific to SuperMUC-NG, it is possible to conduct the benchmarks above on other platforms.

7.3.4 Discussion

Presented results emphasize the need of decoupling communication and computation in asynchronous programming models. However, relying only on asynchronous communication capabilities does not imply any performance improvements. It is necessary to balance communication and computation costs, which depends on HPC applications. The synthetic all-to-all communication benchmark in Fig. 7.4 achieves almost perfect overlap because computation costs are artificially injected to match communication latency. In many real-world applications, this balance can be rarely achieved due to sparse and rather irregular communication patterns.

In the next two sections, we study two proxy applications to understand whether partial aggregation techniques can be used to exploit assessed latency hiding potential.

7.4 Distributed Histogram Sort

In this section we extend our preliminary work of a scalable distributed sort on a large number of PEs [129]. We demonstrate how partial aggregation improves performance in collective communication primitives with large message transfers. The remainder is organized as follows:

- We give a brief summary about the original algorithm, as presented in our paper.
- We discuss possible optimizations of collective communication with partial aggregation.
- We conduct a detailed performance evaluation to assess possible performance speedups.

7.4.1 Algorithm Design

We discuss the problem to sort a vector of size N, which is evenly partitioned on p ranks, i.e. each rank has $n \approx \frac{N}{p}$ elements. The output invariant requires partition p_i to be a sorted permutation of input elements and no element may be larger than any other element on partition p_j, iff $j > i$. Furthermore, each PEs should end up with at most $N(1+\epsilon)/p$, where ϵ is a load balancing threshold. A load balancing threshold $\epsilon = 0$ requires perfect partitioning, where all ranks own *exactly* the same number of elements in the output vector as in the input vector. According to our experience, this is usually not necessary but preferred due to simplicity in application codes.

The algorithm proceeds in four supersteps, as illustrated in Fig. 7.5 with 4 ranks. Involved ranks are arranged from left to right, while time progresses from top to bottom. The white boxes represent local data portions of respective ranks. Below we describe the four supersteps:

1. Local Sort: All ranks sort the local partition of the input vector using a fast shared memory sort algorithm. The expected runtime cost is $\mathcal{O}(n \log n)$.
2. Splitter Selection: Each rank partitions the local portion into p pieces. Our algorithm implementation achieves this using a k-way multi-select algorithm, based on global histograms. The expected runtime complexity accumulates to $\mathcal{O}(p \log^2 pn)$, which we elaborate below.
3. All-to-all Exchange: Each rank routes piece i to rank i using a collective all-to-all exchange. The expected runtime complexity varies between $\mathcal{O}(n(p-1))$ and $\mathcal{O}(n \log p)$, depending on the message size (cf. Ch. 4).
4. Merge: Received pieces are combined into a contiguous sorted sequence. Because the received pieces in the all-to-all exchange are already sorted, a binary merge algorithm can be used with an expected runtime complexity of $\mathcal{O}(n \log p)$.

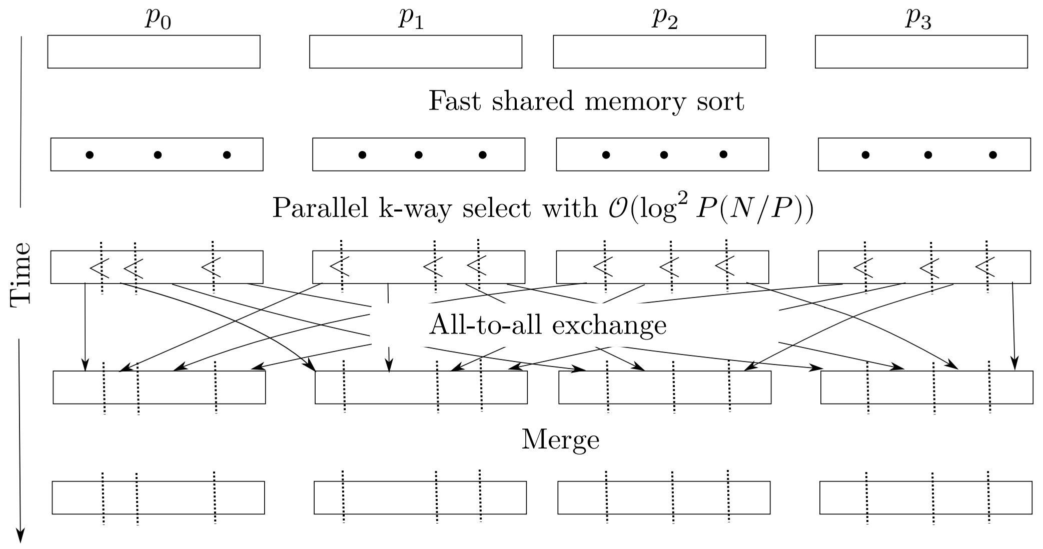


Figure 7.5: Distributed histogram sort algorithm with 4 processors.

The splitter selection phase is a generalization of parallel quickselect to a parallel multi-selection algorithm [129]. Parallel quickselect determines the k-th order statistic in a dataset, which can be solved in linear complexity using the *median-of-medians* strategy [28]. We adopt this algorithm to a multi-select, which allows a partitioning of the local portion into $p-1$ pieces. Our paper shows, that this step can be accomplished with an expected runtime complexity of $\mathcal{O}(n \log p)$ in shared memory [129]. Because all ranks need to agree on a consistent set of $p-1$ *global* splitters, we accumulate the set of *local* splitters in a global reduction, with an expected communication latency of $\mathcal{O}(\log p)$ [184]. After the global reduction, the splitter selection is either complete or needs to be refined with the remaining smaller dataset in subsequent iterations.

The question how many iterations it takes to find the set of global splitters, can be answered as follows. By definition of the median-of-medians strategy, we can discard at least one quarter of the overall dataset each iteration with high probability [28]. Therefore, the recursive depth is at most $\mathcal{O}(\log_{4/3} p)$, which we simplify to $\mathcal{O}(\log p)$. Taking the computation cost per iteration into account, this accumulates to $\mathcal{O}(\log^2 pn)$.

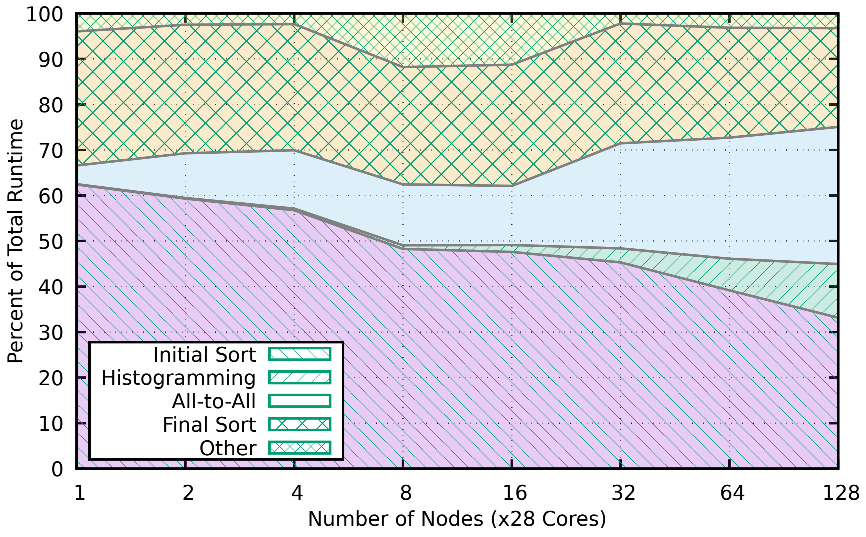


Figure 7.6: Overhead percentage of a weak scaling study.

In the next section we analyze the overhead of each phase. As we heavily rely on collective communication primitives, we show how partial aggregation can improve communication bottlenecks.

7.4.2 Analysis and Optimizations with Partial Aggregation

We analyze the performance of the sort algorithm in a weak scaling study, starting from 1 up to 128 nodes, where each rank allocates 1 GB memory. In weak scaling, the total input size to be sorted scales linearly with the number of nodes. The input data is a vector of 64-bit floating point numbers, following a uniform distribution in the range $[-10^6, 10^6]$. Local work is performed in using multi-threaded sort and merge implementations.

Our paper contains a performance with other state-of-the-art sort algorithms. Here we are only interested to understand the overhead of each phase relative to the total execution time.

Fig. 7.6 visualizes the relative overheads of all phases. The x-axis scales the number of allocated nodes per measurement and the y-axis shows the overhead of algorithm supersteps in percentage. Similar to performance measurements in previous sections, each node executes two MPI Ranks. Each rank in turn runs 24 threads, i.e. one thread per processor core without using SMT capabilities.

The supersteps are ordered bottom-up, starting with the local sort and finishing with the final merge. The top most area accumulates overhead of auxiliary work, which we do not consider in this analysis. For a small number of nodes, the plot shows that local work due to sorting and merging dominates and the all-to-all exchange does not occupy a significant fraction of the total execution time. Further note that the splitter selection cost can be completely neglected even with a large number of nodes.

Scaling up the number of allocated nodes changes the overhead of communication compared to local work. With 128 nodes, the all-to-all exchange takes ≈ 30 of the total execution time compared to the merge phase which takes ≈ 25.

In the original algorithm, the merge phase and the collective all-to-all exchange are not overlapped. Therefore, we adopt the implementation and integrate the FMPI library to support partial aggregation. We utilization two optimization strategies.

- We essentially pipeline collective communication and merging in with a k-ary tree merge. Arrived pieces in the all-to-all exchange are grouped into groups of k pieces which can asynchronously combined in a multi-way merge algorithm. Parallelism is expressed in OpenMP tasks, using the FMPI interface to operate on individual segments of collective communication primitives.
- Arriving segments in the all-to-all communication do not need to be in-order. Tree-based merging satisfies the concept of associative-decomposable functions, which allows arriving pieces to be merged in any order (cf. Sec. 5.3.3) . Therefore, FMPI spawns OpenMP tasks to merge individual segments, independent of the order in which they arrive from the network.

In the next section we discuss attained performance measurements after integrating FMPI.

7.4.3 Performance Results

To understand the effects of partial aggregation, we focus only on the last two phases of the sort algorithm, i.e. the all-to-all exchange and merge. We compare three implementations:

(1) *Baseline*: Collective all-to-all exchange, followed by a parallel merge in shared memory without any overlap.

(2) *FMPI*: Collective all-to-all exchange offloaded to the FMPI library, followed by a parallel merge. Like in the baseline version, the merge does not start before the all-to-all exchange completes.

(3) *FMPI.partial*: Collective all-to-all exchange with communication offload to the FMPI library, including partial aggregation optimizations as described above.

We implement two variants of FMPI, i.e. (2) and (3), to show that assessed performance benefits cannot be attributed to communication offloading itself, but result from improved latency hiding through partial aggregation.

The performance measurements were executed with an increasing number of nodes and data volumes. Similar to the microbenchmarks, each measurement first executed a minimum of 100 warmup iterations until the standard deviation was within 10% of the arithmetic mean. We report the median of 1000 (100) executions for small (large) message blocks, respectively.

Fig. 7.7 visualizes assessed performance results. In each plot, the x-axis scales the message size, and the y-axis shows total execution time in microseconds. With small message sizes, we observe that FMPI versions (2) and (3) perform worse than the baseline implementation.

We attribute this to the offloading overhead, as already suggested in the microbenchmarks (cf. Sec. 7.2).

If message sizes scale beyond 256 B, overlapping communication and computation through partial aggregation pays off. We observe a sweet spot for message sizes in the range 1 to 64 kB, where *FMPI.partial* achieves a speedup factor up to 2.2, compared to the *FMPI* and baseline implementations.

If message sizes surpass the rendevouz threshold of 64 kB, performance improvements shrink to $\approx$ 10–20 %, compared to the baseline version. Remark that in a rendevouz message protocol, sender and receiver are synchronized through an initial handshake before message transmission starts. While occasional synchronization delays between message peers can be reduced through the FMPI offload model, the initial handshake reduces available overlap potential. It is possible to change the rendevouz threshold to a higher level. However, as this is an unspecific feature and platform dependent, we decided to stay with the default settings.

The benchmark series of a distributed sort shows that partial aggregation provides new possibilities for latency hiding in collective communication primitives. It is possible to mimic partial aggregation through pipelining multiple non-blocking collective communications, which can be overlapped in with multiple merge passes. However, it increases the startup costs due redundant collective schedules. And even if scheduling costs can be neglected due to large message sizes, FMPI can exploit more parallelism with the concept of associative-decomposable functions.

The performance evaluation also shows that partial aggregation simplifies interaction between MPI and thread-level parallelism in hybrid programming models. Compared to other approaches described in Ch. 3, users obtain fine-grained control over transmitted segments, while the abstraction of collective primitives is preserved.

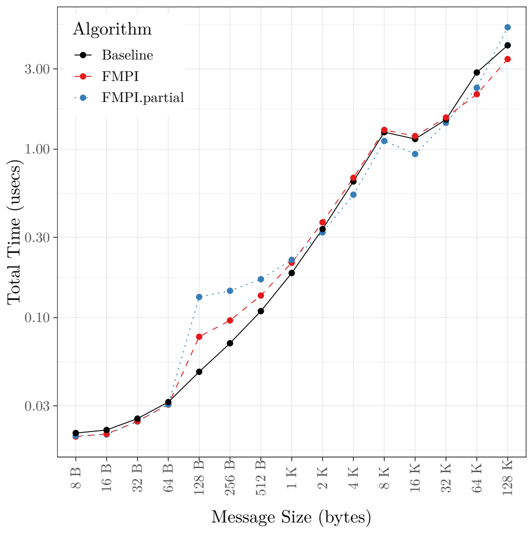
7.5 Heat Equation

As a last example we study a heat equation solver to show how partial aggregation can improve performance in neighborhood collectives.

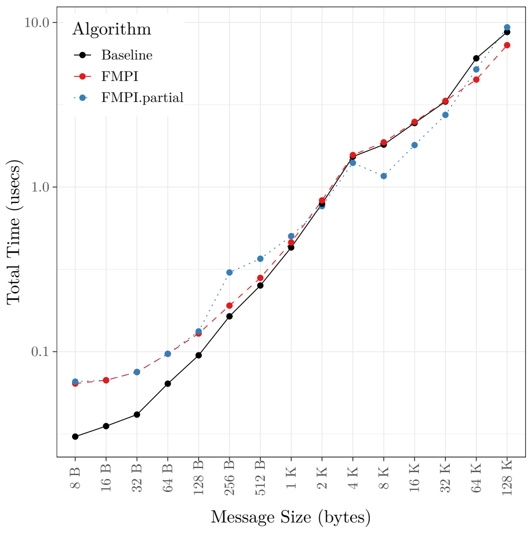
7.5.1 Preparations

The heat equation models heat diffusion through a given region. In a 2-dimensional space, the problem can be expressed through an elliptic Laplace equation [203]:

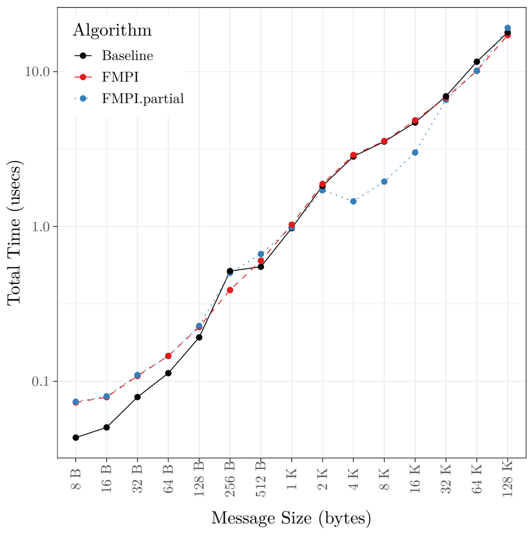
$$\Delta^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{7.2}$$



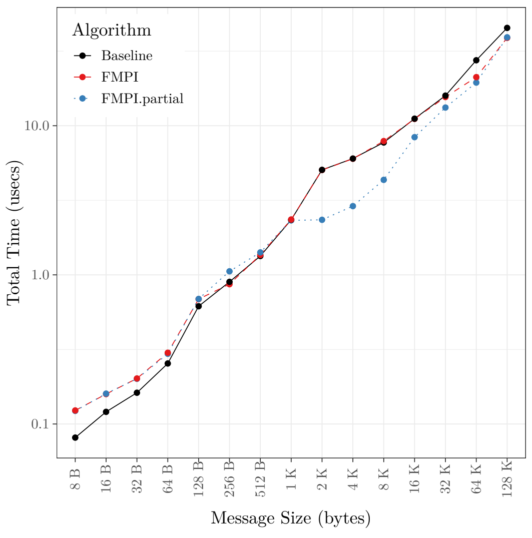
(a) 16 nodes (32 ranks).



(b) 32 nodes (64 ranks).



(c) 64 nodes (128 ranks).



(d) 128 nodes (256 ranks).

Figure 7.7: All-to-all + Merge Performance Comparison.

In Eqn. (7.2), the function $u(x, y)$ is subject to an unknown scalar potential. We solve the problem numerically using an iterative Jacobi solver. Therefore, Eqn. (7.2) can be discretized using central differences in an algebraic solution:

$$u_{i,j}^{n+1} = \frac{1}{4}(u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n), \text{where } i, j \in \{1, 2, \dots, m\} \tag{7.3}$$

In Eqn. (7.3), variable n $(n + 1)$ represents the current (next) iteration of the Jacobi solver and $\frac{1}{m+1}$ is the grid resolution. Usually, computation proceeds until the absolute difference between subsequent iterations lies within a specific threshold, i.e. the success criterion is met. However, the purpose of this benchmark is to understand performance scaling behavior, which is why we define a fixed number of iterations.

For the implementation, we decompose the 2-dimensional problem surface of size m in both directions on a $q \times q$ processor grid. Furthermore, it is ensured that q evenly divides m, i.e. $q \mid m$, which simplifies the benchmark code. Collective neighborhood communication is static and follows the same scheme as the 5-point radius-1 stencil in Fig. 5.2.

Algorithms of collective neighborhood exchanges are studied in both static and dynamic neighborhood relationships [100, 131]. Although the communication step in each iteration can be performed through simple point-to-point communication pairs, collective neighborhood communications enable a persistence property which can be encoded into graph topologies.

A special case of graph topologies are cartesian topologies, which arrange processors in a d-dimensional grid to minimize the number of hops in each neighborhood [78]. The most recent MPI standard provides an interface to construct cartestian communicators for a given number of processors p and dimensionality d. We use this mechanism to get the best possible process topology in our benchmark. Based on a given persistent topology, various aspects of collective neighborhood communication can be optimized [100]:

Fixed Communication Channels The communication can allocated dedicated resources to each communication channel between a set of neighbors. A common example are eager message buffers to improve non-blocking message transmission. Traditional MPI implementations need to support up to p^2 eager buffers, i.e. a single buffer per communication pair. Neighborhood topologies assert MPI libraries to reduce resource allocation only to the set of neighbors.

Synchronization Trees If message sizes vary between communication pairs, optimized trees can be constructed to minimize congestion in neighborhood endpoints. Such optimizations are similar to optimal synchronization trees in dense collective communication patterns [119].

Communication Schedule The communication schedule of messages to different communication peers can be optimized. Common optimization variables are the transport medium, i.e. on-chip vs. off-chip communication, or the load at endpoints to avoid congestion hotspots.

Below, we extend these optimization principles in collective neighborhood communication with partial aggregation techniques. We implement the heat equation benchmark in a hybrid design, where each rank in the processor grid spawns multiple threads to compute Jacobi iterations in parallel.

7.5.2 Optimized Neighborhood Communication

The hybrid implementation relies on a two-level spatial domain decomposition:

1. The first level distributes the problem domain of size $m \times m$ on a 2-dimensional cartesian process topology.
2. In the second level, local blocks of dimensionality $b = \frac{m}{q}$ are decomposed into 1-dimensional column stripes. Accordingly, the size of each block is $\frac{b^2}{t}$, where t is the number of threads.

Fig. 7.8 illustrates the applied decomposition scheme for a single rank with 4 threads. The rectangles represent a processor neighborhood, which perform a boundary exchange each iteration. The colors within a rectangle represent column stripes, which are assigned to available threads.

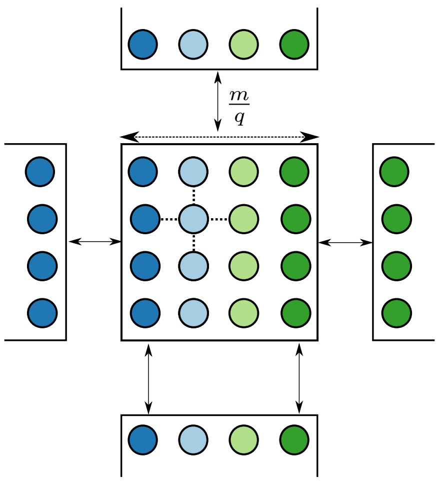


Figure 7.8: Neighborhood of processors in 2-dimensional Jacobi solver. Rectangles represent PEs in distributed memory, while colors represent decomposition among threads in shared memory.

The algorithmic skeleton for the application can be obtained from Alg. 5, where the stencil operation performs an update of each cell according to Eqn. (7.3). In the following, we analyze the communication complexity of the communication step in each iteration.

Fig. 7.8 shows that each thread needs to communicate with its neighbors in the north and south direction, respectively. In the other dimension, only the first and last thread need to communicate along the west and east boundaries, respectively. Therefore, the number of messages in each iteration accumulates to $2t + 2$.

Below we list challenges in existing implementations of the collective neighborhood exchange. In a follow-up step, we show how the integration of our partial aggregation concept solves these challenges.

7.5.2.1 Challenges in Existing Approaches

Current strategies to implement the neighborhood communication pattern described can be classified into the many-sender and single-sender models, as discussed in Sec. 3.4:

- In the many-sender model, each thread independently initiates the neighborhood exchange to/from its neighbors. Because multiple threads issue concurrent communication calls it requires synchronization to guarantee thread-safety within MPI libraries.
- In the single-sender model, all communications are funneled to a single master thread. While it avoids thread synchronization within MPI libraries, it synchronizes all threads twice per iteration. The first synchronization occurs before initiating the neighborhood exchange (Alg. 5, line 3). Another synchronization point occurs before accumulating the boundary conditions as all threads need wait for the master thread to complete outstanding neighbor exchanges (Alg. 5, line 5).

7.5.2.2 Integrating Partial Aggregation

Our approach is a hybrid between the single and many-sender models, based on the partial aggregation concepts. Each message is segmented into individual segments. As an example, messages along the north/south directions in Fig. 7.8 have 4 segments.

A non-blocking collective neighborhood communication call proceeds as follows:

(a) The first thread which enters the collective communication call initiates the communication schedule in the MPI library.
(b) All other threads in turn only signal readiness of their respective segments.

Whether a thread takes path (a) or (b) is internally handled in FMPI by using atomic operations. The FMPI implementation can transmit individual segments or aggregate multiple segments in a single communication channel to trade-off latency and bandwidth costs. Note that this requires to specify the number of segments per dimension at the collective communication call. Otherwise FMPI cannot know if all threads have contributed their segments. As collective neighborhood communications interact only with a small subset of neighbors the additional memory overhead to specify the number of segments can be neglected.

The concept is similar to point-to-point communication which is expected for standardization in the upcoming MPI-4 release [151]. Because collective neighborhood communication is non-blocking, the communication call for each thread immediately returns to proceed with stencil updates to the inner cells. Therefore, we eliminate the single-sender synchronization bottleneck, where the master thread needs to wait for all other threads before initiating the collective communication call. We also avoid coarse-grained synchronization within the MPI library in the many-sender model.

The receiving side is optimized as follows. If computation on the inner cells has finished, threads wait for arriving segments in collective neighborhood exchange. However, it is not necessary to wait until the collective communication is completely done. Instead, we rely on the partial completion feature as described in Sec. 5.2.2. Upon receiving any segment, a single thread can immediately process with stencil updates on the respective message buffer. Therefore, we eliminate the second synchronization point in the single-sender model, where all threads need to wait until the master threads has completed the neighborhood exchange.

Lst. 7.1 lists the essential steps in pseudocode. All iterations are executed in a multi-threaded fashion on a team of threads: The first step is to initiate a segmented collective communication (line 7). It is upon the FMPI library, whether message transmission starts immediately, or multiple segments are aggregated into a single message to trade-off latency and bandwidth. Since collective communication pattern is guaranteed to be non-blocking in FMPI, all threads immediately proceed with computation (line 9). If computation is done, threads cooperatively perform stencil updates on the boundary. If a thread enters the routine `while_some_partial`, it waits for any segment arrival and performs the stencil update on the respective message buffer.

Because all threads can access shared data, it enables to implement a work stealing approach. If some threads are ahead of other threads, they can perform stencil updates on the boundaries for other threads. If all segments have been already processed, threads continue with the next iteration.

Listing 7.1: Neighborhood all-to-all with partial aggregation.

```
    int      d = 2;      // dimensionality
    int      ns[d];      // number of segments in each dimension
    MPI_Comm cart_topo;  // cartesian communicator

 5  while (not done) {
      // all threads participate in the communication call
      auto f = fmpi::neighbor_alltoall(sendbuf, n, ns,...,cart_topo)

      stencil(inner); // local updates
10
      // threads operate on individually arrived segments
      while_some_partial(f).then(segment outer) {
          stencil(outer); // outer cells
      });
15  }
```

7.5.2.3 Persistent Collective Communication

We briefly discuss additional extensions. Since the collective communication patterns is static, it is possible to perform an initial setup of the collective neighborhood exchange outside of the iteration loop. MPI supports this through persistent communication requests. Initializing a persistent request enables the communication library to tune relevant communication parameters as . Persistent communication potentially reduces initialization time of communication patterns, which is required in the above pseudocode in each iteration. However, persistent collective communication requests are not yet supported in the most recent MPI-3 standard and are expected for MPI-4 [151].

7.5.3 Results

This sections presents the benchmark results of the implemented heat equation benchmark. We have implemented four variants to assess performance benefits of the FMPI neighborhood exchange primitives.

1. The baseline implementation with MPI is realized with
   (a) Funnel: Single-sender communication model.
   (b) Multiple: Multi-sender communication model.

   Both models execute the same application and differ only in the communication step, as described above. The neighborhood exchange is implemented with point-to-point communication. Each thread issues a bulk of non-blocking send/receive requests, which are synchronized through a blocking *wait* call after the local stencil updates. Although this is the most simple implementation, it is the most commonly used approach in HPC applications [100].
2. Similarly, the FMPI implementation is also implemented in two flavors which differ on the receiver side:
   (a) Sync: Non-blocking collective communication is synchronized with a blocking *wait* call after updating thread-local cells. Therefore, it is similar to the single-sender model in MPI (1a).
   (b) Partial: This variant integrates the partial concept to operate on individual segments of the receive buffer. This variant does not incur any bulk-synchronization among threads, as discussed in the previous section.

We compare the MPI and FMPI implementations in a weak-scaling study. Weak scaling is generally preferred to evaluate memory-bound applications if total memory requirements cannot be satisfied by a single node. Therefore, execution time scales relative to the parallel fraction of the application code [79].

The benchmark setup is as follows.

- The input parameters of the application are the dimension size m and the number of MPI ranks p. Each node runs two MPI ranks, i.e. one rank per socket.
- All ranks are arranged in a two-dimensional cartesian grid of size $q \times q$. Therefore, each rank *owns* a local block of size $b = \frac{m}{q}$ per dimension.
- Each rank has 20 threads of execution, each operating on a column stripe of size $\frac{1}{20}b^2$ (cf. Fig. 7.8).

We conducted two experiments with block sizes $b = 20\,\text{kB}$ and $b = 40\,\text{kB}$, accumulating to $\approx 50\,\text{MB}$ and $\approx 200\,\text{MB}$ of allocated memory per MPI rank, respectively.

Tbl. 7.4 summarizes the benchmark results. The first column lists the dimension b, i.e. block size per rank. The second column lists the number of processors involved in each experiment. We scaled from from 4 (2 nodes) and up 256 processors (128 nodes).

The third column reports the total execution time. For all experiments, we report the median execution time of 20 runs.

Unsurprisingly, the fastest method is the MPI Funnel mode due to the following reasons:

Table 7.4: Heat Equation Benchmark Results.

Blocksize ($\frac{m}{q}$)	Processors (p)	Total Time (secs)			
		MPI Funnel	MPI Multiple	FMPI Sync	FMPI Partial
20 kB	4	5.80	6.94	6.73	6.66
	16	5.90	7.18	6.83	6.76
	64	6.02	7.28	6.86	6.78
	256	6.14	7.44	6.88	6.79
40 kB	4	1.75	1.98	2.01	1.99
	16	1.66	2.07	1.90	1.87
	64	1.76	2.08	1.93	1.90
	256	1.72	2.14	1.88	1.83

- Computational work in this benchmark is well balanced among all threads. Therefore, there is a low risk for idle waiting time at the synchronisation points described above.
- The number of messages per processors in the neighborhood exchange is significantly smaller compared to the other variants, i.e. 4 vs. 42 messages in the multi-threaded variants in each iteration.
- The transmitted message sizes in the neighborhood exchange are below the rendevouz threshold, enabling very fast communication channels between involved ranks.

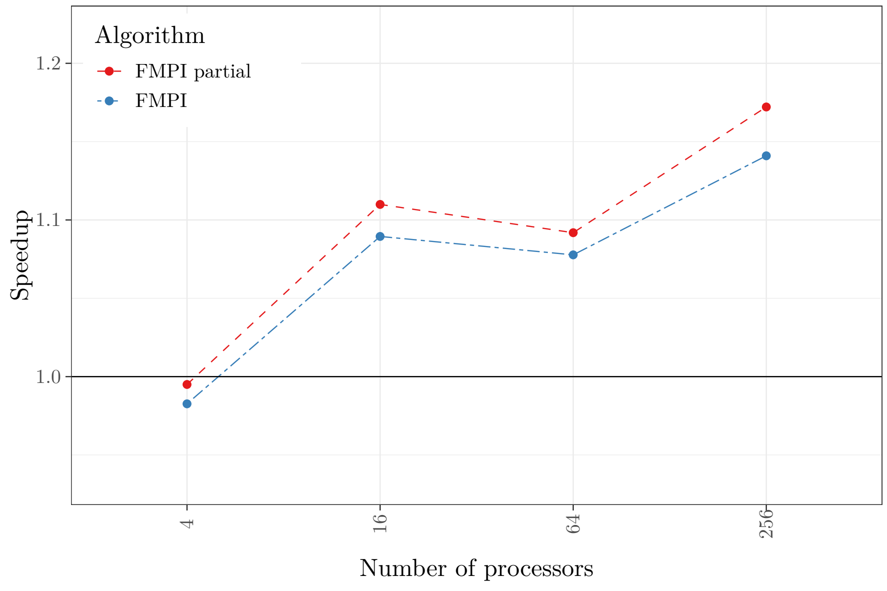
The FMPI implementation transmits each segment in a separate point-to-point communication which is not the most efficient approach due to small message sizes. However, as semantically all threads participate in the collective communication call we need to establish a mapping from sender segments to receiver segments. The FMPI implementation achieves this with message tags, i.e. each pair of threads between two neighbors communicates with an individual tag. In consequence, this increases latency costs due to MPI message matching.

The results in Tbl. 7.4 support this analysis. While the FMPI *funnel* mode is significantly faster in the smaller experiment, the performance advantage reduces to approximately 6% in the largest experiment (bottom row). Therefore we can expect better performance of FMPI with even larger experiments which we consider in future work.

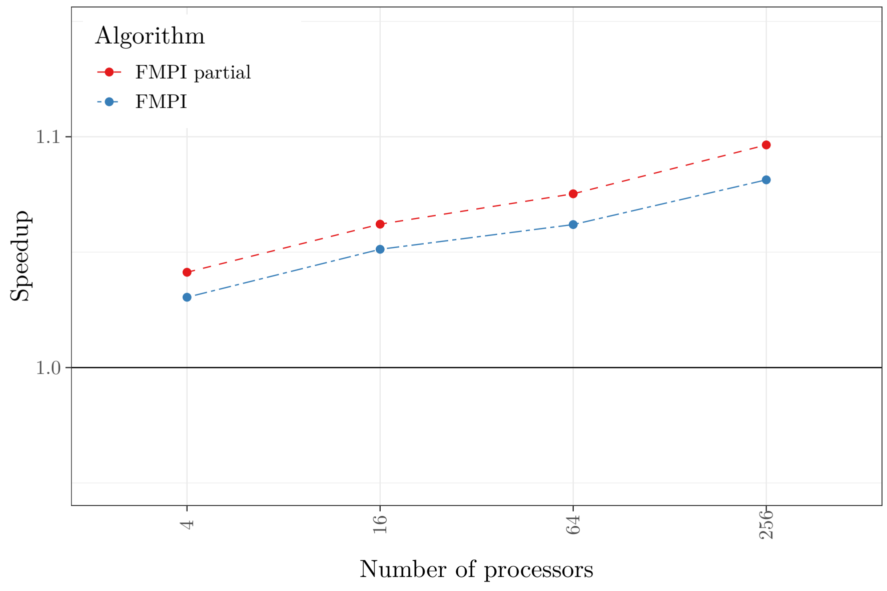
Comparing the FMPI variants, (2a) and (2b), with the multi-threaded MPI version (1b) draws a clearer picture:

- Both FMPI versions outperform the multi-threaded MPI baseline.
- The FMPI partial approach (2b) is consistently faster than the simple FMPI variant (2a) without partial aggregation. The results support our hypothesis that partial aggregation can achieve better performance even in sparse neighborhood collectives.

Fig. 7.9 visualizes attained speedup in the weak scaling study. In both plots, the x-axis scales the number of PEs. The y-axis shows the performance speedup of FMPI relative to the multi-threaded MPI version (2b). With the only exception of the smallest experiment (cf. Fig. 7.9a), FMPI is faster than the baseline implementation, reaching up to 18% speedup.



(a) Blocksize 5k.



(b) Blocksize 10k.

Figure 7.9: Weak scaling study: Heat equation benchmark.

7.5.4 Discussion

We have discussed above that in the FMPI collective neighborhood exchange, each segment is transmitted as a single point-to-point communication to establish a mapping from sender to receiver threads.

A better approach is to reduce message matching overhead by coalescing segments in a single message. This effectively decouples the mapping between sender and receiver segments from message transmission. It is expected that the upcoming MPI-4 release standardizes the *partitioned point-to-point* communication interface, which supports a more efficient implementations of our partial aggregation concepts [74, 151]. Therefore, if this feature is available some time in the near future, we consider to integrate it into FMPI and repeat our experiments.

7.6 Summary

We summarize attained performance results, presented in this evaluation. FMPI provides notable latency hiding potential through a communication offload model in collective communication primitives. However, this alone does not guarantee any performance benefits. With our proposed interface for partial aggregation in FMPI, we are able to efficiently interleave collective communication with local computation. We therefore have answered the following research question:

SQ4 How does partial aggregation improve performance efficiency in collective communications?

Current approaches in HPC codes either replace collective communications with low-level point-to-point calls, which explicitly interleaved with compute tasks, or assemble pipelines of multiple collective communications in software (cf. Sec. 3.3.2). Both approaches significantly increase code complexity, and generally cannot be applied in all use cases.

Partial aggregation instead, relies on rules of binary operations and homomorphism to express data parallelism in collective communications. In the distributed sort case study, we have shown that partial aggregation can interleave all-to-all communication with a recursive merge algorithm in shared memory. The result is a speedup factor of 2, compared to a highly optimized baseline implementation.

In the heat equation case study, we have compared our prototypical implementation of FMPI with two MPI implementations. While the single-threaded MPI funnel version is the fastest implementation in all cases, FMPI performance comes close if we upscale both the number of processors and message sizes. However, FMPI is always faster than the multi-threaded MPI version, which represents current practice in many multi-threaded application codes. We expect that with future features in the MPI standard, we can further improve performance in the FMPI library, as discussed above.

8 Conclusions and Future Work

HPC architects interconnect distributed memory nodes with a high degree of shared memory parallelism, ranging from multi/many-core SMPs to various accelerator architectures. To utilize compute resources on these machines, HPC codes employ a hybrid of distributed and shared memory programming models with explicit and implicit concurrency management. The main challenge is the interaction between explicit and implicit concurrency abstractions.

In our work, we identify interface deficiencies in collective communication primitives. Generally, collective communication primitives abstract complex communication patterns in a group of communication endpoints, defined as MPI processes in the MPI standard. Each MPI process holds local data elements which need to be exchanged with other ranks belonging to the same group. Although algorithms for collective communications are asymptotically optimal, synchronization delays of only few ranks can propagate to all other ranks which affects overall application performance.

Based on ideas of grouping and aggregation in data parallelism we propose the concept of partial aggregation in collective communications. Partial aggregation enables to operate on message buffers of non-blocking collective operation which have not yet been completed. The contributions of our work are two-fold:

C1 Consuming transmitted data in collective communications as early as possible, i.e. early binding.

C2 We formulate the concept of associative-decomposable functions, to decompose opeartions n message buffers of collective communications into independent smalles ones.

C1 has been achieved through the introduction of *partial completion* in MPI collectives. While the MPI implementation processes a non-blocking collective operation, ranks can probe for received data elements to continue with compute tasks. Consequently, the approach exposes inherent data parallelism in collective operations to better absorb communication latency.

C2 utilizes the additionally exposed data parallelism in partial completion (C1). Based on rules of binary operations we compute partial results on received data elements which can be processed in parallel with the respective non-blocking collective communication.

Our concept of partial aggregation (*C1* and *C2*) exposes inherent parallelism in collective communications to improve latency hiding. A theoretical analysis with three representative HPC use cases shows possible performance benefits based on the LogGP model. An empirical evaluation with two of these use cases, a distributed sort and a heat equation solver, confirms that partial aggregation improves performance in practice.

In contrast to existing approaches, partial aggregation supports efficient latency hiding techniques and further improves programmability in hybrid models. Consequently, we have systematically answered the following research question:

> RQ*: How does the integration of partial aggregation improve collective communications?*

In future work, we need to evaluate how the vast amount of asynchronous programming models with implicit concurrency can benefit from partial aggregation concepts. While our work presents an effective interface to improve interaction between MPI collectives and task-based programming abstractions, further HPC applications need to be studied. In this context, it is also interesting how partial aggregation improves the interaction of MPI collectives in accelerator architectures which are gaining increasing popularity in HPC system design.

Another question is how other proposals for MPI standardization complement with our work. We have mentioned partitioned point-to-point communication as a promising approach to improve concurrency in multi-threaded MPI programs. While we show how this proposal integrates into our work, no implementation was available to assess possible performance improvements.

Besides exposing parallelism in collective communication to improve latency hiding, we have not evaluated possible improvements in the underlying communication algorithms in MPI implementations. The MPI standard specifies a fixed ordering of data elements in the receive buffers of collective communications. In some collective algorithms, e.g. Bruck all-to-all, expensive memory copies are required only to obtain the correct order of data elements in the receive buffer. If the MPI implementation would know that computations satisfy the concept of associative-decomposable functions, it can omit these memory copies. However, this requires an interface to provide explicit hints to the MPI implementation.

Acronyms

BSP Bulk Synchronous Parallelism

CFD Computational Fluid Dynamics

CPU Central Processing Unit

CTS Clear-to-send

DAG Directed Acyclic Graph

DMA Direct Memory Access

FFT Fast Fourier Transform

FIFO First-in-first-out

GPU Graphical Processing Unit

HPC High Performance Computing

JVM Java Virtual Machine

LLNL Lawrence Livermore National Laboratory

LRZ Leibniz Rechenzentrum

MPI Message Passing Interface

NIC Network Interconnect

NUMA Non-Uniform Memory Access

NWP Numerical Weather Prediction

OMPI Open MPI

OSU Ohio State University

PDE Partial Differential Equation

PE Processing Element

PGAS Partitioned Global Address Space

PRAM Parallel Random Access Machine

RDMA Remote Direct Memory Access

RTS Ready-to-send

SMP Symmetric Multiprocessor

SMT Simultaneous Multi-threading

UMA Uniform Memory Access

Bibliography

[1] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, *et al.*, "Parallel Programming with Migratable Objects: Charm++ in Practice," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, (New Orleans, Louisana), ser. SC '14, Piscataway, NJ, USA: IEEE Press, 2014, pp. 647–658. DOI: 10.1109/SC.2014.58.

[2] A. Afzal, G. Hager, and G. Wellein, "Desynchronization and Wave Pattern Formation in MPI-Parallel and Hybrid Memory-Bound Programs," in *High Performance Computing*, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, pp. 391–411. DOI: 10.1007/978-3-030-50743-5_20.

[3] S. Agarwal, R. Garg, and N. K. Vishnoi, "The Impact of Noise on the Scaling of Collectives: A Theoretical Approach," in *International Conference on High-Performance Computing*, Springer, 2005, pp. 280–289.

[4] 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 - SPAA '95*, Santa Barbara, California, United States: ACM Press, 1995, pp. 95–105. DOI: 10.1145/215399.215427.

[5] A. Amer, H. Lu, P. Balaji, M. Chabbi, Y. Wei, *et al.*, "Lock Contention Management in Multithreaded MPI," *ACM Transactions on Parallel Computing*, vol. 5, no. 3, 12:1–12:21, Jan. 8, 2019. DOI: 10.1145/3275443.

[6] A. Amer, H. Lu, P. Balaji, and S. Matsuoka, "Characterizing MPI and Hybrid MPI+Threads Applications at Scale: Case Study with BFS," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Shenzhen, China: IEEE, May 2015, pp. 1075–1083. DOI: 10.1109/CCGrid.2015.93.

[7] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka, "MPI+Threads: Runtime Contention and Remedies," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 239–248, Jan. 24, 2015. DOI: 10.1145/2858788.2688522.

[8] Apache Software Foundation. (2006). "Apache Hadoop," [Online]. Available: https://hadoop.apache.org/ (visited on 03/30/2020).

[9] A. A. Awan, K. Hamidouche, A. Venkatesh, and D. K. Panda, "Efficient Large Message Broadcast using NCCL and CUDA-Aware MPI for Deep Learning," in *Proceedings of the 23rd European MPI Users' Group Meeting on - EuroMPI 2016*, Edinburgh, United Kingdom: ACM Press, 2016, pp. 15–22. DOI: 10.1145/2966884.2966912.

[10] J. Bachan, S. Baden, D. Bonachea, P. Hargrove, S. Hofmeyr, *et al.*, "UPC++ Specification v1.0, Draft 8," Sep. 26, 2018. DOI: 10.2172/1477391.

[11] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, *et al.*, “MPI on a Million Processors,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, M. Ropo, J. Westerholm, and J. Dongarra, Eds., vol. 5759, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 20–30. DOI: 10.1007/978-3-642-03 770-2_9.

[12] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, “Fine-Grained Multithreading Support for Hybrid Threaded MPI Programming,” *The International Journal of High Performance Computing Applications*, vol. 24, no. 1, pp. 49–57, Feb. 2010. DOI: 10.1177/1094342009360206.

[13] 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.

[14] M. Barnett, R. Littlefield, D. G. Payne, and R. van de Geijn, “Global Combine on Mesh Architectures with Wormhole Routing,” in *[1993] Proceedings Seventh International Parallel Processing Symposium*, IEEE, 1993, pp. 156–162.

[15] M. Barnett, L. Shuler, R. van De Geijn, S. Gupta, D. G. Payne, *et al.*, “Interprocessor Collective Communication Library (InterCom),” in *Proceedings of IEEE Scalable High Performance Computing Conference*, IEEE, 1994, pp. 357–364.

[16] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: Expressing Locality and Independence with Logical Regions,” in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT: IEEE, Nov. 2012, pp. 1–11. DOI: 10.1109/SC.2012.71.

[17] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, “The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale,” in *2006 IEEE International Conference on Cluster Computing*, Barcelona, Spain: IEEE, 2006, pp. 1–12. DOI: 10.1109/CLUSTR.2006.311846.

[18] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, *et al.*, “An Evaluation of Current High-performance Networks,” in *Proceedings International Parallel and Distributed Processing Symposium*, Nice, France: IEEE Comput. Soc, 2003, p. 10. DOI: 10.1109/IPDPS.2003.1213106.

[19] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, “Optimizing Bandwidth Limited Problems Using One-sided Communication and Overlap,” in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, Rhodes Island, Greece: IEEE, 2006, 10 pp. DOI: 10.1109/IPDPS.2006.1639320.

[20] N. Bell and J. Hoberock, “Thrust: A Productivity-Oriented Library for CUDA,” in *GPU Computing Gems Jade Edition*, Elsevier, 2012, pp. 359–371.

[21] T. Ben-Nun and T. Hoefler, “Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis,” *ACM Computing Surveys*, vol. 52, no. 4, 65:1–65:43, Aug. 30, 2019. DOI: 10.1145/3320060.

[22] M. J. Berger, “Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations,” STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1982.

[23] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, *et al.*, *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead.* 2008.

[24] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R. E. Grant, *et al.*, "A survey of MPI usage in the US exascale computing project," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 3, Feb. 10, 2020. DOI: 10.1002/cpe.4851.

[25] R. S. Bird, "Lectures on constructive functional programming," in *Constructive Methods in Computing Science*, Springer, 1989, pp. 151–217.

[26] G. E. Blelloch, "Scans as primitive parallel operations," *IEEE Transactions on computers*, vol. 38, no. 11, pp. 1526–1538, 1989.

[27] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, *et al.*, "A comparison of sorting algorithms for the connection machine CM-2," in *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures - SPAA '91*, Hilton Head, South Carolina, United States: ACM Press, 1991, pp. 3–16. DOI: 10.1145/113379.113380.

[28] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *J. Comput. Syst. Sci.*, vol. 7, no. 4, pp. 448–461, 1973.

[29] S. H. Bokhari and H. Berryman, "Complete exchange on a circuit switched mesh," in *1992 Proceedings Scalable High Performance Computing Conference*, IEEE Computer Society, 1992, pp. 300–301.

[30] R. Brightwell, S. Goudy, and K. Underwood, "A Preliminary Analysis of the MPI Queue Characterisitics of Several Applications," *International Conference on Parallel Processing*, p. 9, 2005.

[31] R. Brightwell and K. D. Underwood, "An analysis of the impact of MPI overlap and independent progress," in *Proceedings of the 18th Annual International Conference on Supercomputing - ICS '04*, Malo, France: ACM Press, 2004, p. 298. DOI: 10.1145 /1006209.1006251.

[32] J. Bruck, S. Kipnis, and E. Upfal, "Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems," *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, vol. 8, no. 11, p. 14, 1997.

[33] D. Buntinas, G. Mercier, and W. Gropp, "Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem," p. 9,

[34] P.-Y. Calland, J. Dongarra, and Y. Robert, "Tiling on systems with communication/computation overlap," *Concurrency: Practice and Experience*, vol. 11, no. 3, pp. 139–153, 1999.

[35] C. Calvin, "Implementation of parallel FFT algorithms on distributed memory machines with a minimum overhead of communication," *Parallel Computing*, vol. 22, no. 9, pp. 1255–1279, Nov. 1996. DOI: 10.1016/S0167-8191(96)00039-7.

[36] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, Dec. 2014. DOI: 10.1016/j.jpdc.2014.07.003.

[37] E. Castillo, N. Jain, M. Casas, M. Moreto, M. Schulz, *et al.*, "Optimizing computation-communication overlap in asynchronous task-based programs," in *Proceedings of the ACM International Conference on Supercomputing - ICS '19*, Phoenix, Arizona: ACM Press, 2019, pp. 380–391. DOI: 10.1145/3330345.3330379.

[38] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: Theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, Sep. 10, 2007. DOI: 10.1002/cpe.1206.

[39] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of MPI Usage on a Production Supercomputer," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Dallas, TX, USA: IEEE, Nov. 2018, pp. 386–400. DOI: 10.1109/SC.2018.00033.

[40] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[41] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, Jul. 31, 2009, 1314 pp., ISBN: 978-0-262-03384-8.

[42] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, *et al.*, "LogP: Towards a Realistic Model of Parallel Computation," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (San Diego, California, USA), ser. PPOPP '93, New York, NY, USA: ACM, 1993, pp. 1–12. DOI: 10.1145/155332.155333.

[43] D. Culler, L. T. Liu, R. P. Martin, and C. Yoshikawa, "LogP Performance Assessment of Fast Network Interfaces," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, 1995.

[44] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix–matrix multiplication for the GPU," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 4, pp. 1–20, 2015.

[45] A. Danalis, Ki-Yong Kim, L. Pollock, and M. Swany, "Transformations to Parallel Codes for Communication-Computation Overlap," in *ACM/IEEE SC 2005 Conference (SC'05)*, Seattle, WA, USA: IEEE, 2005, pp. 58–58. DOI: 10.1109/SC.2005.75.

[46] H.-V. Dang, S. Seo, A. Amer, and P. Balaji, "Advanced Thread Synchronization for Multithreaded MPI Implementations," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, Madrid, Spain: IEEE, May 2017, pp. 314–324. DOI: 10.1109/CCGRID.2017.65.

[47] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, p. 107, Jan. 1, 2008. DOI: 10.1145/1327452.1327492.

[48] A. Denis, J. Jaeger, and H. Taboada, "Progress Thread Placement for Overlapping MPI Non-blocking Collectives Using Simultaneous Multi-threading," in *Euro-Par 2018: Parallel Processing Workshops*, G. Mencagli, D. B. Heras, V. Cardellini, E. Casalicchio, E. Jeannot, *et al.*, Eds., ser. Lecture Notes in Computer Science, Springer International Publishing, 2019, pp. 123–133.

[49] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, *et al.*, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct. 1974. DOI: 10.1109/JSSC.1974.1050511.

[50] D. DeWitt and M. Stonebraker, "MapReduce: A major step backwards," *The Database Column*, vol. 1, p. 23, 2008.

[51] R. P. Dimitrov, “Overlapping Of Communication And Computation And Early Binding: Fundamental Mechanisms For Improving Parallel Performance On Clusters Of Workstations,” PhD Thesis, 2001.

[52] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, *et al.*, “Enabling MPI interoperability through flexible communication endpoints,” in *Proceedings of the 20th European MPI Users’ Group Meeting on - EuroMPI ’13*, Madrid, Spain: ACM Press, 2013, p. 13. DOI: 10.1145/2488551.2488553.

[53] G. Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, *et al.*, “Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems,” in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, R. Keller, E. Gabriel, M. Resch, and J. Dongarra, Eds., red. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, *et al.*, vol. 6305, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 11–20. DOI: 10.1007/978-3-642-15646-5_2.

[54] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, *et al.*, “OmpSs: A PROPOSAL FOR PROGRAMMING HETEROGENEOUS MULTI-CORE ARCHITECTURES,” *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, Jun. 2011. DOI: 10.1142/S0129626411000151.

[55] A. Dusseau, D. Culler, K. Schauser, and R. Martin, “Fast parallel sorting under LogP: Experience with the CM-5,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 8, pp. 791–805, 1996. DOI: 10.1109/71.532111.

[56] W. Endo and K. Taura, “Parallelized Software Offloading of Low-Level Communication with User-Level Threads,” in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region - HPC Asia 2018*, Chiyoda, Tokyo, Japan: ACM Press, 2018, pp. 289–298. DOI: 10.1145/3149457.3149475.

[57] K. B. Ferreira, P. G. Bridges, R. Brightwell, and K. T. Pedretti, “The impact of system design parameters on application noise sensitivity,” *Cluster computing*, vol. 16, no. 1, pp. 117–129, 2013.

[58] K. B. Ferreira, S. Levy, K. Pedretti, and R. E. Grant, “Characterizing MPI matching via trace-based simulation,” in *Proceedings of the 24th European MPI Users’ Group Meeting on - EuroMPI ’17*, Chicago, Illinois: ACM Press, 2017, pp. 1–11. DOI: 10.1145/3127024.3127040.

[59] S. Fortune and J. Wyllie, “Parallelism in Random Access Machines,” in *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, (San Diego, California, USA), ser. STOC ’78, New York, NY, USA: ACM, 1978, pp. 114–118. DOI: 10.1145/800133.804339.

[60] A. Friedley, T. Hoefler, G. Bronevetsky, A. Lumsdaine, and C.-C. Ma, “Ownership passing: Efficient distributed memory programming on multi-core systems,” *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 177–186, 2013.

[61] M. Frigo and S. Johnson, “FFTW: An adaptive software architecture for the FFT,” in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP ’98 (Cat. No.98CH36181)*, vol. 3, May 1998, 1381–1384 vol.3. DOI: 10.1109/ICASSP.1998.681704.

[62] K. Fuerlinger, T. Fuchs, and R. Kowalewski, "DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Ieee, Sydney, Australia: IEEE, Dec. 2016, pp. 983–990. DOI: 10.1109/HPCC-SmartCity-DSS.2016.0140.

[63] K. Fuerlinger, J. Gracia, A. Knüpfer, D. Hünich, T. Fuchs, *et al.*, "DASH - Distributed Data Structures and Parallel Algorithms in a Global Address Space," in *Software for Exascale Computing - SPPEXA 2016-2019*, ser. Lecture Notes in Computational Science and Engineering, H.-J. Bungartz, S. Reiz, B. Uekermann, P. Neumann, and W. E. Nagel, Eds., Cham: Springer International Publishing, 2020. DOI: 10.1007/978-3-030-47956-5.

[64] K. Fürlinger, R. Kowalewski, T. Fuchs, and B. Lehmann, "Investigating the performance and productivity of DASH using the Cowichan problems," in *Proceedings of Workshops of HPC Asia on - HPC Asia '18*, ACM, Chiyoda, Tokyo: ACM Press, 2018, pp. 11–20. DOI: 10.1145/3176364.3176366.

[65] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, *et al.*, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., red. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, *et al.*, vol. 3241, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 97–104. DOI: 10.1007/978-3-540-30218-6_19.

[66] A. Gara, M. A. Blumrich, D. Chen, G.-T. Chiu, P. Coteus, *et al.*, "Overview of the Blue Gene/L system architecture," *IBM Journal of research and development*, vol. 49, no. 2.3, pp. 195–212, 2005.

[67] A. V. Gerbessiotis and L. G. Valiant, "Direct bulk-synchronous parallel algorithms," *Journal of parallel and distributed computing*, vol. 22, no. 2, pp. 251–267, 1994.

[68] S. M. Ghazimirsaeed and A. Afsahi, "Accelerating MPI message matching by a data clustering strategy," in *High Performance Computing Symposium (HPCS 2017). Kingston*, 2017.

[69] S. M. Ghazimirsaeed, R. E. Grant, and A. Afsahi, "A dynamic, unified design for dedicated message matching engines for collective and point-to-point communications," *Parallel Computing*, vol. 89, p. 102 547, Nov. 2019. DOI: 10.1016/j.parco.2019.102547.

[70] M. T. Goodrich, N. Sitchinava, and Q. Zhang, "Sorting, Searching, and Simulation in the MapReduce Framework," in *Algorithms and Computation*, T. Asano, S.-i. Nakano, Y. Okamoto, and O. Watanabe, Eds., vol. 7074, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 374–383. DOI: 10.1007/978-3-642-25591-5_39.

[71] S. Gorlatch, "Systematic extraction and implementation of divide-and-conquer parallelism," in *Programming Languages: Implementations, Logics, and Programs*, H. Kuchen and S. Doaitse Swierstra, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1996, pp. 274–288. DOI: 10.1007/3-540-61756-6_91.

[72] S. Gorlatch, "Send-receive considered harmful: Myths and realities of message passing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 26, no. 1, pp. 47–56, 2004.

[73] A. Grama, Ed., *Introduction to Parallel Computing*, 2nd ed. Harlow, England ; New York: Addison-Wesley, 2003, 636 pp., ISBN: 978-0-201-64865-2.

[74] R. E. Grant, M. G. F. Dosanjh, M. J. Levenhagen, R. Brightwell, and A. Skjellum, "Finepoints: Partitioned Multithreaded MPI Communication," in *High Performance Computing*, M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, Eds., vol. 11501, Cham: Springer International Publishing, 2019, pp. 330–350. DOI: 10.1007/978-3-030-20656-7_17.

[75] W. Gropp, "MPICH2: A New Start for MPI Implementations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, J. Volkert, P. Kacsuk, and J. Dongarra, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2002, pp. 7–7. DOI: 10.1007/3-540-45825-5_5.

[76] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sep. 1996. DOI: 10.1016/0167-8191(96)00024-5.

[77] W. Gropp and R. Thakur, "Thread-safety in an MPI implementation: Requirements and analysis," *Parallel Computing*, vol. 33, no. 9, pp. 595–604, Sep. 2007. DOI: 10.1016/j.parco.2007.07.002.

[78] W. D. Gropp, "Using Node Information to Implement MPI Cartesian Topologies," in *Proceedings of the 25th European MPI Users' Group Meeting*, Barcelona Spain: ACM, Sep. 23, 2018, pp. 1–9. DOI: 10.1145/3236367.3236377.

[79] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[80] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, 0th ed. CRC Press, Jul. 2, 2010, ISBN: 978-0-429-19061-2. DOI: 10.1201/EBK1439811924.

[81] R. H. Halstead, "MULTILISP: A language for concurrent symbolic computation," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 4, pp. 501–538, Oct. 1, 1985. DOI: 10.1145/4472.4478.

[82] J. M. Hashmi, S. Chakraborty, M. Bayatpour, H. Subramoni, and D. K. Panda, "Designing Efficient Shared Address Space Reduction Collectives for Multi-/Many-cores," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Vancouver, BC: IEEE, May 2018, pp. 1020–1029. DOI: 10.1109/IPDPS.2018.00111.

[83] B. Hendrickson and K. Devine, "Dynamic load balancing in computational mechanics," *Computer Methods in Applied Mechanics and Engineering*, vol. 184, no. 2-4, pp. 485–500, Apr. 2000. DOI: 10.1016/S0045-7825(99)00241-8.

[84] J. L. Hennessy and D. Patterson, *Computer Architecture A Quantitative Approach, 6th Edition*. Morgan Kaufmann, 2018.

[85] D. Hensgen, R. Finkel, and U. Manber, "Two algorithms for barrier synchronization," *International Journal of Parallel Programming*, vol. 17, no. 1, pp. 1–17, 1988.

[86] M. Herlihy, V. Luchangco, N. Shavit, and M. Spear, *The Art of Multiprocessor Programming*, Second. Philadelphia: Elsevier, Inc, 2020, ISBN: 978-0-12-415950-1.

[87] M.-A. Hermanns, N. T. Hjlem, M. Knobloch, K. Mohror, and M. Schulz, “Enabling callback-driven runtime introspection via MPI_T,” in *Proceedings of the 25th European MPI Users’ Group Meeting on - EuroMPI’18*, Barcelona, Spain: ACM Press, 2018, pp. 1–10. DOI: 10.1145/3236367.3236370.

[88] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, *et al.*, “Improving performance via mini-applications,” *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.

[89] T. Hilbrich, M. Weber, J. Protze, B. R. de Supinski, and W. E. Nagel, “Runtime Correctness Analysis of MPI-3 Nonblocking Collectives,” in *Proceedings of the 23rd European MPI Users’ Group Meeting on - EuroMPI 2016*, Edinburgh, United Kingdom: ACM Press, 2016, pp. 188–197. DOI: 10.1145/2966884.2966906.

[90] N. Hjelm, M. G. F. Dosanjh, R. E. Grant, T. Groves, P. Bridges, *et al.*, “Improving MPI Multi-threaded RMA Communication Performance,” in *Proceedings of the 47th International Conference on Parallel Processing*, (Eugene, OR, USA), ser. ICPP 2018, New York, NY, USA: ACM, 2018, 58:1–58:11. DOI: 10.1145/3225058.3225114.

[91] R. W. Hockney, “The communication challenge for MPP: Intel Paragon and Meiko CS-2,” *Parallel Computing*, vol. 20, no. 3, pp. 389–398, Mar. 1, 1994. DOI: 10.1016/S0167-8191(06)80021-9.

[92] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell, “sPIN: High-performance streaming Processing In the Network,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver Colorado: ACM, Nov. 12, 2017, pp. 1–16. DOI: 10.1145/3126908.3126970.

[93] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, *et al.*, “MPI + MPI: A new hybrid approach to parallel programming with MPI plus shared memory,” *Journal of Computing*, vol. 95, no. 12, pp. 1121–1136, Dec. 1, 2013. DOI: 10.1007/s00607-013-0324-2.

[94] T. Hoefler, P. Gottschling, and A. Lumsdaine, “Leveraging non-blocking collective communication in high-performance applications,” in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures - SPAA ’08*, Munich, Germany: ACM Press, 2008, p. 113. DOI: 10.1145/1378533.1378554.

[95] T. Hoefler, A. Lichei, and W. Rehm, “Low-Overhead LogGP Parameter Assessment for Modern Interconnection Networks,” in *2007 IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, USA: IEEE, 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370593.

[96] T. Hoefler and A. Lumsdaine, “Message progression in parallel computing - to thread or not to thread?” In *2008 IEEE International Conference on Cluster Computing*, Tsukuba: IEEE, Sep. 2008, pp. 213–222. DOI: 10.1109/CLUSTR.2008.4663774.

[97] T. Hoefler and A. Lumsdaine, “Optimizing non-blocking collective operations for infiniband,” in *2008 IEEE International Symposium on Parallel and Distributed Processing*, Miami, FL, USA: IEEE, Apr. 2008, pp. 1–8. DOI: 10.1109/IPDPS.2008.4536138.

[98] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for MPI," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing - SC '07*, Reno, Nevada: ACM Press, 2007, p. 1. DOI: 10.1145/1362622.1362692.

[99] T. Hoefler and D. Moor, "Energy, Memory, and Runtime Tradeoffs for Implementing Collective Communication Operations," *Supercomputing Frontiers and Innovations*, vol. 1, no. 2, pp. 58-75–75, Sep. 15, 2014. DOI: 10.14529/jsfi140204.

[100] T. Hoefler and T. Schneider, "Optimization principles for collective neighborhood communications," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT: IEEE, Nov. 2012, pp. 1–10. DOI: 10.1109/SC.2012.86.

[101] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the Influence of System Noise on Large-Scale Applications by Simulation," in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, USA: IEEE, Nov. 2010, pp. 1–11. DOI: 10.1109/SC.2010.12.

[102] 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 - HPDC '10*, Chicago, Illinois: ACM Press, 2010, p. 597. DOI: 10.1145/1851476.1851564.

[103] T. Hoefler, C. Siebert, and A. Lumsdaine, "Scalable communication protocols for dynamic sparse data exchange," *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 159–168, May 2010. DOI: 10.1145/1837853.1693476.

[104] T. Hoefler and J. L. Traff, "Sparse collective operations for MPI," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, Rome, Italy: IEEE, May 2009, pp. 1–8. DOI: 10.1109/IPDPS.2009.5160935.

[105] Z. Hu, H. Iwasaki, and M. Takechi, "Formal derivation of efficient parallel programs by construction of list homomorphisms," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 3, pp. 444–461, 1997.

[106] H. Huang and E. Chow, "Overlapping Communications with Other Communications and Its Application to Distributed Dense Matrix Computations," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rio de Janeiro, Brazil: IEEE, May 2019, pp. 501–510. DOI: 10.1109/IPDPS.2019.00060.

[107] Hyun-Wook Jin, S. Sur, Lei Chai, and D. Panda, "LiMIC: Support for High-Performance MPI Intra-node Communication on Linux Cluster," in *2005 International Conference on Parallel Processing (ICPP'05)*, Oslo, Norway: IEEE, 2005, pp. 184–191. DOI: 10.1109/ICPP.2005.48.

[108] G. Iannello, "Efficient algorithms for the reduce-scatter operation in LogGP," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 9, pp. 970–982, 1997.

[109] IBM Corporation. (2020). "IBM Spectrum MPI," [Online]. Available: https://www.ibm.com/products/spectrum-mpi (visited on 03/30/2020).

[110] InfiniBand Trade Association. (Jun. 24, 2020). "InfiniBand Architecture Specification Release 1.2," [Online]. Available: http://www.infinibandta.org (visited on 06/24/2020).

[111] 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 - PPoPP '01*, Snowbird, Utah, United States: ACM Press, 2001, pp. 133–142. DOI: 10.1145/379539.379592.

[112] Intel Corporation. (2020). "Intel MPI Library," [Online]. Available: https://software.intel.com (visited on 03/30/2020).

[113] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda, "Lightweight kernel-level primitives for high-performance MPI intra-node communication over multi-core systems," in *2007 IEEE International Conference on Cluster Computing*, Austin, TX, USA: IEEE, 2007, pp. 446–451. DOI: 10.1109/CLUSTR.2007.4629263.

[114] T. Jones, P. Tomlinson, M. Roberts, S. Dawson, R. Neely, *et al.*, "Improving the Scalability of Parallel Jobs by adding Parallel Awareness to the Operating System," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing - SC '03*, Not Known: ACM Press, 2003, p. 10. DOI: 10.1145/1048935.1050161.

[115] P. Jungblut, R. Kowalewski, and K. Fürlinger, "Source-to-Source Instrumentation for Profiling Runtime Behavior of C++ Containers," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Exeter, United Kingdom: IEEE, Jun. 2018, pp. 948–953. DOI: 10.1109/HPCC/SmartCity/DSS.2018.00157.

[116] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A Task Based Programming Model in a Global Address Space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, (Eugene, OR, USA), ser. PGAS '14, New York, NY, USA: ACM, 2014, 6:1–6:11. DOI: 10.1145/2676870.2676883.

[117] K. Kandalla, A. Buluc, H. Subramoni, K. Tomko, J. Vienne, *et al.*, "Can Network-Offload Based Non-blocking Neighborhood MPI Collectives Improve Communication Overheads of Irregular Graph Algorithms?" In *2012 IEEE International Conference on Cluster Computing Workshops*, Beijing, China: IEEE, Sep. 2012, pp. 222–230. DOI: 10.1109/ClusterW.2012.40.

[118] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, and D. Panda, "A Novel Functional Partitioning Approach to Design High-Performance MPI-3 Non-blocking Alltoallv Collective on Multi-core Systems," in *2013 42nd International Conference on Parallel Processing*, Lyon, France: IEEE, Oct. 2013, pp. 611–620. DOI: 10.1109/ICPP.2013.75.

[119] R. M. Karp, A. Sahay, E. E. Santos, and K. E. Schauser, "Optimal broadcast and summation in the LogP model," in *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1993, pp. 142–153.

[120] C. Keppitiyagama and A. Wagner, "Asynchronous MPI messaging on Myrinet," in *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, Apr. 2001. DOI: 10.1109/IPDPS.2001.924989.

[121] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, Portland, OR, USA: IEEE, Sep. 2013, pp. 56–65. DOI: 10.1109/IISWC.2013.6704670.

[122] T. Kielmann, H. Bal, and S. Gorlatch, "Bandwidth-efficient collective communication for clustered wide area systems," in *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, Cancun, Mexico: IEEE Comput. Soc, 2000, pp. 492–499. DOI: 10.1109/IPDPS.2000.846026.

[123] T. Kielmann, H. E. Bal, and K. Verstoep, "Fast Measurement of LogP Parameters for Message Passing Platforms," in *Parallel and Distributed Processing*, ser. Lecture Notes in Computer Science, J. Rolim, Ed., vol. 1800, Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 1176–1183. DOI: 10.1007/3-540-45591-4_162.

[124] 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 - PPoPP '99*, Atlanta, Georgia, United States: ACM Press, 1999, pp. 131–140. DOI: 10.1145/301104.301116.

[125] J. Klinkenberg, P. Samfass, M. Bader, C. Terboven, and M. S. Müller, "CHAMELEON: Reactive Load Balancing for Hybrid MPI+OpenMP Task-Parallel Applications," *Journal of Parallel and Distributed Computing*, vol. 138, pp. 55–64, Apr. 2020. DOI: 10.1016/j.jpdc.2019.12.005.

[126] R. Kowalewski, T. Fuchs, K. Fürlinger, and T. Guggemos, "Utilizing Heterogeneous Memory Hierarchies in the PGAS Model," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Cambridge: IEEE, Mar. 2018, pp. 353–357. DOI: 10.1109/PDP2018.2018.00063.

[127] R. Kowalewski and K. Fürlinger, "Nasty-MPI: Debugging Synchronization Errors in MPI-3 One-Sided Applications," in *Euro-Par 2016: Parallel Processing*, ser. Lecture Notes in Computer Science, P.-F. Dutot and D. Trystram, Eds., vol. 9833, Cham: Springer International Publishing, 2016, pp. 51–62. DOI: 10.1007/978-3-319-43659-3_4.

[128] R. Kowalewski and K. Fürlinger, "Debugging Latent Synchronization Errors in MPI-3 One-Sided Communication," in *Tools for High Performance Computing 2016*, C. Niethammer, J. Gracia, T. Hilbrich, A. Knüpfer, M. M. Resch, *et al.*, Eds., Cham: Springer International Publishing, 2017, pp. 83–96. DOI: 10.1007/978-3-319-56702-0_5.

[129] R. Kowalewski, P. Jungblut, and K. Fürlinger, "Engineering a Distributed Histogram Sort," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, Albuquerque, NM, USA: IEEE, Sep. 2019, pp. 1–11. DOI: 10.1109/CLUSTER.2019.8891005.

[130] R. Kumar, A. R. Mamidala, M. J. Koop, G. Santhanaraman, and D. K. Panda, "Lock-Free Asynchronous Rendezvous Design for MPI Point-to-Point Communication," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, A. Lastovetsky, T. Kechadi, and J. Dongarra, Eds., vol. 5205, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 185–193. DOI: 10.1007/978-3-540-87475-1_27.

[131] S. Kumar, P. Heidelberger, D. Chen, and M. Hines, "Optimization of applications with non-blocking neighborhood collectives via multisends on the blue gene/p supercomputer," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, IEEE, 2010, pp. 1–11.

[132] S. Kumar, A. R. Mamidala, D. A. Faraj, B. Smith, M. Blocksome, *et al.*, “PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer,” in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, Shanghai, China: IEEE, May 2012, pp. 763–773. DOI: 10.1109/IPDPS.2012.73.

[133] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, *et al.*, “A large-scale study of MPI usage in open-source HPC applications,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver Colorado: ACM, Nov. 17, 2019, pp. 1–14. DOI: 10.1145/3295500.3356176.

[134] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1, 1978. DOI: 10.1145/359545.359563.

[135] S. Li, T. Hoefler, and M. Snir, “NUMA-aware shared-memory collective communication for MPI,” in *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, 2013, pp. 85–96.

[136] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, *et al.*, “The Linux scheduler: A decade of wasted cores,” in *Proceedings of the Eleventh European Conference on Computer Systems - EuroSys ’16*, London, United Kingdom: ACM Press, 2016, pp. 1–16. DOI: 10.1145/2901318.2901326.

[137] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, “Challenges in Parallel Graph Processing,” *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, Mar. 2007. DOI: 10.1142/S0129626407002843.

[138] X. Luo, W. Wu, G. Bosilca, T. Patinyasakdikul, L. Wang, *et al.*, “ADAPT: An event-based adaptive collective communication framework,” in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, Tempe Arizona: ACM, Jun. 11, 2018, pp. 118–130. DOI: 10.1145/3208040.3208054.

[139] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero, “Overlapping communication and computation by using a hybrid MPI/SMPSs approach,” in *Proceedings of the 24th ACM International Conference on Supercomputing - ICS ’10*, Tsukuba, Ibaraki, Japan: ACM Press, 2010, p. 5. DOI: 10.1145/1810085.1810091.

[140] T. G. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming.* Pearson Education, 2004.

[141] J. D. McCalpin, “SC16 Invited Talk: Memory Bandwidth and System Balance in HPC Systems,” 2016.

[142] F. McSherry, M. Isard, and D. G. Murray, “Scalability! But at what Cost?” In *15th Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[143] P. J. Mendygral, N. Radcliffe, K. Kandalla, D. Porter, B. J. O’Neill, *et al.*, “WOMBAT: A Scalable and High-performance Astrophysical Magnetohydrodynamics Code,” *The Astrophysical Journal Supplement Series*, vol. 228, no. 2, p. 23, Feb. 23, 2017. DOI: 10.3847/1538-4365/aa5b9c.

[144] S. H. Mirsadeghi and A. Afsahi, “Topology-aware rank reordering for mpi collectives,” in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2016, pp. 1759–1768.

[145] S. H. Mirsadeghi, J. L. Traff, P. Balaji, and A. Afsahi, "Exploiting Common Neighborhoods to Optimize MPI Neighborhood Collectives," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, Jaipur: IEEE, Dec. 2017, pp. 348–357. DOI: 10.1109/HiPC.2017.00047.

[146] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, Raleigh, North Carolina, USA: IEEE, Sep. 2009, pp. 261–270. DOI: 10.1109/PACT.2009.22.

[147] G. Moore, "Cramming More Components Onto Integrated Circuits," *McGraw-Hill New York, NY, USA*, vol. 86, no. 1, pp. 82–85, Jan. 1965. [Online]. Available: http://ieeexplore.ieee.org/document/658762/.

[148] C. A. Moritz and M. I. Frank, "LoGPC: Modeling Network Contention in Message-Passing Programs," *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, vol. 12, no. 4, p. 12, 2001.

[149] F. Mößbauer, R. Kowalewski, T. Fuchs, and K. Fürlinger, "A Portable Multidimensional Coarray for C++," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, IEEE, Mar. 2018, pp. 18–25. DOI: 10.1109/PDP2018.2018.00012.

[150] MPI Forum. (2012). "MPI: A Message-Passing Interface Standard (Version 3.0)," [Online]. Available: https://www.mpi-forum.org/ (visited on 03/30/2020).

[151] MPI Forum. (Jul. 6, 2020). "MPI: A Message-Passing Interface Standard (Version 4.0 Draft)," [Online]. Available: https://github.com/mpi-forum/mpi-issues/issues/136 (visited on 07/22/2020).

[152] A. Nataraj, A. Morris, A. D. Malony, M. Sottile, and P. Beckman, "The ghost in the machine: Observing the effects of kernel operation on parallel application performance," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing - SC '07*, Reno, Nevada: ACM Press, 2007, p. 1. DOI: 10.1145/1362622.1362662.

[153] A. Nigay, T. Schneider, T. Hoefler, R. X. M. UPC, and D. R. M. ARM, "MB3 MS13–TinyMPI tasking prototype Version 1.0,"

[154] OpenMP Architecture Review Board. (2018). "OpenMP API Specification (Version 5.0)," [Online]. Available: https://www.openmp.org (visited on 07/20/2020).

[155] M. F. Pace, "BSP vs MapReduce," *Procedia Computer Science*, vol. 9, pp. 246–255, 2012. DOI: 10.1016/j.procs.2012.04.026.

[156] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web.," Stanford InfoLab, 1999.

[157] T. N. Palmer, "Predicting uncertainty in forecasts of weather and climate," *Reports on progress in Physics*, vol. 63, no. 2, p. 71, 2000.

[158] D. K. Panda, K. Tomko, K. Schulz, and A. Majumdar, "The MVAPICH Project: Evolution and sustainability of an open source production quality mpi library for hpc," in *Workshop on Sustainable Software for Science: Practice and Experiences, Held in Conjunction with Int'l Conference on Supercomputing (WSSPE)*, 2013.

[159] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, *et al.*, "Optimizations in a high-performance conjugate gradient benchmark for IA-based multi- and many-core processors," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 11–27, Feb. 2016. DOI: 10.1177/10943420 15593157.

[160] R. van der Pas, E. Stotzer, and C. Terboven, *Using OpenMP—The Next Step: Affinity, Accelerators, Tasking, and SIMD.* MIT Press, Oct. 20, 2017, 392 pp., ISBN: 978-0-262-53478-9. Google Books: Z2k7DwAAQBAJ.

[161] M. M. A. Patwary, P. Dubey, S. Byna, N. R. Satish, N. Sundaram, *et al.*, "BD-CATS: Big data clustering at trillion particle scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15*, Austin, Texas: ACM Press, 2015, pp. 1–12. DOI: 10.1145/2807591.2807616.

[162] K. T. Pedretti, C. Vaughan, K. S. Hemmert, B. Barrett, and P. O. Box, "Application Sensitivity to Link and Injection Bandwidth on a Cray XT4 System," p. 8, 2000.

[163] S. Pellegrini, T. Hoefler, and T. Fahringer, "On the Effects of CPU Caches on MPI Point-to-Point Communications," in *2012 IEEE International Conference on Cluster Computing*, Beijing, China: IEEE, Sep. 2012, pp. 495–503. DOI: 10.1109 /CLUSTER.2012.22.

[164] F. Petrini, Wu-chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The Quadrics network (QsNet): High-performance clustering technology," in *HOT 9 Interconnects. Symposium on High Performance Interconnects*, Stanford, CA, USA: IEEE Comput. Soc, 2001, pp. 125–130. DOI: 10.1109/HIS.2001.946704.

[165] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, *et al.*, "Performance analysis of MPI collective operations," *Cluster Computing*, vol. 10, no. 2, pp. 127–143, May 10, 2007. DOI: 10.1007/s10586-007-0012-0.

[166] J. G. Proakis, *Digital Signal Processing: Principles, Algorithms, And Applications, 4/E.* Pearson Education, Sep. 2007, 1160 pp., ISBN: 978-81-317-1000-5.

[167] R. Rabenseifner, "Optimization of Collective Reduction Operations," in *Computational Science - ICCS 2004*, M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds., red. by T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, *et al.*, vol. 3036, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–9. DOI: 10.1007/978-3-540-24685-5_1.

[168] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes," in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, Weimar: IEEE, Feb. 2009, pp. 427–436. DOI: 10.1109/PDP.2009.43.

[169] P. Sanders, "Algorithm Engineering — An Attempt at a Definition," in *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, Berlin, Heidelberg: Springer-Verlag, Sep. 1, 2009, pp. 321–340. [Online]. Available: https://doi.org/10.1007/978-3-642-03456-5_22.

[170] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev, *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox.* Cham: Springer International Publishing, 2019, ISBN: 978-3-030-25208-3 978-3-030-25209-0. DOI: 10.1007/978-3-030-25209-0.

[171] 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.

[172] P. Sanders and J. L. Träff, “The Hierarchical Factor Algorithm for All-to-All Communication,” in *Euro-Par 2002 Parallel Processing*, B. Monien and R. Feldmann, Eds., red. by G. Goos, J. Hartmanis, and J. van Leeuwen, vol. 2400, Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 799–803. DOI: 10.1007/3-540-45706-2_112.

[173] J. Schuchart and J. Gracia, “Global Task Data-Dependencies in PGAS Applications,” in *High Performance Computing*, M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, Eds., vol. 11501, Cham: Springer International Publishing, 2019, pp. 312–329. DOI: 10.1007/978-3-030-20656-7_16.

[174] J. Schuchart, R. Kowalewski, and K. Fuerlinger, “Recent experiences in using MPI-3 RMA in the DASH PGAS runtime,” in *Proceedings of Workshops of HPC Asia*, ser. HPC Asia ’18, ACM, Chiyoda, Tokyo: Association for Computing Machinery, Jan. 31, 2018, pp. 21–30. DOI: 10.1145/3176364.3176367.

[175] D. S. Scott, “Efficient all-to-all communication patterns in hypercube and mesh topologies,” in *The Sixth Distributed Memory Computing Conference, 1991. Proceedings*, IEEE Computer Society, 1991, pp. 398–399.

[176] J. Shalf, S. Dosanjh, and J. Morrison, “Exascale Computing Technology Challenges,” in *High Performance Computing for Computational Science – VECPAR 2010*, J. M. L. M. Palma, M. Daydé, O. Marques, and J. C. Lopes, Eds., vol. 6449, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–25. DOI: 10.1007/978-3-642-193 28-6_1.

[177] H. Shan, S. Williams, W. de Jong, and L. Oliker, “Thread-level parallelization and optimization of NWChem for the Intel MIC architecture,” in *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, 2015, pp. 58–67.

[178] M. Si and P. Balaji, “Process-Based Asynchronous Progress Model for MPI Point-to-Point Communication,” in *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Bangkok: IEEE, Dec. 2017, pp. 206–214. DOI: 10.1109 /HPCC-SmartCity-DSS.2017.27.

[179] M. Si, A. J. Pena, J. Hammond, P. Balaji, M. Takagi, *et al.*, “Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures,” in *2015 IEEE International Parallel and Distributed Processing Symposium*, Hyderabad, India: IEEE, May 2015, pp. 665–676. DOI: 10.1109/IPDPS.2015.35.

[180] E. Solomonik and L. V. Kale, “Highly scalable parallel sorting,” in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, Atlanta, GA: IEEE, Apr. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470406.

[181] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. (2020). “Top500 List,” [Online]. Available: https://www.top500.org/ (visited on 06/01/2020).

[182] A. S. Tanenbaum and D. J. Wetherall, *Computer Networks*. Pearson Education, Feb. 28, 2012, 961 pp., ISBN: 978-0-13-307262-4. Google Books: IRUvAAAAQBAJ.

[183] K. Al-Tawil and C. Moritz, “LogGP quantified: The case for MPI,” in *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No.98TB100244)*, Jul. 1998, pp. 366–367. DOI: 10.1109/HPDC.1998.710033.

[184] R. Thakur, R. Rabenseifner, and W. Gropp, “Optimization of Collective Communication Operations in MPICH,” *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, Feb. 2005. DOI: 10.1177/10943420 05051521.

[185] J. L. Traff, “Hierarchical gather/scatter algorithms with graceful degradation,” in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, IEEE, 2004, p. 80.

[186] J. L. Träff, F. D. Lübbe, A. Rougier, and S. Hunold, “Isomorphic, Sparse MPI-like Collective Communication Operations for Parallel Stencil Computations,” in *Proceedings of the 22nd European MPI Users’ Group Meeting on ZZZ - EuroMPI ’15*, Bordeaux, France: ACM Press, 2015, pp. 1–10. DOI: 10.1145/2802658.2802663.

[187] J. L. Träff, A. Ripke, C. Siebert, P. Balaji, R. Thakur, *et al.*, “A Simple, Pipelined Algorithm for Large, Irregular All-gather Problems,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, A. Lastovetsky, T. Kechadi, and J. Dongarra, Eds., vol. 5205, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 84–93. DOI: 10.1007/978-3-540-87475-1_16.

[188] J. L. Träff, A. Rougier, and S. Hunold, “Implementing a classic: Zero-copy all-to-all communication with mpi datatypes,” in *Proceedings of the 28th ACM International Conference on Supercomputing - ICS ’14*, Munich, Germany: ACM Press, 2014, pp. 135–144. DOI: 10.1145/2597652.2597662.

[189] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, “System noise, OS clock ticks, and fine-grained parallel applications,” in *Proceedings of the 19th Annual International Conference on Supercomputing - ICS ’05*, Cambridge, Massachusetts: ACM Press, 2005, p. 303. DOI: 10.1145/1088149.1088190.

[190] K. Underwood, K. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell, “A hardware acceleration unit for MPI queue processing,” in *19th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2005. DOI: 10.1109/IPDPS.20 05.30.

[191] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, *et al.*, “Improving concurrency and asynchrony in multithreaded MPI applications using software offloading,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC ’15*, Austin, Texas: ACM Press, 2015, pp. 1–12. DOI: 10.1145/2807591.2807602.

[192] L. G. Valiant, “A Bridging Model for Parallel Computation,” *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. DOI: 10.1145/79173.79181.

[193] S. S. Vazhkudai, B. R. de Supinski, A. S. Bland, A. Geist, J. Sexton, *et al.*, “The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems,” in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Dallas, TX, USA: IEEE, Nov. 2018, pp. 661–672. DOI: 10.110 9/SC.2018.00055.

[194] A. Wagner, D. Buntinas, R. Brightwell, and D. K. Panda, "Application-bypass reduction for large-scale clusters," *International Journal of High Performance Computing and Networking*, vol. 2, no. 2-4, pp. 99–109, 2004.

[195] W. D. Wallis, *One-Factorizations.* New York; London: Springer, 2011, ISBN: 978-1-4419-4766-6.

[196] M. S. Warren and J. K. Salmon, "A parallel hashed oct-tree n-body algorithm," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, 1993, pp. 12–21.

[197] C. Weinhold, A. Lackorzynski, J. Bierbaum, M. Küttler, M. Planeta, *et al.*, "FFMK: A Fast and Fault-Tolerant Microkernel-Based System for Exascale Computing," in *Software for Exascale Computing - SPPEXA 2013-2015*, ser. Lecture Notes in Computational Science and Engineering, H.-J. Bungartz, P. Neumann, and W. E. Nagel, Eds., vol. 113, Cham: Springer International Publishing, 2016, pp. 405–426. DOI: 10.1007/978-3-319-40528-5_18.

[198] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures," 1407078, Sep. 1, 2009, p. 1 407 078. DOI: 10.2172/1407078.

[199] T. S. Woodall, R. L. Graham, R. H. Castain, D. J. Daniel, M. W. Sukalski, *et al.*, "TEG: A High-Performance, Scalable, Multi-network Point-to-Point Communications Methodology," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., red. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, *et al.*, vol. 3241, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 303–310. DOI: 10.1007/978-3-540-30218-6_43.

[200] T. Xiao, J. Zhang, H. Zhou, Z. Guo, S. McDirmid, *et al.*, "Nondeterminism in MapReduce considered harmful? an empirical study on non-commutative aggregators in MapReduce programs," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 44–53.

[201] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement," in *Languages and Compilers for Parallel Computing*, G. R. Gao, L. L. Pollock, J. Cavazos, and X. Li, Eds., red. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, *et al.*, vol. 5898, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 172–187. DOI: 10.1007/978-3-642-13374-9_12.

[202] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[203] D. Zill and W. Wright, *Differential Equations with Boundary-Value Problems.* Cengage Learning, 2012, ISBN: 978-1-111-82706-9. [Online]. Available: https://books.google.de/books?id=0UX8e0xdOr0C.

[204] J. A. Zounmevo and A. Afsahi, "An Efficient MPI Message Queue Mechanism for Large-scale Jobs," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, Singapore, Singapore: IEEE, Dec. 2012, pp. 464–471. DOI: 10.1109/ICPADS.2012.70.

[205] J. A. Zounmevo and A. Afsahi, "A Fast and Resource-conscious MPI Message Queue Mechanism for Large-scale Jobs," *Future Generation Computer Systems*, vol. 30, pp. 265–290, Jan. 2014. DOI: 10.1016/j.future.2013.07.003.

List of Figures

List of Tables

A FMPI Application Programming Interface

The FMPI library is a C++ library and requires the following dependencies.

- A C++14 compatible compiler.
- A MPI library which supports the MPI-3.1 specification [150].

Applications can use the FMPI API and a native MPI library in parallel. It is guaranteed that MPI communicators in FMPI do not interfere with pre-defined MPI communicators.

A.1 Interface Specification

The interface specification serves as an API reference of our FMPI library which we have developed in our work. It consists of three parts:

- Initialization and finalization of the FMPI runtime.
- Collective communication interface.
- Future interface to operate on asynchronous state of respective collective communications.

A.1.1 Initialization and Finalization

Initializing and finalizing the FMPI runtime must be done by only one thread, the *main thread*. At the initialization stage, FMPI internally constructs progress threads and allocates memory for internal bookkeeping. The finalization stage deallocates these resources, respectively.

```
/// @brief Initializes the FMPI runtime.
///
/// @param argc Number of arguments
/// @param argv Pointer to array of arguments
/// @param thread_level The thread level (default: MPI_THREAD_SERIALIZED)
///
bool fmpi::init(int argc, int argv, int thread_level);

/// @brief finalizes the FMPI runtime environment
///
void fmpi::finalize();
```

A.1.2 Collective Communication

Below we list interface to execute collective communication operations in FMPI. We support a subset of multiple all-to-all collectives as specified in the MPI standard.

```
namespace fmpi::collective {

/// @brief Information about segment in collective operations.
struct segment {
    /// The rank in the given communicator to the collective operation.
    int rank;
    /// The number of elements.
    std::size_t count;
    /// The data type of a single element.
    MPI_Datatype type;
};

/// @brief Execution policy for collective operations.
enum class execution_policy : std::uint8_t {
    /// synchronous (blocking) execution.
    synchronous = 0x00,
    /// asynchronous execution
    asynchronous = 0x01,
    /// fire-and-forget execution
    fire_forget = 0x02,
};

/// @brief Options for collective operations
struct options {
    /// the execution policy (default: asynchronous)
    execution_policy policy = execution_policy::asynchronous;

    /// commutative property for partial aggregation, i.e. whether
    /// the segments are placed in-order in the receive buffer.
    bool commutative = false;
};

/**
    @brief Performs a regular non-blocking MPI all-to-all communication.

    @param[in] sendbuf   The starting address of the send buffer.
    @param[in] sendcount The number of elements in the send buffer.
    @param[in] sendtype  Data type of sendbuf elements.
    @param[out] recvbuf  The starting address of the receive buffer.
    @param[in] recvcount The number of elements to receive.
    @param[in] recvtype  Data type of recvbuf elements.
    @param[in] comm      The MPI communicator.
    @param[in] options   The options for the collective operation.
    @Return    future    A future handle.
*/
fmpi::future alltoall(
    const void*   sendbuf,
    std::size_t   sendcount,
    MPI_Datatype  sendtype,
    void*         recvbuf,
    std::size_t   recvcount,
    MPI_Datatype  recvtype,
    MPI_Comm      comm,
    options       opts);

/**
    @brief Performs a regular MPI all-gather communication.
*/
fmpi::future allgather(
    const void*   sendbuf,
    std::size_t   sendcount,
    MPI_Datatype  sendtype,
    void*         recvbuf,
    std::size_t   recvcount,
    MPI_Datatype  recvtype,
    MPI_Comm      comm,
    options       opts);

/**
    @brief Performs a regular MPI neighborhood all-to-all communication.
*/
fmpi::future neighbor_alltoall(
    const void*   sendbuf,
    std::size_t   sendcount,
    MPI_Datatype  sendtype,
    void*         recvbuf,
    std::size_t   recvcount,
    MPI_Datatype  recvtype,
    MPI_Comm      comm,
    options       opts);

} // namespace fmpi::collective
```

A.1.3 Future Interface

Below we list the FMPI *future* interface to operate on individual segments of asynchronous collective operations.

```
namespace fmpi {

/// @brief Provides a mechanism to access state of an asynchronous collective
/// operation.
class future {

    /// @brief Possible return status in the wait_for method.
    /// @see wait_for
    enum class future_status : std::uint8_t {
        /// Means that the operation is ready.
        ready = 0x00;

        /// Means that the timeout period has been expired.
        timeout = 0x01;
    }

    /**
        @brief Waits for the asynchronous result in a collective operation
               to become available. Releases the shared state of the future.
        @Return void
    */
    void wait();

    /**
        @brief Blocks until the specified timeout period has been elapsed or
               the asynchronous result becomes available, whichever comes first.

        @param timeout_period The maximum duration to block in milliseconds.
        @Return void
    */
    future_status wait_for(std::chrono::milliseconds timeout_period);

    /**
        @brief Checks if the future has a shared state.
        @Return bool True, if the future
    */
    bool valid();

    /**
        @brief Checks if the future is ready. Behavior is undefined
               if valid() == false;
        @Return bool True, if the future
    */
    bool is_ready();

    /**
        @brief Attaches a continuation to the future.
               When the future becomes ready the continuation is called on a
               unspecified thread of execution.
        @param F a callable function of signature void(void).
        @Return future a new future with an attached continuation.
    */
    template < class F >
    future then(F&& callable);
};

/**
    @brief Attaches a continuation to the future which is returned from
           collective operations.
           When any segment in a collective operations (e.g. alltoall) becomes
           ready, the callable of type F is invoked.
    @param f  The future encapsulating the asynchronous collective operation.
    @param fun A callable of signature void(collective::segment).
*/
template < class F >
void when_any_partial(future f, F&& fun);
```

```
/**
    @brief Follows the same behaviour as when_any_partial but operates on
           multiple completed segments.
    @param f  The future encapsulating the asynchronous collective operation.
    @param fun A callable of signature void(std::vector<collective::segment>).
*/
template < class F >
void when_some_partial(future f, F&& fun);

/**
    @brief Attaches a continuation to the future which is returned from
           collective operations.
           The continuation is repeatedly called on successfully delivered
           segments until the future becomes ready.
    @param f  The future encapsulating the asynchronous collective operation.
    @param fun A callable of signature void(std::vector<collective::segment).
*/
template < class F >
void while_some_partial(future f, F&& fun);

} // namespace fmpi
```

A.2 Build Instructions

Building and installing the FMPI library:

```
# Clone Repository
git clone https://github.com/rkowalewski/fmpi
cd fmpi
mkdir -p build && cd build
# Configure build using CMake
cmake ..
# Builds the library and installs it to /usr/local
make install
```

An alternative is in-tree CMake integration. Add the following to your `CMakeLists.txt`:

```
# Configure your own executable
add_executable(myapp.x myapp.cpp);

# link FMPI
add_subdirectory(<path to FMPI>);
target_link_libraries(myapp.x fmpi)
```