On Collective Communication and Notified Read in the Global Address Space Programming Interface (GASPI)

Dissertation
zur Erlangung des mathematisch-naturwissenschaftlichen Doktorgrades
"Doctor rerum naturalium"
der Georg-August-Universität Göttingen
im Promotionsprogramm PCS
der Georg-August University School of Science (GAUSS)

vorgelegt von
Vanessa End
aus Neunkirchen (Saar)
Göttingen, 2016

Acknowledgments

I would like to thank Prof. Dr. Ramin Yahyapour for his guidance, continuous support and fruitful discussions throughout the time of research, studies and writing of this thesis. I would also like to thank Prof. Dr. Gert Lube for his guidance and gentle shoves in the right moments. I am sincerely grateful for the motivation, encouragement and advice of these two mentors.
My great thanks also goes to Dr. Alfred Geiger, Dr. Thomas Alrutz and Dr. Christian Simmendinger for not only giving me the opportunity to dive into high performance computing and research, but also for their insightful comments and stimulating discussions.
I am very thankful for having met many great researchers throughout my doctoral studies, who have always helped me in discussions and through adding new view points to my research - including especially the members of the eScience working group of the GWDG.
Last but not least, I would like to express my thanks to my family and friends. They have helped me through all emotional ups and downs of this thesis and were the ones who enabled me to reach this point in my life in the first place. Very special thanks needs to go to Juliane, who always found the right words at the right time to get me back on track and to my husband, who was so very supportive in the past years.

Abstract

In high performance computing (HPC) applications, scientific or engineering problems are solved in a highly parallel and often necessarily distributed manner. The distribution of work leads to the distribution of data and thus also to communication between the participants of the computation. The application programmer has many different communication libraries and application programming interfaces (APIs) to choose from, one of the most recent libraries being the Global Address Space Programming Interface (GASPI). This library takes advantage of the hardware and especially interconnect developments of the past decade, enabling true remote direct memory access (RDMA) between nodes of a cluster.

The one-sided, asynchronous semantic of GASPI routines opens multiple research questions with respect to the implementation of collective communication routines, i.e., routines, where a group of processors is involved in the communication. The GASPI specification itself only offers two of these collective operations: the allreduce, computing a global result from the data of all participants, and the barrier, constituting a synchronization point for all members of the group. For these collective routines, appropriate underlying algorithms have to be chosen. In the scope of the one-sided, asynchronous and split-phase semantic of GASPI collective routines, algorithms used in other wide-spread communication libraries like the Message-Passing Interface (MPI) may not be fitting any more. In this thesis, existing algorithms have been reevaluated for their usability in GASPI collective routines in the context of a newly designed library GASPI_COLL, amending the existing GASPI implementation GPI2 with additional algorithms for the allreduce and with further collective routines: reduce and broadcast.

For the split-phase allreduce, algorithms with a butterfly-like communication scheme have been extensively tested and found to be very suited due to their low number of communication rounds and involvement of all participants in each communication round. This ensures few repeated calls to the allreduce routine and also very small idling times for all nodes. One of the most wide-spread algorithms for barrier operations, the dissemination algorithm, has been adapted to be usable for the allreduce operation as well. The adapted n-way algorithm shows very good results compared to the native implementation of the GPI2 allreduce and different MPI implementations.

To make the one-sided communication semantic of GASPI manageable for the application programmer, the GASPI specification introduces weak-synchronization primitives, notifying the destination side of the arrival of data. This notification mechanism prevents the necessity of global synchronization points or the waiting on multiple communication requests. This

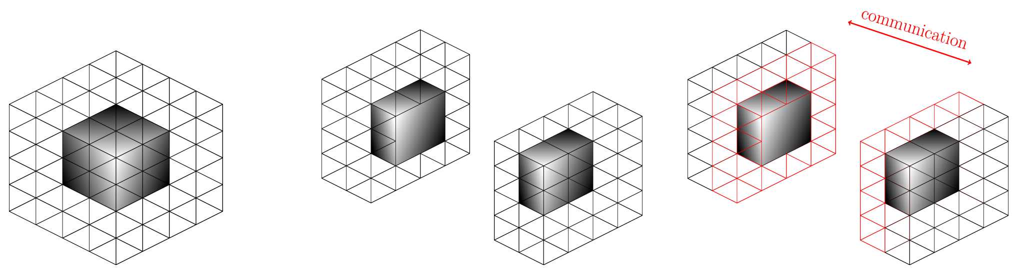
notification mechanism has previously only been available for write-based operations but has been extended to the read routine in the scope of this thesis, introducing `gaspi_read_notify`. With this new routine, the thesis establishes the basis of a completely one-sided, asynchronous graph exploration, implemented with the notified read operation. This enables a broader audience to use data analytical methods on big data. Big data poses a real challenge for graph analytical methods, because the data needs to be distributed on multiple nodes, introducing high communication overhead if two sides are involved in the communication. This issue is eliminated through `gaspi_read_notify`.

Last but not least, the potential usage of `gaspi_read_notify` for a distributed matrix transpose was investigated. Not only is a matrix transpose a wide-spread communication scheme in HPC applications, it can also be considered as a special case of an alltoall communication. The split-phase, one-sided paradigm of GASPI collective routines, has inspired the idea of a partially evaluable alltoallv and as a first step towards this routine, the applicability of `gaspi_read_notify` for the implementation of the alltoall can be deduced from the matrix transpose. On the available systems, this kind of implementation can not be encouraged though. Yet, the experiments in this thesis have also shown the high dependence of communication routines and algorithms on the underlying hardware. Thus, extensive tests on different system architectures will have to be done in the future.

Contents

1. Introduction and Motivation

Many scientific and engineering problems, such as weather forecasts, computational fluid dynamics, molecular dynamics or bioinformatic problems, are being solved by means of modern computer systems. The more accurate the computation of a given problem is, the more data is needed and produced by the application and the more computational power is needed to solve the problem. To cope with the increasing problem sizes, using state of the art hardware on large distributed systems is essential. In these high performance computing (HPC) systems, each component of the system deals with the computation of one part of the complete problem. A simplified example would be the computation of the airflow around a box. To compute the airflow, a grid is laid around the box and the airflow will be computed in each cell in dependence of the results of the neighboring cells. Such a grid is depicted in Fig. 1.1a. If the box is very large, or the grid around the box has too many cells for one computing entity to cope with the computation in a reasonable amount of time, the computation is split onto several computing entities. This immediately necessitates the partitioning of the grid onto the two entities as well, as shown in Fig. 1.1b. Because the computation within each cell is dependent on the neighboring cells, communication is introduced between the two entities where the grid was partitioned (Fig. 1.1c).



(a) box with mesh **(b)** partitioned box **(c)** necessary communication

Figure 1.1.: Steps for computing the airflow around a box: (a) putting a mesh around the box, (b) partitioning the problem (c) communication induced by partitioning of the problem.

In real-world applications, the computation within each cell is more complex, the cells of the grid are not necessarily regular and the problem is partitioned onto hundreds or even thousands of computing entities. The communication introduced through this partitioning of the problem is one of the main bottlenecks in parallel, distributed computation and is becoming more and more important as the HPC community is heading into the exascale era, i.e., an era where large computing systems are theoretically capable of quintillions (10^{18}) of floating point operations per second. Regarding the development of processors, this will necessitate thousands of processors working on the same problem and these thousands of computing entities will have to communicate with each other in order to solve the given problems. The overhead of this communication rises dramatically and the communication bottleneck gains even more relevance when not only two entities need to communicate with each other, but a whole group of entities are involved in a given communication procedure. Typical examples of such group-wise communication routines are the dissemination of data among all entities, or the computation of the global sum or maximum of data spread among these entities. These routines are called collective communication routines and are one of the focal points of this thesis.

Especially when dealing with classical message-passing systems, where all participants need to be active in the communication and need to provide additional memory for the communication buffers, these overheads become a major problem for the scalability of programs. To mitigate the overhead of communication, a perfect overlap of communication and computation is the ultimate goal. To save memory resources, the goal needs to be zero-copy communication. The hardware development of the past years has made both possible through remote direct memory access (RDMA). Through RDMA routines, one entity is capable of accessing the remote entity's memory without any involvement of the central processing unit (CPU) or the remote entity.

As with every new development, new approaches to communication between distributed entities were triggered. One of these approaches was a new view on memory: the partitioned global address space (PGAS). With this new view, several communication libraries, application programming interfaces (APIs) and languages have been developed, one of which is the Global Address Space Programming Interface (GASPI). Asynchronous, one-sided communication routines are the emphasis of this new specification, developed in the scope of a Bundesministerium für Bildung und Forschung (BMBF) project from 2012 to 2015. The routines for the specification were picked with strong limitations, discriminating it from other specifications and standards that offer a whole flood of communication routines. Due to the one-sided, asynchronous approach taken by GASPI, algorithms for collective communication routines have to be reevaluated for their usability in this setting. In addition to that, the semantic of GASPI introduces new hurdles for collective communication routines, maybe one of the reasons that only the barrier and the allreduce are included in the specification.

The definition of the GASPI specification raised many research and design questions, some of which are tackled in this thesis. The core question is what new possibilities a completely asynchronous, one-sided communication interface will introduce. With only one entity being active in communication at all times, one of the main issues is the notification of the remote process that data has been written into its memory. GASPI introduces weak synchronization techniques for this, which enable the remote process to check on a notification instead of flushing a communication queue or using a barrier. This weak synchronization and the possibility of

true RDMA opens new possibilities for split-phase communication routines, where progress can be made without the process remaining in the call and without any CPU involvement.
These new communication paradigms together with the increasing bandwidth and decreasing latency of RDMA-capable, HPC interconnects like the InfiniBand (IB) architecture, raise the question whether collective communication algorithms must be redesigned or whether the return to older algorithms, neglected due to former congestion problems, makes sense in this new setting. The implicit parallelism of these new networks are capable of handling many messages concurrently, and thus, algorithms that can profit from this are predestined for GASPI collective communication routines. This thesis especially deals with the n-way dissemination algorithm and Bruck's algorithm, both transferring n messages per communication round. In addition to this, also a return to algorithms with a butterfly-based communication scheme is investigated, as these algorithms may also benefit from the rising bandwidth of modern networks.
Even though the GASPI specification does not define many collective communication routines, additional collective routines should be investigated in this new setting. What problems arise, when implementing all-to-all, one-to-all or all-to-one collective routines with one-sided communication routines? One question directly arising when thinking of all-to-one or one-to-all collective is, how the overwriting of data can be avoided within such a collective routine. The GASPI specification requires special memory preparation in form of segments for using the one-sided communication routines. The general questions of handling memory segments and notifications within a library are dealt with in the scope of implementing multiple collective communication routines in a GASPI library.
Regarding the notification mechanism included in the GASPI specification more closely, the questions of why only the write-based one-sided communication routines can make use of this mechanism and why the notification always acts as a fence are raised. This thesis evaluates the possibility of a notification for read-based communication routines. Such a routine would be useful in many different situations, for example for a distributed, one-sided graph traversal, which is not possible in this manner so far. For this, `gaspi_read_notify` has been implemented in GPI2 and tested in several use-cases, i.e., in a graph exploration setting for big data analysis problems and in a distributed matrix transpose. In addition to new possibilities in the Big Data area, the notification mechanism has been modified to a more fine-grained model, enabling message-wise notifications instead of fencing notifications.
The main contributions of this thesis can thus be summed up as follows:

1. Algorithms for collective communication are reevaluated for their usability in the scope of GASPI. This especially includes an adaption of the n-way dissemination algorithm for its usage in GASPI allreduce and barrier operations. This algorithm exploits the implicit parallelism offered by modern interconnects, thus needing less communication rounds.
2. A GASPI library for collective communication (GASPI_COLL) is designed and implemented. The library includes additional allreduce algorithms as well as newly imple-

mented reduce and broadcast routines. All library routines are implemented solely with one-sided communication routines of GASPI. In the scope of this, pitfalls of asynchronous, one-sided collective communication are analyzed.

3. The weak synchronization of GASPI write operations is extended to read operations. With this new routine, a distributed, asynchronous graph exploration scheme is presented. Additionally, the notified read is evaluated for the usage within alltoll routines by means of a matrix transpose communication kernel benchmark.

The remainder of the thesis is organized as follows: Chap. 2 will elaborate on the basics of parallel computing glanced at previously in the introduction. It covers different memory architectures, networks and interconnects as well as the basics of communication in parallel computing. Chapter 3 delves further into communication, presenting different communication libraries, languages and APIs relevant to this thesis together with a historic overview on their development. These two chapters establish a foundation and context for the contributions described in the subsequent chapters. Chapter 4 will present an adaption to the n-way dissemination algorithm, initially developed for barriers. The adaption makes it possible to use this algorithm for allreduce operations but also improves the runtimes of barrier operations.
In Chap. 5, the algorithm is then compared to further allreduce algorithms in the scope of the library GASPI_COLL. This library amends the GASPI specification and implementations through additional collective routines and - in the case of the allreduce routine - through additional algorithms. The only available implementation of the GASPI specification is the GPI2, in which only one algorithm is implemented for the allreduce routine, even though the use of different algorithms, chosen in dependence of reduction routine, message and group size, ameliorates the runtimes of the allreduce.
Chapter 6 then introduces a new communication routine, which will be included in the next GASPI specification version. It is a notified read routine - again one-sided and asynchronous but with a weak synchronization mechanism through the notification. The calling entity can thus check on the read data without having to block or involve the remote entity in any way. This notified read routine is predestined for all consumer driven problems and a dynamic work distribution. This is underlined through two use-cases also described in this chapter: the matrix transpose and a graph exploration. The latter would not be possible in such a one-sided, asynchronous manner, with only those routines defined by the GASPI specification prior to the amendment of `gaspi_read_notify`. The addition of this routine necessarily led to a change in the semantic of other weak synchronization routines, which were also adapted to regain a uniform semantic across all routines. The last chapter will conclude the work of the thesis and give an outlook on future research questions related to or based on this thesis.

2. Basics of Parallel and High Performance Computing

This chapter will introduce the most important fundamentals of parallel computing and HPC to put this thesis in a context. In the first section, different memory architectures are introduced: shared memory, distributed memory and the PGAS. The description of the first two, well-known memory architectures is especially given to later emphasize the differences between these and the PGAS. Since there is a close connection between the underlying memory architecture and the communication within a program, different communication techniques will be connected to the memory architectures. A more exhaustive description of different communication languages, APIs and libraries will be done in Chap. 3.
Another important factor, influencing communication algorithms, is the underlying network and interconnect, because network hardware may have a wide range of functionalities, which can differ immensely between two compared networks. In addition to the hardware, there are different software components and the topology of the hardware, both playing an important role. Thus, Sec. 2.2 will introduce the state of the art concepts in HPC interconnects and related fields.
Section 2.3 will then focus on different communication possibilities, with a special emphasis on collective communication. Several different collective communication routines are described (Sec. 2.3.2) together with different algorithms that may be used to implement these collective communication routines (Sec. 2.3.3). This list is not exhaustive but will concentrate on those collective operations, that are relevant to this thesis.

2.1. Memory

When it comes to parallel or even distributed programming, memory architectures need to receive much more attention than in the classic von Neumann architecture model [103] depicted in Fig. 2.1. In the von Neumann model, a CPU, consisting of an arithmetic and logic unit (ALU) and a control unit (CTRL), can run one program at a time. Over the years, this simple model has become more and more complex through the addition of multiple computing cores per processor and the interconnection of several processors through one of many networks. These state of the art systems in HPC, named clusters, combine different memory architectures like distributed and shared memory. Hence, a cluster is often called a hybrid memory architecture.

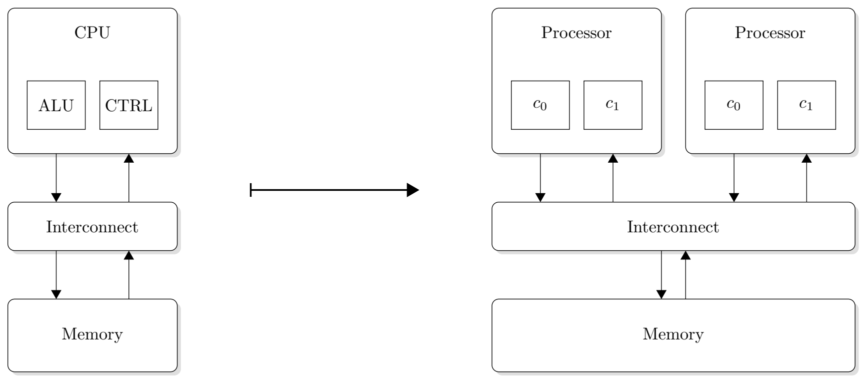
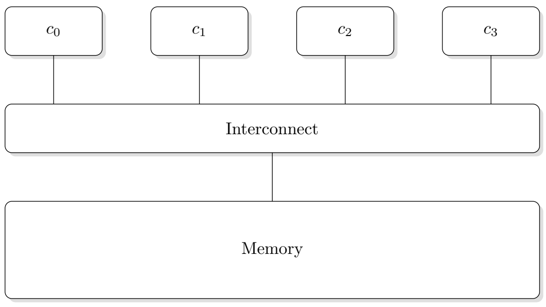


Figure 2.1.: Computer architectures have developed from the classical von Neumann model on the left (1993) to more complex architectures with multiple processors and multiple cores per processor.

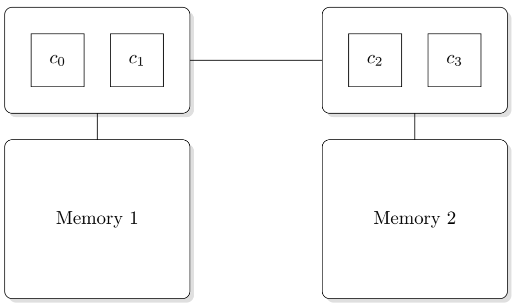
Because an application programmer has to deal with different memory architectures, he also has to deal with different programming paradigms. Good knowledge of the underlying memory is thus a necessary prerequisite for a good parallel program.

2.1.1. Shared Memory

Nowadays, almost everyone uses shared memory architectures, maybe without even being aware of it. Most desktop computers and notebooks have processors with multiple cores and high performance clusters will be equipped with the most up-to-date high-performance hardware with, e.g., 22 cores per processor [63]. When multiple cores are put on one processor, these cores typically share access to at least one cache and of course also to main memory. This is an example of uniform memory access (UMA), as depicted in Fig. 2.2a. Here, every core c_0, c_1, c_2 and c_3 has the same access time to main memory. The different processors are connected either through a bus, a centralized or a hierarchical switch ensuring equal access times of all cores to every memory location [36]. If in contrast, the processors are connected in a way that the access times to different memory regions differ, this is called non-uniform memory access (NUMA) (Fig. 2.2b). On each of the sockets, several cores share the affinity of a larger block of main memory. The sockets are interconnected, letting each core on one socket also access the memory of the other socket. This means, when p_0 accesses some data in memory 2, this will take longer than accessing data in memory 1.
Parallel programs on these architectures are run with multiple threads, such that the resources of each processing unit may be completely exploited. Communication between the different threads may be done through shared variables, because each thread can access every memory region. This introduces race conditions, when two threads try to access the same shared variable at the same time. To handle these competing accesses, an application programmer



(a) UMA



(b) NUMA

Figure 2.2.: Schemes of UMA and NUMA architectures.

may use one of several libraries, that enable control over the different threads and their memory accesses. The most popular libraries, giving a programmer this control, are Open Multi-Processing (OpenMP) [81] and POSIX threads (pthreads) [99].
If a single processor or socket does not deliver enough computing power, more computational resources are needed. The connection of several processors with one another will not only introduce additional computing power, but also remote memory only accessible over the network, i.e., the introduction of a distributed memory architecture.

2.1.2. Distributed Memory

When it comes to cluster computing, the programmer not only has to deal with shared memory, but also with distributed memory. In a cluster there are multiple compute nodes with their own, private memory, connected via some interconnection network, as depicted in Fig. 2.3. Due to this, access times to different memory locations vary depending on source and destination node and the kind of network topology implemented below. This also means, that each process has certain memory regions which have affinity to it and other processes need further software or hardware solutions, that enable explicit communication between the different memory regions to work on data in this memory region. One of these possibilities is message-passing, where the most popular and wide-spread implementation is Message-Passing Interface (MPI) [75], which is described in more detail in Sec. 3.3.
Each of the nodes of a cluster also has a shared memory architecture within the node, as it consists of several cores accessing the same memory. This can either be a UMA or a NUMA architecture and may even differ among the nodes. Efficient usage of these hybrid architectures need a more complex programming approach, which combines shared memory communication with distributed memory communication. One of the most popular hybrid approaches is the combination of MPI and OpenMP. A newer approach is the PGAS programming paradigm, where the distributed memory is (partially) accessible by all processes. This particular paradigm is

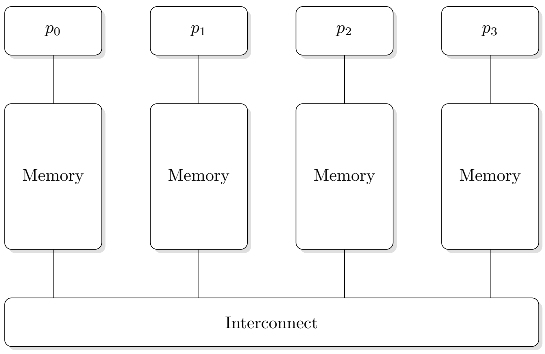


Figure 2.3.: Four processing units p_0 to p_3 each have their own memory, which is connected to the others via some interconnect.

described in the next section.

2.1.3. Partitioned Global Address Space

An important factor in parallel programming is the locality of data, to reduce the runtime-relevant amount of necessary communication. While the memory models distributed memory and shared memory are broadly known and crisply defined, the trend is to merge these two programming models into a hybrid model. The first step towards a hybrid programming model has already been done by using programming APIs for the distributed memory space together with ones designed for the shared memory space. By joining these two ideas, hybrid programs are not only able to use the compute power of multiple nodes, but also to exploit the full capabilities of the compute nodes through threading.
A different approach is a relatively new programming paradigm called PGAS. More on the history of the development of the PGAS can be found in Sec. 3.1. The definition of the PGAS is more abstract than the two memory models previously described and also leaves more room for interpretation. For example, the *Encyclopedia of Parallel Computing* [83], defines that a PGAS system consists of the following:

[[83], p. 1540]

- A set of processors, each with attached local storage. Parts of this local storage can be declared *private* by the programming model, and is not visible to other processors.
- A mechanism by which at least a part of each processor's storage can be *shared* with others. Sharing can be implemented through the network device with system software support, or through hardware shared memory with cache

coherence. This, of course, can result in large variations of memory access latency (typically, a few orders of magnitude) depending on the location and the underlying access method to a particular address.

- Every shared memory location has an *affinity* – a processor on which the location is local and therefore access is quick. Affinity is exposed to the programmer in order to facilitate performance and scalability stemming from "owner compute" strategies.

There is no definition of how the programmer can access or transfer data, as it was defined in the shared or distributed memory concepts. This abstract level of the definition of the PGAS is also made very clear in a blog post of T. Hoefler [46]:

> PGAS is a concept relating to programming large distributed memory machines with a shared memory abstraction that distinguishes between local (cheap) and remote (expensive) memory accesses. PGAS is usually used in the context of PGAS languages such as Co-Array Fortran (CAF) or Unified Parallel C (UPC) where language extensions (typically distributed arrays) allow the user to specify local and remote accesses. In most PGAS languages, remote data can be used like local data, for example, one can assign a remote value to a local stack variable (which may reside in a register) — the compiler will generate the needed code to implement the assignment. A PGAS language can be compiled seamlessly to target a global load/store system.

Such a model is depicted in Fig. 2.4, where each processor has defined part of its memory as local (also called private) and other parts as global (also called shared). In this thesis, the terms local and global will be used, to distinguish memory locations, that are made available in the PGAS from truly shared memory as described above. The set of global memory regions span the PGAS, as highlighted in blue.

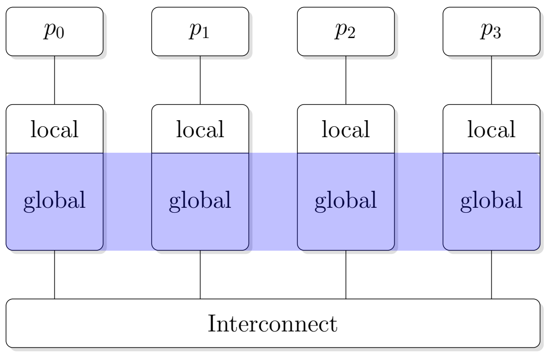


Figure 2.4.: The partitioned global address space (blue) is spanned by by the global segments of each processing unit's memory.

Also for this memory model, the programmer has a variety of communication libraries and APIs to chose from. The first ones that received broader attention by the research community were Unified Parallel C (UPC), an extension to the C standard, and Co-Array Fortran (CAF) [78], an extension to the Fortran language. One of the main disadvantages of these two libraries is, that an existing parallel program with, e.g., MPI communication has to be completely rewritten. A more recent approach, exposing the actual communication to the programmer and being interoperable with state of the art communication libraries, was done by the GASPI-Forum, which has released the first GASPI specification in 2013 [33]. More detail on different possible PGAS communication schemes will be given in Sec. 3.2.4.

Since the global memory segments are only accessible over the interconnect, all PGAS approaches must be implemented on top of some messaging system. These messaging systems strongly depend on the underlying hardware, i.e., the interconnect used for the cluster. The next section will describe different interconnects in more detail.

2.2. Interconnects and Networks

In this section, networks, different interconnects and topologies are introduced, as these all influence the choice of an algorithm for collective communication and the communication possibilities. When talking of networks and communication, a very general, but widely used model for communication layers comes to mind: the Open Systems Interconnection (OSI) model depicted in Fig. 2.5. Routing comprises the lowest three layers of this model and has a large influence on message transferals. Nonetheless, routing techniques will not be discussed in this chapter, because the influence of different routing techniques on communication libraries is out of the scope of this thesis.

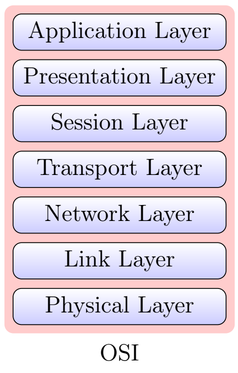


Figure 2.5.: The layers defined by the general OSI model.

The following subsection will introduce several terms and definitions necessary for a comparison between different clusters and the testing of the work in this thesis. Subsection 2.2.3 will then introduce different interconnects available and broadly used in HPC.

2.2.1. Formal Definitions

The following definitions are needed to characterize and thus also compare interconnection networks with each other. These definitions follow those of T. Rauber and G. Rünger in [90] but are adapted to have a consistent notation throughout this thesis. Two important terms when it comes to timing message transfers over a network are bandwidth and latency.

Definition 2.1 *Bandwidth*
The bandwidth of a network describes the maximum rate at which data can be transported through the network. Often the term throughput is used instead of bandwidth.
The aggregated bandwidth of a network is the total bandwidth available to a network.

Definition 2.2 *Latency*
The term latency refers to the time needed for the first bit of a data packet to leave the source until the last bit of the packet has reached the destination.

With these two definitions, it is possible to give a formal definition of the time a message m of size M needs to traverse the network: Let λ be the latency of the given network and β be the bandwidth. Then

$$T_m = \lambda + \frac{M}{\beta} \tag{2.1}$$

is the time needed for the message m to be completely transferred in this network. This time is of course merely a theoretical lower bound, as it does not take contention or overhead (possibly) caused by switches into account.
Depending on the use case of the interconnect network, the topology of the network may vary significantly. To talk about different networks in a general manner and to compare given network topologies with one another, a set of characteristics is needed. Network topologies can be depicted as connection graphs.

Definition 2.3 *The network* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Let $\mathcal{V}$ be a set of vertices (i.e. nodes) to be connected and $\mathcal{E}$ be the edges (i.e. physical links) connecting the nodes. If there is a direct connection e between nodes $u, v \in \mathcal{V}$, then $e = (u, v) \in \mathcal{E}$.

In such a network, there might always be a number of node pairs connected directly through a given edge, but there may also be node pairs, which are not directly linked. Let $v_0, v_k \in \mathcal{V}$ be such a node pair with $(v_0, v_k) \notin \mathcal{E}$. Then we need a different possibility to route a message from v_0 to v_k, i.e., a path through the network.

Definition 2.4 *Path φ*
A sequence of nodes $\varphi(v_0, v_k) = (v_0, v_1, \ldots, v_k)$ is called path of length k between v_0 and v_k if $(v_i, v_{i+1}) \in \mathcal{E}\, \forall i \in \{0, \ldots, k\}$.

These paths through the network may be of different lengths, depending on the node pair. Most interesting is the shortest path through the network, which is often described through the number of hops a message takes.

Definition 2.5 *Hops*
The minimum number of links a message needs to pass on the way from the source node v_0 to the destination node v_k is called the number of hops. One could also say the length of the shortest path $\varphi(v_0, v_k)$ a message takes is the number of hops $h(\varphi(v_0, v_k))$.

Once the number of hops for all node pairs are determined, it is possible to talk of a very important characteristic of network topologies: the diameter of the network.

Definition 2.6 *Diameter*
The diameter $\delta(\mathcal{G})$ of a network is defined as the maximum number of hops between any pair of nodes in the graph:

$$\delta(\mathcal{G}) = \max_{u,v \in \mathcal{V}} \{h(\varphi(u, v))\}$$

Two further important characteristics of networks also have to do with the number of edges: the degree of a network and the bisection width of a network.

Definition 2.7 *Degree*
The degree $d(v)$ of a node $v \in \mathcal{V}$ is the number of links e attached to the node.
The degree $d(\mathcal{G})$ of a network $\mathcal{G}$ is the maximum node degree in the network:

$$d(\mathcal{G}) = \max\{d(v) | v \in \mathcal{V} \text{ and } \mathcal{G} = (\mathcal{V}, \mathcal{E})\}.$$

Definition 2.8 *Bisection Width*
The bisection width of a network describes the minimum number of links which need to be removed to divide the network into two unconnected halves $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1), \mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$, with $\mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset$. If $|\mathcal{V}|$ is odd, $|\mathcal{V}_1|$ and $|\mathcal{V}_2|$ differ by 1.

While the degree of a node or the whole network makes it possible to talk of the reliability of a network and the reachability of a node, the bisection width concretizes this to the connection between two halves of the network. A very similar property of networks - but not to be confused with the bisection width - is the connectivity of a network. The connectivity of a network describes how many nodes or links must fail to disconnect the network - no matter how large or small the different parts of the network are.

Definition 2.9 *Connectivity*
The node connectivity of a network is defined as the minimum number of nodes that must be deleted to obtain two unconnected network parts.
The link connectivity is accordingly the minimum number of links that must be deleted to obtain two unconnected networks or node sets.

In addition to these definitions of characteristics, which are very important in the planning and design of a cluster, the following two terms will most likely be well known under users of the cluster as well: contention and congestion. Unlike the above definitions, the definitions of contention and congestion here are not directly taken from any source but are rather put into context of this thesis in my own words.

Definition 2.10 *Resource Contention*
The competition of different instances trying to access the same resource is called resource contention. With respect to communication over a network, the resource could be a switch, a link or a network interface controller (NIC) while the instances could be threads or processes in a program.

Definition 2.11 *Resource Congestion*
A congested network resource is a overfilled or blocked resource, for example when more data is supposed to be routed by a switch than it can buffer.

A main goal in designing a cluster as well as in designing an application is to avoid both contention of resources and congestion of the network. With these definitions, the different topologies discussed in the following section can be compared to one another.

2.2.2. Different Network Topologies

Since the beginning of distributed computing, the possible network topologies have changed immensely. This section will give an overview of different network topologies in the course of time, as they may influence the development of communication algorithms immensely. Figure 2.6 shows several different topologies, all of which will be presented in this section.
The ring topology in Fig. 2.6a is one of the simplest topologies available. The nodes are connected with bidirectional links, which are subject to a lot of traffic due to the low node degree of 2 and the resulting limited number of paths a message may take. This limit together with the high diameter of $\frac{p}{2}$ results in fast congestion of the network, limiting this topology to usage for small systems or as a part of a larger network.
The fully connected topology in Fig. 2.6b solves the problem of low connectivity and high

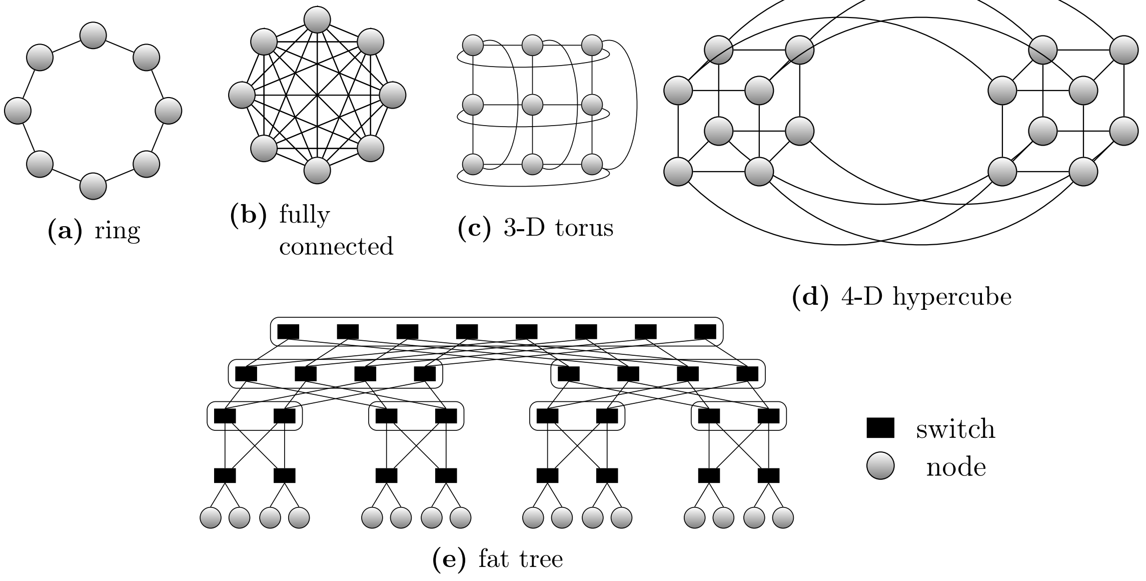


Figure 2.6.: Different network topologies.

contention of the links by adding $P-2$ link to each node. This decreases the diameter of the network to 1 and increases the connectivity to $P-1$, which makes this an ideal topology. Due to the great cost of so many links, this topology will not be feasible for large systems, but may very well be a part of more complex networks like the one described on p. 17.

In terms of link count, connectivity and diameter, mesh-based networks like the 3-D torus in Fig. 2.6c pose a compromise between the ring topology and the fully connected. In a mesh topology, the nodes are arranged in connected rows and columns. In a torus, the last node of each row is additionally connected to the first node of the row. The number of nodes connected through a d-dimensional torus depends on the layout: A symmetrical torus, i.e., one that extends equally in all dimensions, has r^d nodes, where r is the number of nodes in any one dimension. They have a connectivity of $2d$ and a diameter of $d \cdot \lfloor \frac{\sqrt[d]{P}}{2} \rfloor$, which would be equal to $d \cdot \lfloor \frac{r}{2} \rfloor$ in a symmetric torus. Torus networks are very prominent in HPC systems, e.g., as a 5-D torus in the BlueGene/Q [52] or as a 6-D torus in the K computer [30].

Another multidimensional network topology is the hypercube. A d-dimensional hypercube consists of two $(d-1)$-dimensional hypercubes, connecting equivalent nodes in the two $(d-1)$-dimensional hypercubes. Figure 2.6d shows how two 3-D hypercubes are connected to one 4-D hypercube. Hypercubes thus connect $P = 2^d$ nodes and have a connectivity and diameter of $\log_2(P) = d$. In comparison to a d-dimensional torus, a d-dimensional hypercube cannot connect as many nodes, because it is limited to 2^d nodes. A 7-D enhanced hypercube is used at IT4Innovations [66].

Most HPC systems, and especially those used for benchmarking in this thesis, are connected through a switched network topology named fat-tree topology. This network is depicted in

Fig. 2.6e. A fat-tree is essentially a binary tree with the root and inner nodes being switches and the leaf nodes being compute nodes or processors. Towards the root of the tree, the number of links between the different levels of the increases to compensate the increasing link load and to avoid bottlenecks. The fat-tree topology is a multistage switching network, almost all of which go back to the Clos telecommunications network [12].

In addition to these different network topologies, also different types of interconnects, i.e., the actual hardware, play an important role in the setup of an HPC system in order to reduce latency, increase scalability and profit from higher bandwidths. Because different interconnects not only differ in the type of wiring used, but also have very different built-in communication features, a selection of HPC-relevant interconnects are introduced in the upcoming section.

2.2.3. Different Interconnects

Over time, different interconnection hardware and standards have emerged on the high performance computing market. Most systems on the TOP500 list [101] have either an Ethernet based interconnect or an IB based interconnect [100]. The most wide-spread interconnect family is the Ethernet family with 10G Ethernet and Gigabit Ethernet. This is closely followed by IB based systems, while the third largest group of systems with the same interconnect are those with a Cray Aries interconnect, being represented in 7.6% of the TOP500 systems. These three interconnect families all go into different depths with the definition of the components, which can be compared with the OSI model (Fig. 2.7). First, the IB interconnect will be described, as this interconnect is the most important in this thesis. After that, Ethernet, Cray interconnects and some other emerging interconnects will also be described.

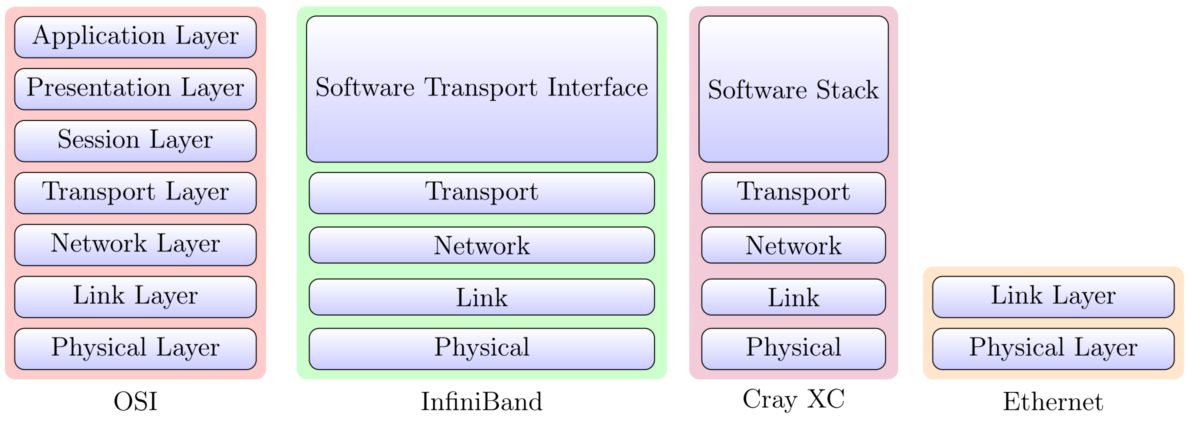


Figure 2.7.: The layers defined by the general OSI model, IB and Ethernet.

InfiniBand

The IB Architecture already emerged in 1999, when Next Generation I/O and Future I/O merged [39]. The InfiniBand Trade Association (IBTA) has then released the "InfiniBand™ Architecture Specification" in 2007 [61], defining a complete stack of communication layers, similar to the OSI model [65]. In Fig. 2.7 the similarities between the two architectures are shown. Any application using Infiniband has direct access to the messaging service defined in the Architecture Specification and needs no involvement of the operating system to communicate with another application or storage.
While the OSI model is a very generic and theoretical model for any kind of network communication, the Infiniband model defines everything from the hardware to the application interface. Starting from the bottom of the stack, the physical layer defines the hardware of the Infiniband stack, including cables, switches, routers and backplane connectors. In this layer, also the physical link speeds are defined: 1x, 4x or 12x. In the first case, a physical link consists of four wires. Two of these wires are reserved for each direction of communication. Accordingly, the 4x and 12x links offer four times or twelve times the speed. Table 2.1 lists the development of IB link speeds in the past years and the perspective aimed at by the IB roadmap [57].

Table 2.1.: Development of Infiniband theoretical raw data rate performance. Values taken from [57] and [59], values for the proprietary Mellanox FDR-10 taken from [72].

Name	SDR	DDR	QDR	FDR-10	FDR	EDR	HDR	NDR
Year	1999	2004	2008		2011	2014	2017	after 2020
Data Rate 1X (Gbit/s)	2.5	5	10	10	∼14	∼25	∼50	
Latency (μs)	5	2.5	1.3	0.7	0.7	0.5		

The link layer includes communication specific work within a local subnet: switching, packet layout and point-to-point link operations. A maximum of 4000 Bytes of payload can be transmitted per packet. Within the link layer, the addressing of the devices is defined through the specification. In addition, a Local Route Header (LRH) is added to the packet. The link layer also supports Quality of Service (QoS) through Virtual Lanes and ensures data integrity.
The network layer then transports the packets from one subnet to another, adding a Global Route Header (GRH) to each packet. In the following the transport layer, the in-order packet delivery is ensured and different transport services are enabled: reliable connection, reliable datagram, unreliable connection, unreliable datagram and raw datagram. Each of these transport services features different aspects, which have direct influence on the the top layer, the Software Transport Interface. Here, a set of verbs are defined for an application to interact with the lower layers of the model. While the semantic of the verbs are defined in the IB architecture specification, the actual implementation and also the naming of the verbs are free to the implementors. The most important, because wide-spread, implementation of these verbs is distributed in the ibverbs library with the OpenFabrics Enterprise Distribution (OFED) stack

by the OpenFabrics Alliance [80]. A more elaborate description of the verbs, especially in the implementation of the ibverbs, is given in Sec. 3.2.1
The most outstanding feature of InfiniBand is the complete offload of communication to the RDMA capable hardware.

Ethernet

Ethernet is still the most wide spread interconnect, used not only in HPC systems, but especially in Local Area Networks. It is standardized in the IEEE 802.3 standard [54], and looks back on a long history, described in [105, 56]. The standard covers everything in the physical layer and in the data link layer of the OSI model in Fig. 2.7 on p. 15, e.g., the cabling, plugs, switches and data packet descriptions. Over time, the standard had to be adapted to the rapid hardware development, going from shared media to a switched network and from coaxial cables to optical cables. With many changes in hardware, the standard was also adapted or amended with new definitions for faster data transfer, i.e., in 1998 Gigabit Ethernet standard was released and 2002 the 10G standard was released [55]. Since 2010 also 40G and 100G Ethernet standards are available. The development of the Ethernet bandwidth can be found in Tab. 2.2.

Table 2.2.: Development of Ethernet standards and theoretical performance, taken from [55] and [53].

Standard	802.3	802.3u	802.3ab	802.3ac	802.3ba	
Year	1983	1995	1999	2003	2010	~2017
Throughput (Gbit/s)	0.01	0.1	1	10	100	400

In most cases, the Transmission Control Protocol (TCP) and the Internet Protocol (IP) are used for communication on top of Ethernet networks, i.e., used for the transport and the network layer in the OSI model. For HPC communication, much lower latency is needed, than store-and-forward routers can offer and therefore the IBTA introduced another standard, as an appendix to the IB specification [61]: The RDMA over converged Ethernet (RoCE) specification [58] in 2010. By now, this is also available in a second version [60], which is no longer based directly on the Ethernet Protocol but rather on the User Datagram Protocol (UDP). RoCE is defined to use the same verbs as IB, introducing a good basis for portability of applications. Still, one of the main problems of Ethernet based networks is the latency.
After having described the two most common interconnects, the following subsection will describe one of the most important proprietary networks, the Cray XC series network.

Cray Interconnection Networks

Some of the most important proprietary HPC interconnection techniques are those developed by Cray™, as these account for approximately 10% of the interconnects in the TOP500 list, as of June 2016 [100]. Similar to the InfiniBand Architecture Specification, the Cray™ networks

define almost the whole range associated with a network: the cables, the hardware, the routing techniques and the network topology. The newest Cray™ interconnection technology is the Cray XC series network [3] integrated in the Cray XC distributed memory systems and often called Aries™ network.
The main idea behind the development of this network is to have a high global bandwidth while at the same time being very cost effective. The developed *Dragonfly* [69] network topology is thus a direct network topology, eliminating the cost for top level switches, as we would see in switched networks like the fat tree network topology. Considering the goal of cost effectiveness, the lowest layer consists of low cost electrical links, connecting the NICs and the local routers. Each router is connected to eight NICs, which in turn are connected to four nodes in a blade. These local routers are then in turn again grouped together. The dragonfly topology itself does not give any restrictions on the number of routers to be grouped together. The Cray™ Aries™ network includes 16 Aries™ routers in one group called a *chassis*. This group is connected by a chassis backplane and several chassis (in the XC network, six chassis) are then again connected to form one large group. This last connection is made through active optical cables. This can be seen in Fig. 2.8.

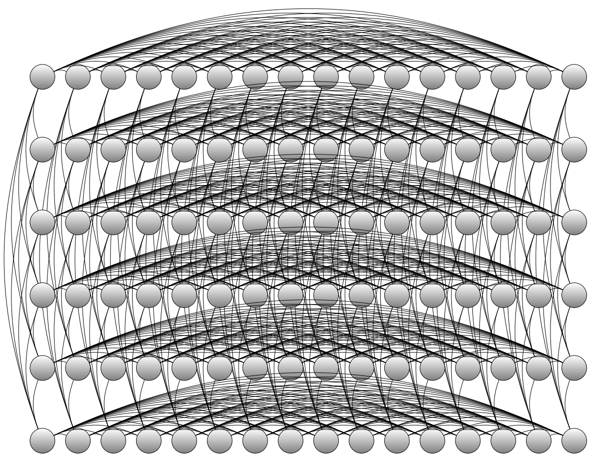


Figure 2.8.: A Cray XC network group, consisting of 6 chassis with each 16 Aries routers. Each node in the graphic resembles one router to each of which 4 nodes are connected.

In the TOP500 list of June 2016 [101], 50% of the top ten systems have a Cray interconnect, underlining the importance of this network type in HPC. The only German system included in the top ten also has this interconnect: the Hazel Hen in Stuttgart [51]. Besides the described networks, also other (proprietary) networks can be found in the TOP500 list.

Other HPC Interconnects

The most important interconnects have already been described above: IB and Ethernet having the largest share of systems in the TOP500 and Cray networks, interconnecting half of the top 10 systems. Apart from that, there are also other interconnection possibilities, especially in the top ten list. The top systems are interconnected with very specialized interconnects or topologies, not used by many HPC systems due to their high cost. These highly customized or proprietary interconnects include the IBM custom interconnects in the BlueGene/Q systems [76], the TH Express-2 [96], the Torus Fusion (Tofu) interconnect [2] and the Sunway interconnect [23]. One will not find many of these systems, as they are custom configured for exactly this one system, which is too expensive for most supercomputing facilities. One emerging interconnect is the Intel®Omni-Path Architecture (OPA) [62], which is already represented in the TOP500, but as delivery has only started in 2016, it might become more present in the upcoming TOP500 lists. One very important aspect of this architecture is the planned portability through an IB verbs API.

The first two parts of this chapter have dealt with hardware components relevant to HPC communication. The following section will got into the software of HPC systems and describe the communication routines and algorithms relevant to this thesis.

2.3. Communication

Data transfer plays an important role in distributed memory applications and is called communication. Depending on the participants of the communication, there are two main categories of communication: peer to peer communication and collective communication. Both will be briefly described here for an overview of communication possibilities, before going into more detail in Chap. 3.

Because all communication routines come in different flavors, and some of the following terms are used in a slightly different manner in different contexts. Therefore, the terms will here be defined as used throughout this thesis. These terms describe the behavior of any callable routine. One of the most important properties of the following communication routines is the question whether it is a blocking or a non-blocking routine.

Definition 2.12 *blocking and non-blocking*
A routine is called blocking, if the calling process stays in the routine until it has completed successfully.

A non-blocking routine may return control to the application before it has been completed successfully. In this case, the application needs to check on the successful completion at a later point of the program.

A finer notion of non-blocking is time-based blocking, introduced in the GASPI specification and described in Sec. 3.5. Another important property, often confused with blocking and non-blocking, is the synchronicity of a routine.

Definition 2.13 *synchronous and asynchronous*
A synchronous routine only achieves progress towards successful completion while the application is within the call.
An asynchronous routine may achieve progress towards successful completion even though it has returned control to the application.

This means, that every blocking routine is necessarily also a synchronous routine, but not every synchronous routine is also blocking. One more important property, especially when talking in terms of communication, is the question of locality. This definition is taken from the GASPI specification [32].

Definition 2.14 *local and non-local*
A procedure is local if completion of the procedure depends only on the locally executing process.
A procedure is non-local if completion of the operation may depend on the existence (and execution) of a remote process

All of these properties influence not only the implementation of the defined routines, but also the implementation of an application using these routines. Many communication routines come in different flavors regarding synchronicity and blocking behavior, but all are non-local.

2.3.1. Peer-to-Peer Communication

The first communication category, or pattern, is peer-to-peer communication. This kind of communication involves two processes: one source process and one destination process. In classical message-passing, this kind of communication is implemented through send and receive routines. Both processes are active in the communication and thus this send and receive scheme has a two-sided semantic. The message will be sent by the source process and will only be delivered at the destination process, if it has previously called a receive routine. This means that on both the source and the destination side some resources are bound and the CPU is involved in the communication.
Hardware and protocol developments have made RDMA communication possible, totally bypassing the CPU. This includes the above described IB, RoCE and Aries™. In case of RDMA communication only one process is actually active in the communication. The calling process initiates a routine that either transfers data from its own memory into a remote memory location, or the other way around. The origin memory of the transferred data will have affinity to one process, this being the source process. The destination process is the one, to whose share of the RDMA accessible memory the data is transferred. This kind of communication is called one-sided communication. Sometimes this term of one-sided communication is also extended to remote memory access (RMA) communication, as in the case of the MPI standard.

This communication comes in many more flavors, i.e., strided communication, where the data transferred is not a contiguous block, asynchronous, synchronous, as well as blocking and non-blocking communication. If it is necessary for a whole group of processes to exchange data, the application programmer may use collective communication routines, as described in the next section.

2.3.2. Collective Communication

Collective communication routines are routines where, opposed to peer-to-peer communication, a whole group of processes is involved in the communication. The collective communication routine can only finish successfully if all processes of a given group have entered the routine and - if applicable - have finished their share of work and transferred their share of data.
In this section some of the collective communication routines available in different parallel communication APIs or libraries are introduced. The selection does not show all collective routines available and those shown are only explained in a general manner, because different communication libraries or APIs may pose certain restrictions or add some features to the according routines defined in their specifications. This section describes the smallest common subset of all definitions.

Barrier

A barrier is a synchronization point in a program, that all processes (in a group if applicable) must call this procedure to continue. This may be useful whenever the programmer needs to make sure a certain part of the program has been reached by all processes before proceeding with the next part of the program, e.g., for timing. A barrier does not necessarily synchronize data, i.e., previously posted communication routines may not have completed the data transferal yet.

Broadcast

The broadcast routine is a collective communication which has a root. The root is one designated process in the group, which disseminates given data to all other processes in the group. At the successful finish of the call, all processes of the group are in possession of the data disseminated by the root process.

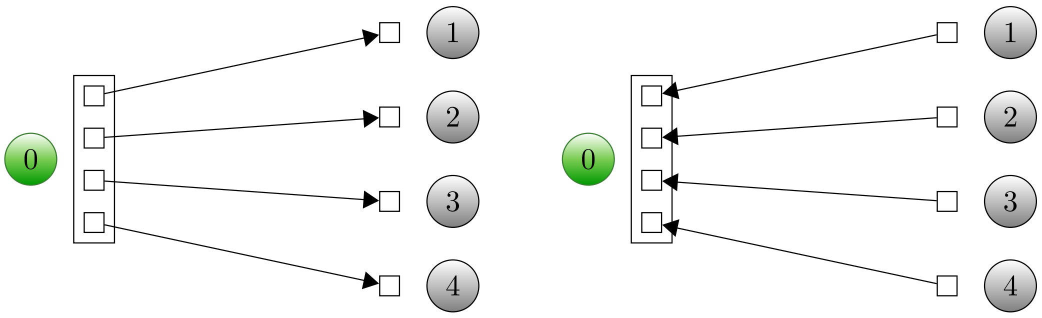
Reduce and Allreduce

The reduce is also a routine with a root process. Here, all processes in a group provide some data which is then reduced to one global result with a given reduction operation. The root process will have this global result upon return from the routine. The most common reduction operations are summation, the minimum and the maximum functions. The root

process accordingly obtains the sum, the smallest element or the largest element of all data elements provided.
A variation to the reduce function is the allreduce routine. Again all participating processes provide their data but in contrast to the above routine, there is no root process. Instead all processes will have the reduced result upon return from the procedure.

Gather and Scatter

The gather and scatter routines are both routines with a designated root process. In case of the gather routine, the root process receives data from all participating processes and after the routine has a sorted array of the data. In this array, the i^{th} element holds the data from rank i in the group, as depicted in Fig. 2.9b.



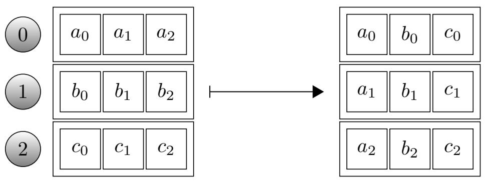
(a) Scatter. **(b)** Gather.

Figure 2.9.: Scatter and gather communication schemes.

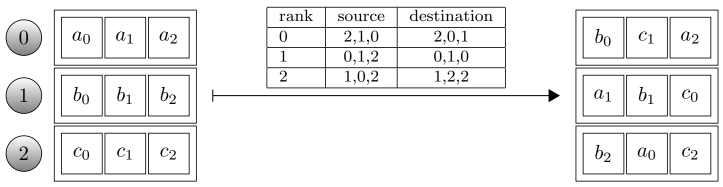
Scatter works the other way around (Fig. 2.9a): The root process has an array of data items and every participating process receives one of these items in the course of the procedure. Here, the i^{th} element of the root's source array will be transferred to rank i.

Alltoall and Alltoallv

In the alltoall routine, every rank has an array, which is distributed element-wise to the other participating ranks. In the simplest implementation, every rank sends the i^{th} element to rank i within the group, as depicted in Fig. 2.10a. This routine corresponds to a distributed matrix transpose, which will be further explained in Sec. 6.5.
In another notion of this routine, the programmer may give a different mapping for the distribution to the different ranks. By explicitly giving the offsets of the data elements in the source buffer and the in the destination buffer, the elements are distributed in a different manner. In Fig. 2.10b such an alternative mapping is depicted, where rank 0 communicates element 2 to offset 2 in its own destination buffer, element 1 of its source buffer to offset 0 in rank 1's destination buffer and element 0 to offset 1 of rank 2's destination buffer. Similarly, the other ranks have different mappings defined as shown in the table in Fig. 2.10b.



(a) Alltoall



(b) Alltoallv

Figure 2.10.: Classical alltoall communication scheme compared to an alltoallv communication scheme with adapted mapping.

These collective communication routines are only a small part of all collective routines already implemented in different communication APIs, above all in the MPI standard. But they are probably also the ones used most and especially those most important to this thesis. Because within these routines, communication between multiple ranks is necessary, different algorithms have been developed for different routines and platforms. The next section will give an overview of algorithms to frame a historical context with an emphasis on those relevant to this thesis.

2.3.3. Collective Communication Algorithms

The last sections have dealt with hardware basics of HPC systems and have introduced a number of collective communication routines. In this section, the emphasis lies on different algorithms to implement these collective routines. Special focus lies on those algorithms used for alltoall communication routines, like the barrier and allreduce routine. The tree-based algorithms in this section can also be used for one-to-all or all-to-one collective communication, like reduce and broadcast routines. An overview over a great part of these algorithms is also given in [49], focusing on the barrier operation.

Central Counter Algorithm

This barrier is described by Freudenthal et al. in [29]. For this barrier method, a global counter is necessary. This counter is held by one process and every process entering the barrier increments the counter by one through a "fetch and increment" routine. The process that incremented the counter to P then informs the other processes, that all processes have reached the barrier. Because all processes access the same shared variable or global counter, there are P serialized accesses, resulting in a lot of time spent waiting for other processes to have accessed

the variable and thus contention rises.
This algorithm is predestined for shared memory architectures and maybe for distributed memory architectures with global atomic operations. But because all ranks entering the barrier need to access the same variable, it is not very scalable. Hence, an implementation in times where thousands of processes may be involved in such a barrier does not make sense and more complex algorithms are needed to schedule the communication and balance the load.

Combining Tree Algorithm and Adaptive Combining Tree Algorithm

The combining tree algorithm, described in [104], is designed as an improvement of the central counter algorithm, to distribute the contention on one shared variable. In central counter algorithm, each process accesses the shared counter in the barrier phase, which leads to a so called hot-spot as described in [85]. Instead, this contention is distributed, by forming groups of processes, each having their own central counter. Each of these groups is assigned to a leaf of a three-level tree with a predefined fan-in, where again central counters are used to reach the root of the tree. Every group decrements the counter of the parent node. When it has turned to zero, the next parent node counter will be decremented and last but not least also the root counter will be decremented. Thus, the number of total accesses to shared variables has been increased, but the number of accesses to each counter and with that the contention has been decreased. As soon as the root knows, that all processes have entered the barrier, it informs the other processes by broadcasting the information.

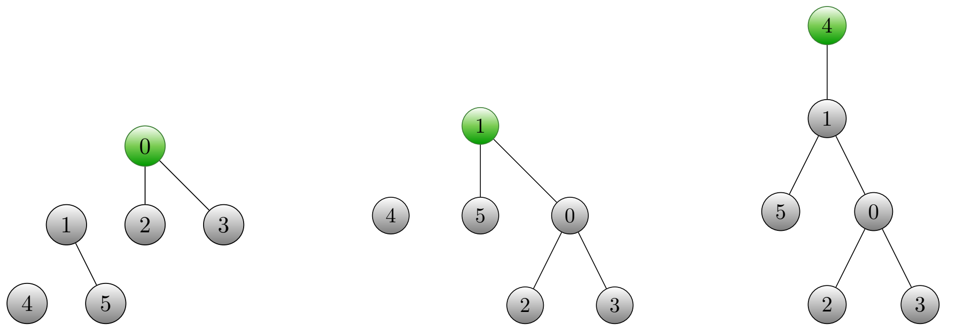


Figure 2.11.: The token is passed from rank 0 to rank 1 to rank 4 in the course of the adaptive combining tree algorithm.

An adaption to the combining tree barrier was presented in [41]: the adaptive combining tree barrier, shown in Fig. 2.11. Again, several child nodes acknowledge their arrival in the barrier at their parent node. As soon as the first acknowledgment has reached the root, it passes the “root token” down to one of the children, that has not confirmed its arrival in the barrier. The rank carrying the token will be the new root. This concept is carried on until the token has either reached a leaf or until the token has reached a node whose children have already acknowledged their parent and the new root then immediately releases all nodes out of the barrier.

Tournament Algorithm and n-way Tournament Algorithms

This algorithm was designed for usage in barrier operations and is described in [44]. It is divided into different communication rounds, needing $k = \lceil \log_2(P) \rceil$ rounds for completion. Always two processes are grouped together with one process being the so called winner of the round. The loser will inform the winner of his arrival in the barrier. In the following rounds, this principle is repeated, such that two winners of the former round are grouped together and the predefined winner of that round will be informed by the loser. If the number of processes involved, or the number of groups in one round is uneven, this communication is done in an extra step. When the overall tournament winner - often also called the root - is informed, that all processes have entered the barrier, it broadcasts this information to all other processes. The communication scheme is very similar to the binomial spanning tree scheme, described on p. 25.
Variations to the tournament barrier algorithm are given by Grunwald and Vajracharya, who described a static and a dynamic n-way tournament barrier in [40]. In these barrier algorithms, a group of n ranks has one winner, that is informed of the arrival of the other $n-1$ ranks in the group. In the following rounds, groups of up to n winners are formed to continue in the tournament. In the static version, the winners of each group and round are predefined, while in the dynamic version, the last rank of the group to arrive in the barrier will be active in the next round.
In all cases, the barrier can only be completed with a broadcast following the successful completion of the tournament barrier, adding communication rounds not mentioned here. Additionally, the dynamic version of the n-way tournament barrier is only suited for shared memory architectures with, e.g., a central counter per group, because otherwise it would involve too much additional communication to find out who is the last rank to continue in the tournament. Alternatively, atomic operations like "fetch and add" could be used to implement the dynamic, n-way tournament algorithm. But this would again introduce some overhead in sequentializing the access to the atomic counter.

Binomial Spanning Tree Algorithm

The binomial spanning tree (BST) is described in many different papers, for example in [49] as well as in [102]. The principle of the binomial spanning tree is the same, as for all trees: all ranks first need to inform the root of their arrival in the barrier and afterwards the root releases all ranks out of the barrier by broadcasting the information.
The binomial spanning tree is built by first numbering all ranks 0 to $P-1$. These are then assigned to tree nodes by representing the ranks in binary numbers and following these two rules: 1. rank 0 is the root of the tree, and 2. the children of the processor with rank p_0 are those with rank $p_0 + 2^i$, where $\log_2(p_0) \leq i \leq \lceil \log_2(P) \rceil$.
The parent of each process is then found by flipping the leftmost 1-bit of the binary repre-

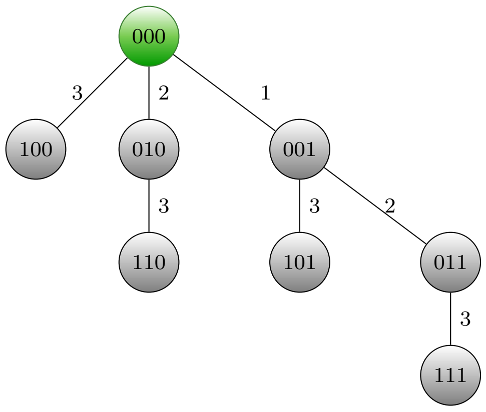


Figure 2.12.: A binomial spanning tree for 8 ranks. The small numbers at the edges of the tree indicate the order in which the barrier will be released again at the end.

sentation of the rank. Thus a BST with $P = 8$ processors would lead to the tree depicted in Fig. 2.12.

In the first phase of the barrier, every node waits for its children to reach the barrier. As soon as this happens, this node will inform its parent node and so on until the root is reached. When the root has learned from all its children that they have reached the barrier, it will sequentially inform its children in the order of the numbers next to the edges. Thus messages along edges can be seen to travel over the network at the same time, i.e., after three communication rounds, all ranks are freed from the barrier.

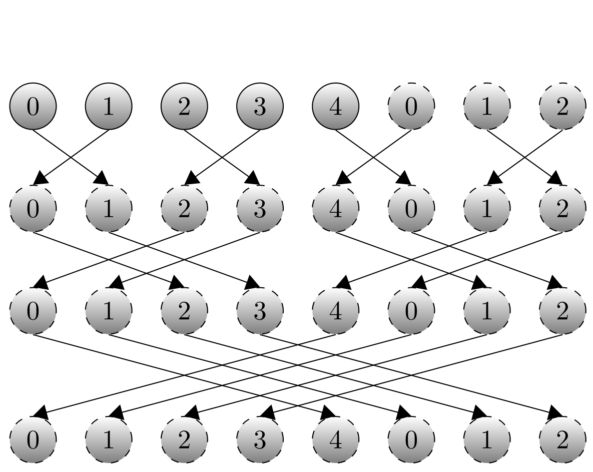
Butterfly Algorithm and Pairwise Exchange Algorithm

The butterfly algorithm is described by Brooks in [7] and is designed for a barrier with $P = 2^k$ participating ranks, needing $k = \log_2(P)$ communication rounds for completion. In each round $l \in \{0, \ldots, k-1\}$ processor p_0 writes information to

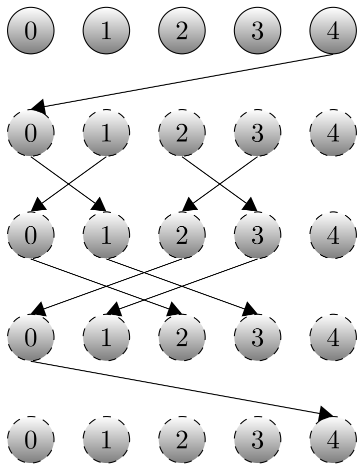
$$s_l = \begin{cases} p_0 + 2^l, & \text{if } p_0 \mod 2^l < 2^{l-1} \\ p_0 - 2^l, & \text{if } p_0 \mod 2^l \geq 2^{l-1}. \end{cases} \tag{2.2}$$

In each round, the message transferred to the communication peer not only informs the peer of the arrival of p_0 in the barrier, but also of the arrival of all ranks that have previously informed rank p_0.

If $P \neq 2^k$, but rather $P = 2^k - q$, then the butterfly barrier can be adapted, such that the first l ranks additionally act as virtual ranks $P, P+1, \ldots, 2^k - 1$ and the above described scheme is applied on the total 2^k ranks. This is depicted in Fig. 2.13a, where it can also be seen, that the first l ranks will be involved in more communication than the other ranks due to their additional role as virtual ranks. This leads to a very unbalanced work distribution, where the



(a) Butterfly.



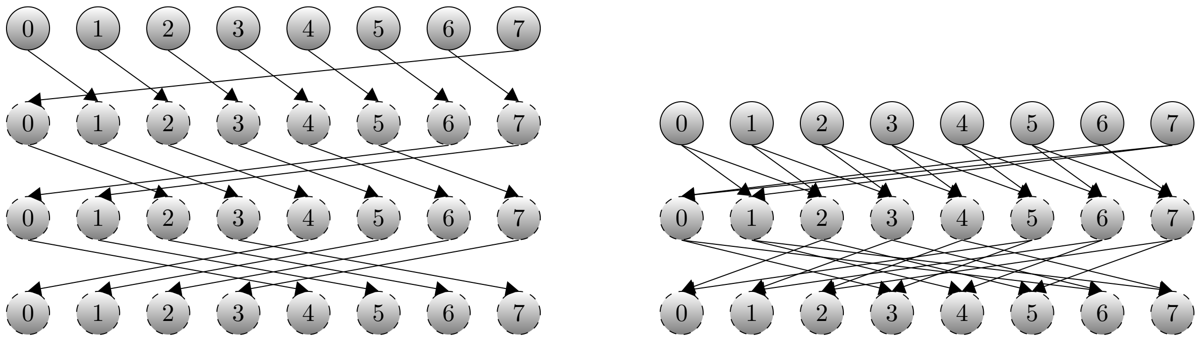
(b) Pairwise exchange.

Figure 2.13.: Direct comparison of the butterfly algorithm communication scheme for $P = 5$ and that of the pairwise exchange algorithm.

ranks without a dual role might have a lot of idling time.
A different approach to adapting the butterfly barrier for $P \neq 2^k$ is done through the pairwise exchange (PE) algorithm, described for example in [42]. If the number of processes is not a power of two but rather $2^k + q$, then the first 2^k ranks perform a standard butterfly algorithm, after the q ranks $2^k, \dots, P-1$ have transferred their information to the first q ranks. After the butterfly algorithm communication is done, the first q ranks again communicate with the last q ranks and the algorithm is finished, as depicted in Fig. 2.13b.
This algorithm needs one round more for 5 processes than the butterfly algorithm but does not have any multiple synchronization or a duplication of communicated data. In the example with 5 ranks, the number of messages transferred decreases significantly from 32 in the adaption with virtual ranks to 10 in the pairwise exchange algorithm. On the other hand, rank 4 in Fig. 2.13b is idle during the butterfly communication phase of the first 4 ranks.

Dissemination Algorithm and n-way Dissemination Algorithm

The dissemination algorithm is described in [44]. It was originally designed for the use in barriers to disseminate the information which ranks have already reached the barrier in an efficient manner, i.e., such that not every rank has to inform every other rank about its arrival in the barrier. It has been designed so well, that it is still used today in various barrier implementations [6, 75, 28]. The barrier is executed in different communication rounds, where the number of rounds k is dependent of the number of processes involved. Let P be the number of processes involved, then $k = \lceil \log_2 P \rceil$. Further, let p_0 be an arbitrary but fixed rank participating in the barrier. In every round l, process p_0 sends a message to process $s_l = (p_0 + 2^l) \mod P$ and after doing so, waits for the message of process $r_l = (p_0 - 2^l) \mod P$. In each round, the received message not only states, that the source process has reached the barrier, but also that certain other ranks have reached the barrier in the preceding rounds.



(a) dissemination algorithm **(b)** 2-way dissemination algorithm

Figure 2.14.: Comparison of the 1-way dissemination algorithm and the 2-way dissemination algorithm communication schemes with 8 participating ranks.

Figure 2.14a shows the communication scheme of the dissemination algorithm for 8 processes. The algorithm will need $\log_2(8) = 3$ communication rounds to notify all processes that all other processes have reached the barrier. Going through the scheme for rank 0, it will be informed by rank 7 in round 1 that it has reached the barrier. In the second round, rank 0 will receive information from rank 6, this directly implying that rank 5 has also reached the barrier, because rank 5 needed to notify rank 6 in the first communication round. In the third communication round, rank 0 will receive the information that ranks 1 through 3 have reached the barrier from rank 4. This way, rank 1 is informed that all other ranks have reached the barrier with only three messages.

The n-way dissemination algorithm is an extension to the dissemination algorithm, presented by T. Hoefler et al. in [48]. Also for this algorithm, several communication rounds are necessary, in which the participating ranks transfer a snippet of information to n other participating ranks. Let n be the number of messages transferred in every communication round and P be the number of ranks involved in the collective communication. Then $k = \lceil \log_{n+1}(P) \rceil$ is the number of communication rounds the n-way dissemination algorithm needs to go through, before all ranks have the same information. In every communication round $l \in \{1, \ldots, k\}$, every process p has n peers $s_{l,i}$, to which it transfers data and also n peers $r_{l,j}$, from which it receives data:

$$s_{l,i} = p + i \cdot (n+1)^{l-1} \mod P \tag{2.3}$$

$$r_{l,j} = p - j \cdot (n+1)^{l-1} \mod P, \tag{2.4}$$

with $i, j \in \{1, \ldots, n\}$. Thus in every round p gets (additional) information from $n(n+1)^{l-1}$ participating ranks - either directly or through the information obtained by the source ranks

in the preceding rounds. Figure 2.14b exemplarily shows this communication scheme for the receiving rank 0 and a 2-way dissemination with eight ranks.
In the first round, rank 0 obtains the information from ranks 6 and 7. In the second round, rank 5 notifies rank 0 of the arrival of ranks 3 and 4 and the message from rank 2 carries the information of ranks 0 and 1. Even though rank 0 already knows, that it has reached the barrier, rank 2's message includes this information, because it was informed by rank 0 in the previous round. In the case of a barrier, this repeated information does not play a role, but Chap. 4 will deal with this detail in a more exhaustive manner. An additional adaption to the algorithm will be presented in that chapter.
Compared to the dissemination algorithm in Fig. 2.14a, the 2-way dissemination algorithm only needs two communication rounds to obtain the information, that all ranks have entered the barrier. At the same time, more messages are transferred to disseminate the information. It will depend on the size of the messages and the level of implicit parallelism of the network, whether it will be more efficient to wait on messages in three rounds, potentially introducing a lot of idle time, or whether two messages can be transferred over the network in nearly he same time as one message.

Bruck's n-port Global Combine Algorithm

In [9], Jehoshua Bruck and Ching-Tien Ho present two algorithms for global combine operations in n-port message-passing systems[1]. The first of the two shows many similarities to the n-way dissemination algorithm presented above. While the dissemination algorithm and the n-way dissemination algorithm were both designed for barrier operations, Bruck's algorithm is explicitly designed for global combine operations, i.e., allreduces.
In $\lceil \log_{n+1}(P) \rceil$ communication rounds, every participating process transfers and receives n partial reduction results from other processes. Let $\circ$ be the reduction operation used and x_p be the initial data of process p. The partial results transferred by rank p in round l are computed in two versions: $S_l^p[0]$ is the reduction of all previously received results without the initial data of the computing process and $S_l^p[1] = x_p \circ S_l^p[0]$. In each round, the group of destination ranks is split up into two groups, one of which will receive S_p^0, and the other will receive S_p^1. For determining these groups, two things are necessary: the base $(n+1)$ representation of $P-1$ and the counter c, which counts the number of elements on which the reduction has already been performed.
For ease of readability, the algorithm will here be described with the help of an example for $P = 8$ and $n = 2$ from the view of rank 0. The complete communication scheme for this example is depicted in Fig. 2.15. The general description and the proof can be found in [9]. The algorithm will need $k = \lceil \log_3(8) \rceil = 2$ communication rounds. For each of these rounds l, an α_{l-1} is needed to split the destination ranks in two groups: one receiving $S_l^p[0]$ and the other

[1]The notation of the original paper has been adapted to fit the notation throughout this thesis.

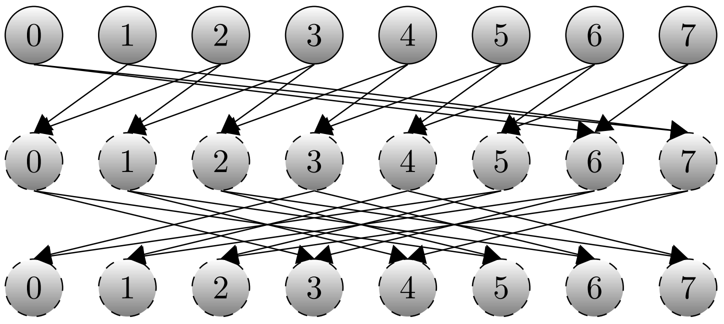


Figure 2.15.: Communication scheme of Bruck's global combine algorithm for $P = 8$.

$S_l^p[1]$. These α_i are computed through the representation of $P - 1 = 7$ in a base 3 notation:

$$7 = (21)_3 = (\alpha_1 \alpha_0)_3. \tag{2.5}$$

In the first round, only the partial result $S_1^0[1] = x_0$ is transferred to $\alpha_{k-1} = \alpha_1 = 2$ process. The destination processes are

$$s_{1,1} \equiv \quad p - 1 \mod (P) \equiv -1 \mod 8 \quad \equiv 7 \tag{2.6}$$

$$s_{1,2} \equiv \quad p - 2 \mod (P) \equiv -2 \mod 8 \quad \equiv 6. \tag{2.7}$$

Rank 0 will simultaneously receive partial results from its peers $r_{1,1} \equiv p + 1 \mod (P) \equiv 1$ and $r_{1,2} = 2$, namely $S_1^1[1] = x_1$ and $S_1^2[1] = x_2$. Rank 0 can then calculate new partial results to be transferred in the following round:

$$S_2^0[0] \quad = S_1^0[0] \circ S_1^1[1] \circ S_1^2[1] \quad = x_1 \circ x_2 \tag{2.8}$$

$$S_2^0[1] \quad = S_1^0[1] \circ S_1^1[1] \circ S_1^2[1] \quad = x_0 \circ x_1 \circ x_2. \tag{2.9}$$

At the same time, c is increased to $c = \alpha_1 = 2$, which will be needed for the computation of the communication peers in the next round. Rank 0 will now transfer $S_2^0[1]$ to $\alpha_0 = 1$ rank:

$$s_{2,1} \equiv p - \alpha_0 \cdot (c + 1) \mod (P) \equiv -3 \mod 8 \equiv 5, \tag{2.10}$$

and $S_2^0[0]$ to the remaining $n - \alpha_0 = 2 - 1 = 1$ rank:

$$s_{2,2} \equiv p - c - \alpha_0 \cdot (c + 1) \mod (P) \equiv -5 \mod 8 \equiv 3. \tag{2.11}$$

In the same round, rank 0 will receive partial results from ranks

$$r_{2,1} \equiv \qquad p + (c + 1) \mod (P) \equiv 3 \mod 8 \qquad \equiv 3 \tag{2.12}$$

$$r_{2,2} \equiv \quad p + c + \alpha_0 \cdot (c + 1) \mod (P) \equiv 5 \mod 8 \quad \equiv 5. \tag{2.13}$$

Then, rank 0 can compute the final result

$$S_3^0[1] \quad = S_2^0[1] \circ S_2^3[1] \circ S_2^5[0] \quad = x_0 \circ x_1 \circ x_2 \circ (x_3 \circ x_4 \circ x_5) \circ (x_6 \circ x_7). \tag{2.14}$$

Important to note is, that the order of applying the reduction operation is also defined through the algorithm. Thus at the end, every rank will have the initial data elements reduced in the same order, but not in the same associative order.

While Bruck's algorithm is the only one presented here, that is actually designed for being used in an allreduce operation, all of the tree- and butterfly-based algorithms may be used in an allreduce operation. The tree-based algorithms can be used exactly as described, but will have to transfer some more data and a computation step has to be built in between receiving data from the children and forwarding the partial result to the parent. As soon as the root has received all partial results and computed a final result, it can broadcast the data down the tree. Other allreduce algorithms, like those described in [98] are not of interest for GASPI collective communication routines, as will be discussed in more detail in the summary below.

2.4. Summary

In this chapter, different aspects and components of an HPC system have been introduced. All are needed in order to design performant parallel application running on one of these systems. It starts with the hardware setup of the system, i.e., the memory (Sec. 2.1) and interconnects (Sec. 2.2). An application programmer has to know, whether the underlying system is a shared memory, a distributed memory or a hybrid system to be able to select the correct programming and communication paradigm for the application. In addition to this, the interconnect plays an important role in distributed and hybrid systems. Depending on the interconnect, different communication models are feasible. Whether only message-passing or even RDMA is possible plays an important role in the design of an application and of course also in the design of communication routines.

The difference between peer-to-peer communication and collective communication has been described in Sec. 2.3, emphasizing on collective communication. Different collective communication routines relevant to this thesis have been presented in Sec. 2.3.2, along with possible algorithms for their implementation in Sec. 2.3.3. It started with simple, counter-based barrier algorithms, that are absolutely not scalable and intended especially for shared memory systems. The next step were tree-based algorithms with multiple counters and from there on more and more complex algorithms for the dissemination of information have been developed. All of the presented algorithms, except for Bruck's algorithm, have been designed for barrier operations. But some of them can also be used for allreduce operations, by simply adding a computation step between two communication steps. Others, like the n-way dissemination algorithm would lead to erroneous results when used for an allreduce. Hence, in Chap. 4, an adaption to the

n-way dissemination algorithm will be presented. The different algorithms will then again play a role in Chap. 5, where they are needed to implement collective communication routines for GASPI_COLL.
Before that, the next chapter will deal with state of the art communication libraries and APIs already used in HPC.

3. HPC Communication Libraries and Languages

As already mentioned in Sec. 2.1, different memory architectures need different communication paradigms to be used efficiently. In addition to these different paradigms, different interconnects (Sec. 2.2) come with different interfaces, which will be called low-level languages in the following. These low-level languages, like the InfiniBand Verbs (IB Verbs) [61, 73], Distributed Memory Application (DMAPP) [18, 97] and also Global Address Space Networking (GASNet) [6, 31] are intended to be used as a basis for communication libraries and programming languages by developers. These latter programming languages are intended to be used by the end-user, i.e., the application programmer and are called high-level languages. The different factors, like memory architecture, interconnect hardware and low-level languages, influence the implementation of high-level programming languages like C [68], Fortran [4], Python [88, 89], C++ [14] or Java [37, 82], to only name a few.
This chapter will introduce some of the HPC communication libraries, languages and interfaces relevant to this thesis. Section 3.1 gives a historic overview of the development of the different communication paradigms and languages. Section 3.2 will describe two low-level interfaces (GASNet and IB Verbs), the shared memory API OpenMP and additionally give an overview on different PGAS languages and approaches. The following sections are then dedicated to single APIs: MPI in Sec. 3.3, Global Programming Interface (GPI) in Sec. 3.4 and GASPI in Sec. 3.5.

3.1. Historic Overview

In Sec. 2.1, different memory architectures have already been introduced, and then in Sec. 2.2 different possibilities to connect distributed memory architectures have been described. It has been stated that both influence the development of programming paradigms and languages and this section will now give a brief overview over the historical development of parallel programming communication interfaces, where possible directly linked to emerging hardware trends of the time.
In the early 1980's, many different message-passing systems were developed, many of which formed the basis for MPI. The goal of MPI was to design an application programming interface for efficient communication and created a new message-passing interface to combine the best

features of, e.g., Intel's NX/2, NCUBE's Vertex, PVM and PARMACS [75]. The NX/2 operating system was designed explicitly for the Intel iPSC2 and supported multitasking as well as communication [87]. Similar, the NCUBE ran Vertex on its nodes, which already used direct memory accesses for message transfers [43]. PARMACS was already a machine-independent implementation of a message-passing system, working with macros and library functions for message transferal. One important feature of PARMACS was the possibility of mapping of processes on the compute nodes with different predefined topologies [22]. PVM offered several point-to-point communication routines, as well as broadcast and barrier operations. According to Dekker, Smit and Zuidervaart, the performance of PVM was already so poor in 1994, that they predicted the replacement of PVM with MPI ([22], p. 9). Remarkably, this was the year the first MPI standard was published [74].

Only one year earlier, in 1993, the High Performance Fortran (HPF) standard was released [91], a data-parallel language extension to Fortran. HPF aims at a global address space with distributed data structures and implicit communication, tempting De Wael et al. to classify HPF as a retrospective PGAS language, i.e., one that emerged before the term PGAS was established [21]. Even before that, Cray implemented Cray Research Adaptive Fortran (CRAFT) for its T3D system, but had only mediocre success ([83], p. 437). CRAFT provided a global address space (GAS) memory model, with private objects or shared objects. While shared objects are are accessible by all processes and may even be distributed, if the object is an array, private objects are always stored on the process-local memory and may not be accessed by other processes [84]. A similar memory model was implemented on the BBN TC 2000 architecture as early as 1991, where memory was divided into local and private memory, shared interleaved memory and team-private memory, i.e., distributed memory shared among a team or subset of processes [8]. Even though there had been many notions of GAS languages - that even partitioned their shared memory - by 1994, the term PGAS was not directly associated to any of these languages until the late 1990's, when the specifications of UPC and CAF were published [21]. The next large advance in PGAS interfaces was made around 2004, when Chapel, X10 and Fortress were developed in the scope of the High Productivity Computing Systems (HPCS) project [24].

MPI was successful right from the beginning, bringing together the best of many previous message-passing interfaces and having a large support from the community ([83], p. 295). Many parallels can be drawn to the development of PGAS interfaces. Before the Message-Passing Interface Forum (MPIF) standardized message-passing, there were many different approaches, all being accepted and used by a certain user group. Now the same can be observed in the PGAS universe: there are many different languages and interfaces, all specified for a certain use-case or platform. It is to be expected, that in near future, either MPI will include more functionalities that are now distinct to PGAS (as it is already doing by extending and refining the one-sided and RMA communication routines [75]), or a new interface will be developed, uniting the best of the different PGAS interfaces, languages and libraries.

3.2. Related Communication APIs

This section is going to introduce several different communication APIs, languages and libraries. First, the low-level languages GASNet and the IB Verbs will be introduced. GASNet is a low-level language already used for, e.g., the PGAS language UPC while IB Verbs is the interface used for IB networks and in use in the GASPI implementation worked with for this thesis. Due to the distribution of an IB Verbs interface in the OFED stack, the interface is gaining more and more popularity also for other interconnects, e.g., RoCE (p. 17) and Omni-Path (p. 19). An overview of OpenMP is given, as this is the most commonly used shared memory library, often in combination with distributed memory communication like MPI or GASPI. Other PGAS languages will be shortly described in the last subsection.

3.2.1. IB Verbs

The IB Architecture Specification [61] defines a set of Verbs that a compliant installation of the IB network needs to implement, but, unlike MPI or GASPI, it does not define the actual interface. The semantic of the Verbs is described in the specification, but the naming of the routines and implementation details are left to the developers and vendors. The implementation distributed with the OFED stack, is wide spread and available on most, if not all, clusters with an IB interconnect. To clarify between the Verbs in general and this implementation, the term IB Verbs will be used when speaking generally and ibverbs will be used when referring to this special implementation. As of June 2016, more than 40% of the TOP500 list of supercomputers have an IB network connecting the single nodes of the compute cluster, underlining the importance of this network [100]. With this wide spread availability, the decision to implement a highly performant, open source GASPI version for this network (GPI2 [28]) does not come as a surprise. While there is also a Mellanox specific implementation of the IB Verbs [73], this section will concentrate on and take examples from the OFED implementation ibverbs [70], because this is the version that the GPI2 builds on.
The IB architecture makes communication without involvement of the CPU or operating system (OS) possible. This is done by creating communication channels, with queue pairs (QPs) at each end of the channel. Each QP consists of a send queue and a receive queue managing according work requests from the application. In addition to this, it is also possible to create completion queues (CQs), on which the completion of previously issued work requests can be polled. To enable the network of RDMA operations, the memory regions that will be read from or written to need to be registered. When registering these memory regions, keys are generated to distribute to the remote nodes that need access to these memory regions.

```
struct ibv_send_wr {
  uint64_t                wr_id;
  struct ibv_send_wr     *next;
  struct ibv_sge         *sg_list;
  int                     num_sge;
  enum ibv_wr_opcode      opcode;
  int                     send_flags
  uint32_t                imm_data;
  union {
    struct {
      uint64_t            remote_addr
      uint32_t            rkey;
    } rdma;
    struct {
      uint64_t            remote_addr;
      uint64_t            compare_add;
      uint64_t            swap;
      uint32_t            rkey;
    } atomic;
    struct {
      struct ibv_ah      *ah;
      uint32_t            remote_qpn;
      uint32_t            remote_qkey;
    } ud;
  } wr;
};
```

Listing 3.1: The send work request struct as defined in the IB Verbs distribution of the OFED stack [70].

Through the communication call `ibv_post_send()`, one or more work requests are posted to a QP. Such a work request, as implemented in ibverbs, is shown in List. 3.1. IB offers different methods of data transferal, message-passing as well as RDMA transfers. The `opcode` defines the kind of communication requested by a specific request. The following opcodes are defined in ibverbs: RDMA_WRITE and RDMA_WRITE_WITH_IMM for RDMA write operations, RDMA_READ for RDMA read operations, SEND, SEND_WITH_IMM for send operations and ATOMIC_CMP_AND_SWAP, ATOMIC_FETCH_AND_ADD as atomic operations.
Depending on the `opcode` set, the remote side information needs to be given by the user in one of the structs `rdma`, `atomic` or `ud`. So for different kinds of communication, different prerequisites have to be fulfilled: for send operations, the `ud` struct has to be set, i.e., an address handle (`ah`), a QP (`remote_qpn`) and a `remote_qkey` has to be known by the application. All of these are known from the initialization of the QP. For RDMA communication, the application needs to know an address (`remote_addr`) and a key for the according remote memory region (`rkey`). The key for the memory region is exchanged, when the remote memory region is registered to the initiating process, a necessary prerequisite for RDMA communication in IB.

The scatter/gather array `*sg_list`, contains the local addresses to write from or read to. It is possible to have multiple scatter/gather elements in one work request, but the order in which the list of elements is processed is not defined. On the other hand, the ordering of multiple work requests is very well defined. As the InfiniBand Architecture Specification Vol. 2 [61] states in chapter 9.5, p. 280:

> C9-25: A requester shall transmit request messages in the order that the Work Queue Elements (WQEs) were posted.
>
> C9-26: For messages that are segmented into PMTU-sized packets, the data payload shall use the same order as the data segments defined by the WQE.
>
> Packets from a given source QP to a given destination QP travel on the same path through the fabric and are received in the same order they were injected.

This means that the ordered arrival of two read requests is guaranteed, which will be used for the implementation of the notified read in Sec. 6.2.
An additional property, that a work request may have, is given through the `send_flag`. With this flag, the send request can be made fenced, signaled, solicited or inline. The inlined flag is defined by ibverbs and enables the usage of unregistered memory regions for RDMA writes and sends. A solicited write or send will wake up a potentially waiting remote side and a fenced work request will only be processed when all previously posted RDMA and atomic work requests have been completed. The latter is only available in queues that establish a reliable connection. Especially interesting is the notion of a signaled work request, because this will create a work completion when finished. This work completion will be posted in the CQ, which can then be polled by the application. Only when the application knows, that a work request has been completed, it can safely reuse the associated buffers.
This has been a very superficial overview of the IB Verbs and the functionality of the IB Software Transport Interface, but it should suffice for the scope of this thesis. The next section will describe another low-level interface: GASNet. Even though GASNet is designed as a low-level interface, i.e., to be used for the development of interfaces or libraries and not by an application programmer, it is built on top of different interfaces, including the IB Verbs.

3.2.2. GASNet - Global Address Space Networking

GASNet is a highly portable communication network layer designed by Dan Bonachea for the PGAS. It is based on the Active Messaging Interface [71] and described in [6]. It has been designed as a layer between the hardware layer and the actual API, i.e., a low-level language. This section will give an overview of the functionalities of GASNet but without detailed information on single routines not directly associated with communication. All statements made in this chapter that excess the scope of the specification, especially when referencing any readme files, refer to the GASNet release 1.18.2 and included documents obtainable online [31].

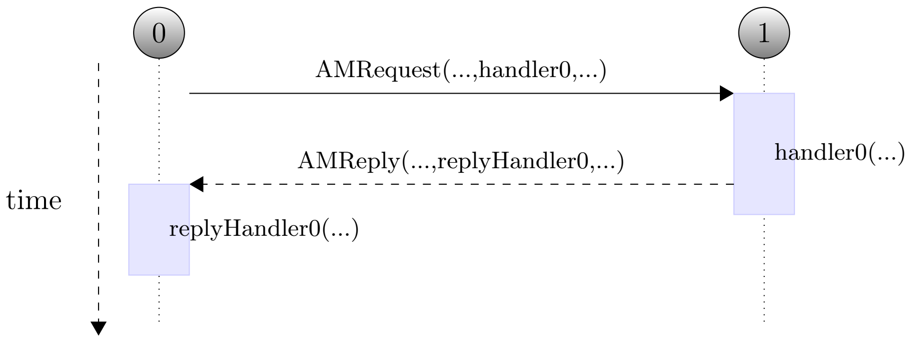


Figure 3.1.: The active messaging communication scheme. The dashed reply and its invoked reply handler are not mandatory but must be sent by handler0.

The GASNet programming interface is divided into two APIs: the Core API, which is the minimum interface that every implementation of GASNet has to deliver, and the Extended API, which is more user-friendly and may be implemented solely in terms of the Core API. During the configuration of GASNet, the machine on which it will be installed is checked by a script and according to the result of the tests, one or more of twelve conduits will be installed. These conduits ensure the portability of GASNet to different platforms while still reaching highest possible performance. To achieve maximum performance in terms of runtime saving and resource friendliness the GASNet routines are implemented in a conduit specific manner. GASNet routines can thus be seen as a wrapper around given hardware specific languages such as IB Verbs. As it would lead to too much detail to explain every single conduit, a list can be found in App. A. For further reference, please see the readme files included in the different conduits [31].
The Core API consists of the two-phased initialization with the two routines `gasnet_init` and `gasnet_attach`, the exiting routine `gasnet_exit`, environment queries and several active messaging routines. There are also atomicity control routines, which enable a race condition free memory access.
The initialization routines bootstrap the job to the computation nodes, allocate a global memory segment on each node and establish the communication network between the nodes. The global memory segments will then be accessible by all compute nodes as shared memory. The established communication network is of an alltoall character, such that every node can afterwards communicate with any other node which has been bootstrapped.
The communication routines of the Core API consist of different active messaging routines. An active messaging communication typically consists of a request message sent by node p_i, invoking a handler on node p_j, and an optional reply message sent by node p_j, again invoking a handler on node p_i. The initiated handler on the other node then processes the sent data, giving this type of communication the name Active Messages. This communication scheme is depicted in Fig. 3.1.

To steer the workflow of a GASNet application, a polling routine and a blocking routine also belong to the Core API. Both query, whether a certain user-defined condition has been fulfilled, the latter blocking the CPU until the condition is fulfilled.

The Extended API of GASNet defines more user-friendly routines than the Core API does, e.g., there are no longer different communication routines for different message sizes. Instead, the message size is taken as an input argument to the communication routines of the Extended API. It includes memory-to-memory transferring routines, which are available in a blocking and a non-blocking manner. For the non-blocking routines there are also corresponding synchronization routines. Additionally there are register-memory operations, also in non-blocking and blocking versions, which enable a better use and control of memory resources. There is also a split-phase barrier to synchronize the threads in their workflow. As already implied through the last sentence, the Extended API also supports threading.

The blocking put and get functions will write the data into the remote memory segment immediately, meaning before the function returns. The non-blocking routines on the other hand will return immediately but the application programmer has to ensure that the data has been written before using it again by calling a synchronization routine. Every non-blocking routine comes in two versions: an explicit handle version and an implicit handle version. These handles - passed as arguments - become important when synchronizing.

In case of the explicit handle version, the programmer has the possibility to only synchronize those routines, which were started with a certain handle. This gives the programmer more flexibility and makes it possible to minimize synchronization points by only synchronizing when the according data is actually needed. Another possibility for the programmer to avoid synchronizing at undesired points is available by choosing to wait only for puts, gets or actually waiting for all memory-transfer routines to have finished.

If the implicit handle versions are used, any implicit synchronization call will wait until all data is written. If not all implicit calls shall be synchronized, but only a subset, the GASNet specification supplies the so called access regions. These access regions return a handle, which can then be used for synchronizing all implicit handle communication calls made in this region.

For the synchronization of the non-blocking calls, the programmer may choose between the `gasnet_wait_...` calls and the `gasnet_try_...` calls. The latter only checks if all calls in question are done and returns immediately. If all calls are have completed successfully, `gasnet_try_...` will behave like `gasnet_wait_...`, which would block until all routines in question have returned.

These synchronization routines should not be confused with the split-phase barrier also included in the Extended API. Every node calling `gasnet_barrier_notify` notifies the system, that it has reached a certain point in the application. It is possible to hand a token to the routine as an argument, which affects the second phase of the barrier. This second phase may be called in a blocking manner as `gasnet_barrier_wait`, which will wait until all nodes have notified

the system, or in a non-blocking manner through `gasnet_barrier_try` which only checks if all nodes have notified the system and may be called several times after one another. This barrier has nothing to do with the synchronization of data and thus it may not be assumed, that all write and/or put operations called before the barrier have completed.
GASNet and IB Verbs have been two low-level interfaces for distributed memory systems with the possibility of creating a PGAS through their RDMA routines. Both may be used as an underlying layer for an implementation of higher level languages, like GASPI. The API described in the next section, is a shared memory API, which is often used in combination with distributed memory languages like GASPI or MPI on hybrid systems.

3.2.3. OpenMP

OpenMP [81] is one widely spread possibility of a shared memory programming paradigm used together with MPI for hybrid programs, i.e., programs running on a hybrid architecture with distributed and shared memory (see p. 7). Within shared memory programs, the user controls multiple threads, implicitly communicating with each other through shared variables. In OpenMP, the user controls the threads and parallel regions through precompiler directives. This means, that OpenMP implementations need to be included in the compiler used. This has the nice side effect, that each program only needs to be written once, but can either be compiled in a parallel version, when setting the according compiler flag, or as a sequential program. OpenMP is interoperable with MPI and GASPI and has been used in some benchmarks presented in this thesis. A very short introduction on OpenMP will thus follow here.
OpenMP relies on the forking on joining of multiple threads, as shown in Fig. 3.2.

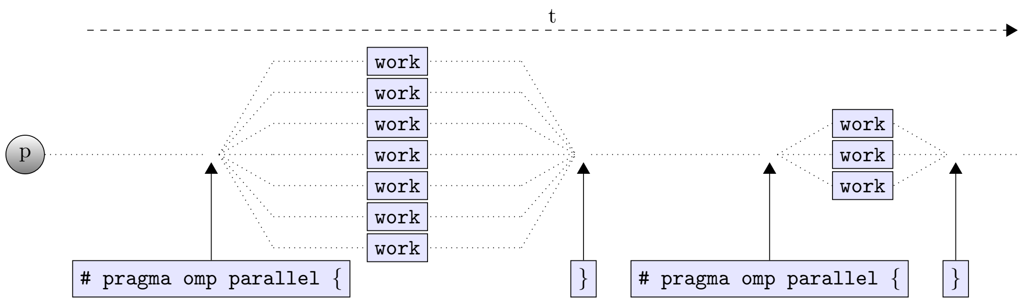


Figure 3.2.: A program with OpenMP pragmas spawning threads at several points in the program.

The very basic `#pragma omp parallel` introduces a parallel region in the program, forking as many threads as the user has requested. The number of threads to fork can be defined in several different ways, which are all described in the standard. The directive `#pragma omp parallel` introduces a parallel region applied to the next block, i.e., if no curly brackets are used to define a larger block, the parallel region will be limited to the next statement. Another directive initiating a parallel region is the `#pragma omp parallel for` directive. As the name

implies, it is used for parallelizing for loops and has according restrictions in comparison to the more general `parallel` directive.
The behavior of the parallel region induced through one of the directives can be further refined through clauses. For example, it is possible to make the execution of the parallel region dependent of some scalar expression or define the sharing status of different variables. While the end of a parallel region usually implies a barrier, even this behavior can be turned off through the `nowait` clause.
An important aspect of shared memory programming is the scope and sharing status of memory regions and variables. All accesses to shared variables will induce some management overhead, because the implementation needs to make sure that only one thread accesses the element at a time. Thus it is necessary to be able to distinguish shared and private variables, and to control the access to these variables. This can, for example, be done through critical regions, where OpenMP sequentializes the access to the shared variable. One of the main reasons for multiple threads to need access to a shared variable is the computation of a global sum. For this special case, OpenMP also offers a reduction clause, performing a collective reduce (p. 21).

3.2.4. PGAS Language Overview

The development of the PGAS has led to the emergence of several new communication languages and libraries, many either following a very distinct approach to the programming paradigm, being designed for a specific platform or being interoperable with an already existing high-level language. A small list of PGAS languages can be found on [86] and this section will give brief descriptions of several PGAS languages for an overview of state of the art PGAS communication possibilities, other than GASNet, GPI and GASPI, which are described in separate Secs. 3.2.2, 3.4 and 3.5.
OpenSHMEM is a PGAS API defined in [79]. Similar to GASPI, OpenSHMEM defines explicit putting and getting routines to move data to or obtain it from remote memory. One major difference between OpenSHMEM and GASPI is the memory management. While GASPI allows the user to define segments and manage memory however it is favorable for the application, OpenSHMEM requires a symmetric memory layout, i.e., the globally accessible memory partitions on each node must have the same layout.
UPC [10] is an extension to the C programming language, hiding the underlying communication from the application programmer. The programmer has the possibility to define a certain memory affinity to each variable or parts of an array. While in OpenSHMEM or GASPI the application programmer needs to explicitly access and communicate data with remote affinity, the programmers can access the data elements in UPC as they would in plain C. Thus the programmer does not have to take care of any data transferals or explicit communication, which makes the implementation of new programs more intuitive. But at the same time, this makes it very hard to optimize an UPC program in terms of communication, potentially leading

to unscalable programs. Other than the UC Berkeley, also the GNU Compiler Collection (GCC) is incorporating UPC in its compiler [64]. CAF [78] is, similar to UPC, an extension to Fortran, one of the most important programming languages in HPC. In CAF, the user has the possibility to distribute arrays across the memory of all nodes and makes them accessible for all nodes. Today, CAF is fully supported by the GCC [13].

Chapel [19] is a language designed explicitly for Cray architectures in the scope of the HPCS program. By now, it has been developed to run on a wider variety of systems, but it is still optimized for Cray architectures [11]. Titanium [45] is also developed by the UC Berkeley as a "parallel dialect of Java" [1]. Similar to UPC and CAF, also Titanium does not have explicit communication routines. Nonetheless, also here the user is able to control the affinity of the data.

All of these languages are either developed for certain architecture or base languages and have different approaches to handling the PGAS. Accordingly, some of these languages are only niche APIs and others are determined for a wider range of users. As in pure distributed and shared memory paradigms, there will be some more development in the future years, growing some of these languages and bringing forth more new languages and APIs. The next sections will introduce the PGAS languages dealt with in this thesis, as well as the de-facto standard for distributed memory architectures: MPI.

3.3. MPI - Message Passing Interface

MPI [75] is probably the most widely spread communication library used in scientific applications running on distributed systems [47]. MPI was initiated by the MPIF [77], which published the first standard in 1994. Ever since, the MPIF has continued to refine the standard and adapt it to the needs of the community and it has reached a huge scope of functionalities: It includes point-to-point message-passing, collective communications, group and communicator concepts, process creation and management functionalities, one-sided communication routines, a profiling interface and many more. Also, it defines language bindings for C, C++ and Fortran. Due to this range of functionalities that are covered and the time of its existence, it is not surprising that it has so many users.

Since MPI is the de-facto standard in distributed memory communication at the moment, GASPI implementations must be evaluated against it. Thus, this chapter will provide a short introduction of MPI. Since the MPI standard excesses the scope of the GASPI standard, the emphasis lies on operations that are relevant to this thesis, namely blocking and non-blocking collective communication, one-sided communication, groups and communicators. For further reference and more detail, please refer to the standard [75].

The following section introduces basic concepts that are inherent to every MPI application and necessary for all described MPI communication. Section 3.3.2 will then elaborate on one-sided communication routines in MPI, before collective communication will be described in Sec. 3.3.3.

3.3.1. Basic Concepts

The general concepts described in this section are necessary for all programs using MPI communication routines. Before being able to use any MPI communication routines, MPI has to be initialized through the `MPI_Init` routine. Only after this call returns successfully, other library routines may be called. At the end of an MPI program, the routine `MPI_Finalize` has to be called, to clean up and free allocated resources. After this routine has returned successfully, no further call to MPI library routines may be made.

Groups and Communicators

MPI communication relies on the concept of communicators, which are responsible for the actual communication, the distinction between different kinds of messages and the discrimination of communication universes. In order to do so, the communicator provides the scope for communication routines. This scope consists of contexts, groups, virtual topologies and attribute caching. The contexts partition the communication space, such that collective communication does not interfere with point-to-point communication and that different communicators do not interfere. For a detailed description of the elements of communicators, please refer to the MPI standard [75], Chap. 6, but to emphasize the difference between groups in MPI and groups in GASPI, MPI groups are described here.
A group defines the ranks of a set of processes, thus two groups consisting of the same set of processes but ranked in a different order are considered as two different groups. There are several different routines, making a group's ranks and properties accessible and groups themselves creatable and comparable. Based on these groups and ranks, the communicator enables communication between the processes in the group. There is an important distinction to be made between two different types of existing communicators. On one hand there are intracommunicators, enabling the communication within a single group of processes. Then, there are also intercommunicators which enable communication between two non-overlapping groups of processes.
While communicators are necessary for all communication routines, the completion calls described in the following subsection are only needed for non-blocking communication calls, i.e., calls that will return immediately after they have initiated the communication, without waiting for the communication to actually finish.

Completion Calls

Asynchronous or non-blocking MPI communication routines need completion calls to be able to complete the communication. In blocking communication routines, all local buffers can be reused after the successful return of the communication routine. This is delayed to the completion calls in non-blocking communication, i.e., the buffers can only be reused safely after the successful return of the according completion call. Which communication call is to be completed through a completion call is defined through request handles, which are an argument of non-blocking communication routines and completion routines. The different completion calls can roughly be split into waiting routines and testing routines.

The waiting routines block until the desired non-blocking communications, identified through request handles, are complete. No matter how the waiting function returns (successful or not), the routine will update the status (an argument of the routines) of the communication. One can either wait for one certain communication, for any one out of a given array of handles, for some communications associated with a given array of handles or for all communications, whose handles are in the request handle array to be completed.

For the testing routines, the situation is quite similar. They are available in the same versions, but they return immediately with one or more flags stating whether the communication identified through the request handle(s) is complete or not. If it is (or they are) complete, the testing routines will act as if they were waiting routines, namely stating that local buffers may be used again.

Even though many different completion calls are defined through the MPI standard, only the `MPI_Wait(req, status)` and `MPI_Test(request, flag, status)` routines are supported. Each of these will check on exactly one collective communication routine.

Having described groups and communicators, which are necessary for all communication routines, and completion calls, which are necessary for all non-blocking routines, the following concepts of windows, epochs and synchronization calls are distinct to one-sided communication routines.

Windows

Windows are used to make a processor's memory region visible and accessible to other processes participating in one-sided communication with this process, a concept similar to the memory regions in IB Verbs. The creation of a window is possible in several different ways, but all need a communicator. First, a window may be created via `MPI_Win_create`, a routine through which already allocated memory is exposed for RMA. Then one can allocate new memory for the created window by using either the call `MPI_Win_allocate`, directly exposing it to RMA, or `MPI_Win_allocate_shared`, allocating the memory as shared memory. This will allow the remote processes to directly store and load data into or from the window. A last possibility is to create a window to which memory will be dynamically attached later in the program. This is especially useful if it is not clear from the beginning on, how much RMA exposed memory is needed on a given process.

All processes keep a public and a private copy of their window such that these have to be kept synchronized. This is done in different ways depending on the chosen memory model, target communication and others. One of the most important tools for window synchronization is `MPI_Win_sync` which enables the application to synchronize at any necessary point. Further synchronization calls will be described in the following subsection together with epochs, another necessary concept of one-sided communication in MPI.

Epochs and Synchronization Calls

A further structure needed by all one-sided communication calls are the epochs, because RMA routines may only be called within these. Epochs are delimited through different synchronization calls and the user must distinguish between active target communication and passive target communication. In active target communication, the target process is engaged in the synchronization, while in the passive target communication, the target process has absolutely nothing to do with the data transfer. For all RMA communication, an access epoch has to be created on the origin process. In addition to that, an exposure epoch has to be induced on the target process in active target communication.

The origin process can only start passive target communication within a pair of locking calls. The passive target epoch is started on one process, where a lock type describes whether the calling process will have exclusive access to the window, if the access is shared, or if the epoch shall be started on all processes of the window group. The epoch is then ended by unlocking the window. After the return of an unlocking call, the communication is complete on both sides. Since the target process is not involved in synchronization and one might need some of the transferred data before the end of the epoch, it is possible to flush a window. Calling one of the flush functions lets the calling process wait for one or all RMA operations on a given window.

For active target communication there are several different possibilities of delimiting the access epoch and the exposure epoch. The most general approach is using `MPI_Win_fence`, which is a collective synchronization call and starts and ends access epochs as well as exposure epochs in all processes in the group of the window. A more resource saving possibility is to pair only those processes, that need to communicate. This is done on the origin process through `MPI_Win_start` and `MPI_Win_complete` and on the target process by calling `MPI_Win_post` and `MPI_Win_wait`. Alternatively one may also call `MPI_Win_test` to check if all communication in this epoch has completed. If so it will behave as if a call to `MPI_Win_wait` had been made. The ending of an epoch always implies completion of the communication on the origin process as well as on the target process.

Within these epochs, the one-sided communication calls described in the next section can be executed.

3.3.2. One-sided Communication

Apart from the traditional message-passing, MPI also offers one-sided communication. The one-sided communication calls come in two flavors: either as a RMA calls or as calls with request handles. The RMA calls are `MPI_Put` and `MPI_Get` for simple data placement or retrieval, being synchronized only at the end of the epoch. Concurrent RMA calls to the same location are undefined, i.e., generate race conditions. `MPI_RPut` and `MPI_RGet` return a request handle, such that the completion calls (p. 43) may be used and may only be initiated during passive target epochs.
Additionally, the standard defines several atomic RMA routines, which can somewhat be seen as extensions to the put and get routines. First there is `MPI_Accumulate`, which can roughly be described as a two-party remote reduce. `MPI_Accumulate` takes an operations or `MPI_REPLACE` and performs this operation on the remote data together with some local data. The result is stored remotely. Thus `MPI_Put` can be seen as `MPI_Accumulate` with `MPI_REPLACE` as operation argument. The important difference is the differing handling of concurrent calls on the same remote data. While this yields undefined behavior with `MPI_Put`, the accumulating calls will be handled as if they were initiated sequentially. A variation of `MPI_Accumulate` is given through `MPI_Get_accumulate`, where the remote data is fetched, before the operation is performed. This routine is very similar to `MPI_Fetch_and_op`, which also retrieves the remotely stored data and then performs an operation on the remote side. Another atomic one-sided routine is `MPI_Compare_and_Swap`, where a local value is compared to a remote value and if they are identical, the remote value is exchanged by a third value.
Another important communication possibility that MPI offers, besides peer-to-peer communication and one-sided communication, is collective communication. Similar to the one-sided communication routines described in this subsection, collective communication routines come in different variations and are described in the subsection below.

3.3.3. Collective Communication

The MPI standard provides a flood of collective communication routines. Starting with essential barriers, the standard also guarantees different flavors of broadcast, scatter and reduce operations. The set of processes involved in the collective communication is defined through the communicator and all processes in the group must call the collective communication operation in order for the routine to succeed. If there is more than one collective operation to be performed in the group, these have to be called in the same order on all processes. This is especially important when working in a threaded environment. Great care also has to be taken considering the differences of collective communication in intercommunicators or intracommunicators, because the behavior of the collective routines may be somewhat surprising. This is explained in more detail in the subsection dealing with the reduce routines.
The user must also distinguish between blocking and non-blocking operations. While the block-

ing collective communication routines do not return before the local part of the communication is completed, meaning the send and receive buffers may be used again, the non-blocking routines return immediately. Only after the return of the matching completion call with its flag set to true (see p. 43), the user may be sure, that the local part of the operation is completed. Either version - blocking or non-blocking - is not required to synchronize the processes. This means that the successful return of the blocking communication call respectively of the completion call does not allow any assumptions about the status of the other processes involved in the communication.

The next subsections will describe the different MPI collective communication calls in more detail.

Barrier

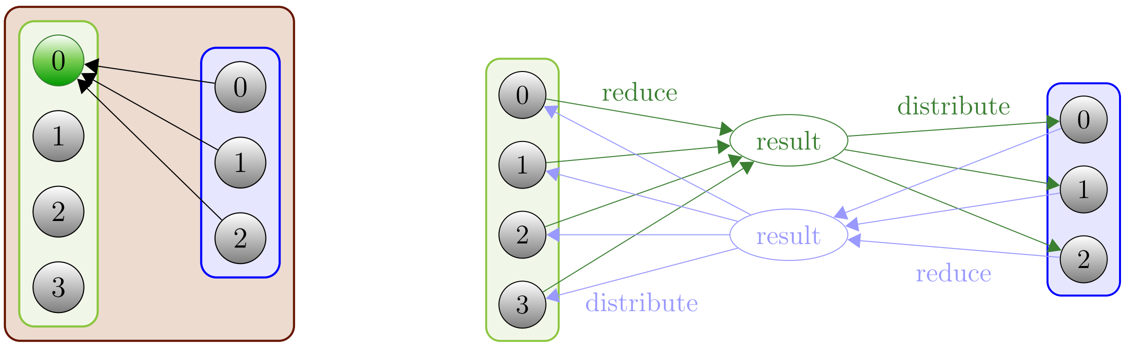
The barrier comes in two different versions: blocking and non-blocking. The blocking version `MPI_Barrier(communicator)` only returns, when all other processes in the communicator have also called the barrier. Concurrent calls to a blocking barrier will lead to an error.

The non-blocking version `MPI_IBarrier(communicator, request)` takes a request handle as a second argument, which is needed by the completion calls. The barrier call itself returns immediately and the above described barrier semantics, i.e., successful return only when all processes in the communicator have called the barrier, are transferred to the completion calls (see p. 43).

Reduce Routines

The reduce routine is defined in many different flavors in MPI. Yet, there are some prerequisites all must fulfill: All reduce routines take as arguments an input buffer `inbuf`, an output buffer `outbuf`, the number of elements to reduce `count`, the datatype of those elements `datatype` and the reduce operation `op` to perform. The MPI standard defines several reduce routines, like summation, minimum, maximum and product, to only name a few, but the user also has the possibility to pass user-defined operations to the reduce. All predefined operations are associative and commutative, whereas the user-defined reduce operations only need to be associative. If `inbuf` is an array of elements, the operation is performed element-wise.

Special emphasis has to be put on the different communicators available in MPI when talking about collective communication routines. If a reduce is to be done in an intercommunicator, only one of the groups has a root process. The processes of the rootless group provide the data to be reduced, as shown in Fig. 3.3a. The allreduce routine also behaves different from the description on p. 21 when used with an intercommunicator. While all processes participating in an intracommunicator allreduce will have the result of the reduction in their `outbuf` after successful completion, the reduction result of the reduce operation in one group is distributed



(a) Intercommunicator reduce **(b)** Intercommunicator allreduce

Figure 3.3.: (a) Intercommunicator reduce, where the green and the blue colored rectangles each represent a group and the brown rectangle represents the intercommunicator. (b) The results of the reduce operation are computed in both groups respectively and distributed to the processes of the other group. The implementation must not work the way depicted.

to all processes in the other group and vice versa in the intercommunicator case. Figure 3.3b schematically shows the data distribution in an intercommunicator allreduce.

Further reduce-related functions are the reduce-scatter functions. These routines make it possible for the application to scatter the result in blocks, i.e., not every process gets the same result. In its simplest version, this would be semantically equivalent to calling a reduce and afterwards calling a scatter on the root process. See p. 22 for a description of the scatter routine.

All of these different reduce-related routines are available in a blocking and in a non-blocking manner. As for the barrier, also the non-blocking reduce routines take a request handle as an argument, with which a following completion call can complete the routine, or test whether the routine has completed in the mean-time. The different completion calls are described on p. 43.

With MPI, a very comprehensive, high-level message-passing interface has been introduced in this section. The next section will deal with a quite contrary interface: the minimalist, low-level interface GPI, which focuses on one-sided communication.

3.4. GPI - Global Programming Interface

GPI [38] is a commercial programming interface designed by the Fraunhofer Institute for Industrial Mathematics (ITWM) [67] providing an API for the PGAS. It is based on the IB architecture [61] and the verbs described in this specification (and on p. 35). GPI applications may make use of RDMA routines instead of relying on message-passing. It provides communication primitives, environment runtime checks and synchronization routines. All these routines are designed such that an asynchronous programming model for overlapping communication with computation is promoted.

Since the GPI is the basis for GASPI, it is described in this chapter. Special emphasis will be taken on the similarities and differences to the MPI standard described in chapter 3.3.

3.4.1. Basic Concepts

This section will describe basic concepts that are required to fully understand GPI and the design of according applications. Some of these concept differ greatly from concepts known from MPI, others are rather similar or at least have an equivalent. One of these is similarities is that also the GPI has to be initialized and shutdown through the routines `startGPI` and `shutdownGPI`. Very different from MPI are the following concepts of daemons.

Daemons and Ranks

GPI works with daemons on every compute node which manage GPI applications on the compute nodes. A given binary is started on one node which becomes the master node and the remote nodes involved are the worker nodes. The nodes are given ranks, with the master node having rank 0 and the worker nodes being assigned ranks from 1 to $P-1$, where P is the total number of nodes used in the application. Besides this hierarchy there is no further concept of groups or similar in GPI, a concept in heavy contrast to MPI or GASPI.

Memory Blocks

Every node hosts one partition of the PGAS, which may be accessed by all other nodes directly. Not all memory available to one node must be made globally accessible, a node may also have private memory, as previously shown in Fig. 2.4 on p. 9. The globally accessible memory is called the GPI memory block and is accessible after start up of the GPI through a designated pointer. All data transferred with one-sided routines needs to reside in these memory blocks. During communication all locations are given through byte-offsets, indicating the distance from the start of the memory block, instead of addresses.
This concept is similar to the concept of windows in MPI, but here, the memory blocks can not be created during runtime. Instead, there is one large block accessible until the shutdown of GPI. There is also no synchronization necessary between global and local memory.

Queues

The direct memory access (DMA) communication described below relies on queues to manage and monitor the different operations. Every node has a fixed number of available communication queues to which the communication calls are posted. The underlying IB network then executes the queued calls asynchronously. The user is responsible to keep track of the posted calls so the queue is not overfilled at some point. Should the queue be overfilled, it would break. To take care of this, there are several management calls to either check the number of outstanding DMA calls in a queue or to wait for all outstanding DMA calls in a queue to be finished. This queuing concept also appears in IB Verbs, on which this interface is built.

3.4.2. One-sided Communication

While all GPI communication routines directly access remote memory, there are communication routines with one-sided semantics and routines with two-sided semantics. Those with the two-sided semantics are the passive communication routines and the send and receive routines. For both the one-sided and the two-sided communication calls the memory locations are given through relative offsets from the start address of the involved GPI memory blocks. Thus only data and memory locations lying in the global memory of the nodes may be transferred with these routines.
The one-sided communication calls are `readDmaGPI` and `writeDmaGPI`, both only involving the process issuing the call and return immediately. In each case the communication call is posted to one of the queues available to the node and the actual communication is then handled by the underlying network layer. This especially implies that the uninvolved node does not know whether the information has been written or read and also that the issuing node needs to query the associated queue to get the status of the issued call.
Besides these one-sided communication routines, GPI also defines some collective communication routines.

3.4.3. Collective Communication

Very different from the above one-sided communication, the GPI collective communication takes actual addresses of data to be communicated instead of local offsets. This means that also data in the local private memory may be communicated and not only data in the GPI global memory block and it also implies, that the data will be copied to some internal buffer.
GPI does not offer a great variety of collective communication routines. Only the barrier, as a global synchronization routine and the allreduce operation are defined. Here the term global synchronization has to be taken literally, as GPI does not offer the concept of groups. This very slim specification is the basis for the development of GASPI, described in the next section.

3.5. GASPI - Global Address Space Programming Interface

The GASPI specification [33] has been developed within a BMBF funded project from 2011-2014, based on the GPI (section 3.4), which has before been developed by the Fraunhofer ITWM. It is a PGAS communication API, aiming at high scalability, flexibility and fault tolerance. At the same time, an important goal of the project was to keep the specification as slim as possible. After the end of the project, the GASPI Forum was founded to further develop the specification. Already the GPI had been based on ibverbs (see section 3.2.1) from

the OFED stack to exploit the possibilities offered through the IB network. Since requiring an IB network would be very limiting to the API, the reference implementation GPI-2 by the Fraunhofer ITWM has been extended to RoCE devices (p. 17) and TCP/IP [28]. In addition a closed source version operating on the Cray Aries network has been developed [50].
This section will describe those features of the specification relevant to this thesis. It does not, for example, include passive communication routines. Please refer to the specification for further information on these features. The `gaspi_read_notify` routine will be emphasized in Chap. 6, because this routine was included in the specification based on a proposal by C. Simmendinger and me in 2016 [95]. First, basic concepts like groups, segments, queues and the timeout mechanism will be described. Afterwards, one-sided communication routines together with weak synchronization primitives will be explained. The last subsection will then give an overview of collective communication routines in GASPI. In all sections the described features will be put into relation to according MPI concepts as described in Sec. 3.3.

3.5.1. Basic Concepts

Before going into a more exhaustive description of one-sided and collective communication routines, the basic concepts of GASPI will be described. These form the basis of every GASPI program and are an important part of the GASPI programming paradigm.

GASPI Life Cycle

Every GASPI program is divided into several execution phases, forming the GASPI life cycle (section 5 in [32]):

1. Setup,
2. Initialization,
3. Working, and
4. Shutdown.

All routines are defined for one of these execution phases and yield undefined behavior if called in a different execution phase. The only mandatory phases are the initialization phase and the shutdown phase. The initialization phase consists of a call to `gaspi_proc_init`, preparing the internal management of GASPI and allocating necessary resources. After successful return of `gaspi_proc_init` the user is in the working phase, into which most of the GASPI routines fall. After having completed the working phase, the user needs to explicitly end the phase by calling `gaspi_proc_term` to free the previously allocated, internal resources and clean up.

Groups

In order to be able to limit the collective communication routines to certain subsets of nodes, GASPI has the notion of groups. The group `GASPI_GROUP_ALL` is set up during the initialization of GASPI and assigns ranks to the different processes. These are the only rank numbers which the processes will have, i.e., the processes do not have different ranks in different groups. Additional groups can be created at any point of the working phase. The creation of GASPI groups is a multistage operation, consisting of local and collective operation steps.

`gaspi_group_create` creates an empty group. This group then needs to be filled up with the ranks, which shall be included in the new group with `gaspi_group_add`. These steps have to be locally invoked on every rank in the group. Finally the collective operation `gaspi_group_commit` must be called in order to set up the communication infrastructure and internal management within the group. Being a collective operation means that every rank in the group must call `gaspi_group_commit` with identical values and a successful return of the routine is only possible if all ranks in the group have called the routine.

While the concept of groups is also present in the MPI standard, the term group is used in different ways in these two APIs. While in MPI the group is only one element that defines a communicator, a GASPI group should rather be compared to an MPI communicator, because it manages everything necessary for communication and does not only give rank numbers to the included processes. One important difference between the communicators in MPI and the groups in GASPI is that there are no intra- and intercommunicators in GASPI. If a collective communication routine is supposed to run across two groups, a new group combining these groups has to be built. Alternatively there has to be some explicit, user implemented communication between the two groups at some point.

Timeout Mechanism

One key feature of writing scalable and failure tolerant programs is the usage of non-blocking communication routines. A process calling a communication routine in a blocking manner will stay in this routine until it has successfully completed. In case of an error, the process will infinitely stay in this routine and no further process can be achieved. One of the main issues with communication routines is the involvement of other ranks and network resources. This means a process within a communication routine may spend a significant amount of time idling and waiting for status updates from other resources - this amount of time possibly multiplying when ever larger systems need to handle messages growing in size and numbers. To prevent this idling time and the dead-lock in case of error, non-blocking routines enable the user to do some other work while no progress can be achieved within the communication call, or to check for an error, if the call does not return successfully.

While MPI offers several different routines for changing the execution mode, i.e., blocking or non-blocking communication routines, GASPI introduces a so called timeout mechanism for non-local operations, enabling the implementation of failure tolerant programs. It is triggered by the argument `gaspi_timeout_t timeout` in the function call. The GASPI specification defines three possible modes: a blocking mode triggered through the usage of the predefined

GASPI_BLOCK, a testing mode triggered by using GASPI_TEST and a user-defined timeout mode. In the two non-blocking cases, the routine will perform some progress while called and return with GASPI_TIMEOUT if it has not completed all necessary work within the given time frame. The user will then have to repeatedly call the same routine until it has completed successfully, unlike having to call MPI_Wait or MPI_Test in MPI. Listing 3.2 shows a possible usage of a non-blocking GASPI routine.

```
while((ret = gaspi_barrier(buffer_send,
                           buffer_receive,
                           num,
                           operation,
                           datatype,
                           group,
                           GASPI_TEST)) != GASPI_SUCCESS){
  if(ret != GASPI_TIMEOUT){
    handle_error(&ret);
  }
  else{
    do_unrelated_work();
  }
}
```

Listing 3.2: Possible usage of a non-blocking barrier in GASPI.

The user repeatedly returns into the barrier, which checks if all other processes have also reached the barrier. Within the loop, the return value of the barrier is checked for errors, which could then possibly be handled or at least a clean shutdown of the program is made possible. In MPI at least two different routines are needed to implement a similar workflow. The timeout mechanism is part of every non-local routine in GASPI. Another very important concept, necessary for almost all communication in GASPI, are the segments described below.

Segments

In order to use the one-sided communication routines of the GASPI API, the user needs to allocate and register segments. All data stored in these segments may then be accessed, also by remote ranks, via one-sided communication routines. The segments may be allocated and registered for single nodes or group-wise. The easiest way to create these segments is group-wise with the gaspi_segment_create routine. To do so, a group has to be created previously. The routine then allocates a local segment of the desired size and registers it with the other ranks in the group. Because the registration process is executed group-wise, this is a collective routine, i.e., all ranks in the group must invoke the routine. An exemplary setup of multiple segments is depicted in Fig. 3.4. All data transferred by or to remote ranks must lie within this registered segment. The local partition of the segment registered to the PGAS can be accessed through a pointer retrieved with gaspi_segment_ptr. Thus each rank can normally work on

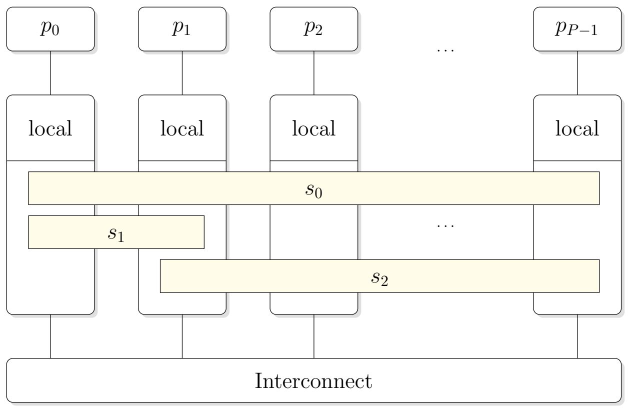


Figure 3.4.: Exemplary segmentation of the partitioned global address space with GASPI.

local data without having to invoke any communication routines.

The different segments also play an important role for the weak synchronization primitives described in Sec. 3.5.2. Internally, a segment of the desired size plus some additional space for notifications is allocated. The weak synchronization routines, described with one-sided communication below, will access this notification buffer. So, contrary to GPI, GASPI enables the application programmer to allocate several smaller segments. While this sounds similar to MPI's windows, some differences remain, e.g., there is no need of synchronizing the local and global memory partitions in GASPI. One more important concept in GASPI are the communication queues, necessary in one-sided communication and weak synchronization. Queues will be described in the next section, before Sec. 3.5.2 will bring the different concepts together.

Queues

Communication queue concepts have already been described in Secs. 3.2.1 and 3.4.1, and also GASPI takes on this principle for separation of concerns. Every rank has its own set of queues needed for communication. Each one-sided communication routine will generate a communication request which is posted in one of the available queues as requested by the user. The requests in the different queues will be worked on in a fair manner, i.e., no request will be delayed infinitely because another queue is being worked on.

The queues can only take on a limited number of communication requests, thus the user needs to handle the freeing of the queues and needs to take care that she does not post a request into a full queue. The specification offers several routines to do so, which can be found in sections 8.5 and 12.3 of [33]. Overall, the handling of queues in GASPI is very similar to that in GPI.

3.5.2. One-sided Communication

The GASPI specification has a variety of communication routines to offer, with a special focus on one-sided RDMA communication. The basic one-sided communication routines are `gaspi_write` and `gaspi_read` and can be issued at any time during the working phase of the GASPI life cycle. The first transfers data from the local partition of the PGAS to a remote partition of the PGAS, while the latter acts the other way around. In addition to these basic routines, GASPI also gives the possibility of listed writes and reads, where the user may give a list of local and remote offsets to transfer data. All of these communication routines are equipped with a timeout argument, because they are all non-local.

The one-sided communication routines offload the communication work to the system, ideally to an RDMA capable network, by posting the communication request to one of the queues. This introduces two important issues: 1. the successful return of one of these communication routines does not imply anything on the status of the data transferal and 2. since only one process is active in the communication, the other process does not know, whether it has received data. The first issue is addressed through the `gaspi_wait` routine. It takes a queue ID as input and waits for the communication requests in that queue to have been processed. As soon as the routine returns successfully, all data transfers connected to the communication requests in that queue have been processed and the read data will be available respectively the local buffer of a write request may be reused without jeopardizing the correct transferal of the respective data.

To address the second issue, the GASPI specification also offers so called weak synchronization routines. These routines enable the application programmer to notify the passive rank of written data. For this weak synchronization, each process has a notification array as described on p. 54. While there is no guarantee on the ordering of the data transferal of successive writes, notifications are guaranteed to not overtake previous writes from the initiating rank to the same destination rank and segment in the same queue. This means, if process p_0 issues multiple writes to rank p and segment s in queue q and afterwards issues a notification to the destination tuple $< p, s >$ in queue q, this notification will be written after all previous writes have been successfully completed. The receiving side, which has so far been totally passive, can issue a call to `gaspi_notify_waitsome` whenever it needs the data to check whether it is already available.

Figure 3.5 depicts this process, exemplarily showing, that the order of the issuing of the write requests does not necessarily influence the order of writing of the data. Processes 0 and 1 are both active in some application. The communication requests issued by the two processes are depicted in order on the two dashed lines. The writes and notifications of process 0 are issued to different queues. From there on, the network infrastructure handles the data transfers from the memory of process 0 directly into the memory of process 1 (blue arrow). As soon as process 1 needs the data of writes `w0` and `w3`, it checks in its local memory, whether notification `n0` has

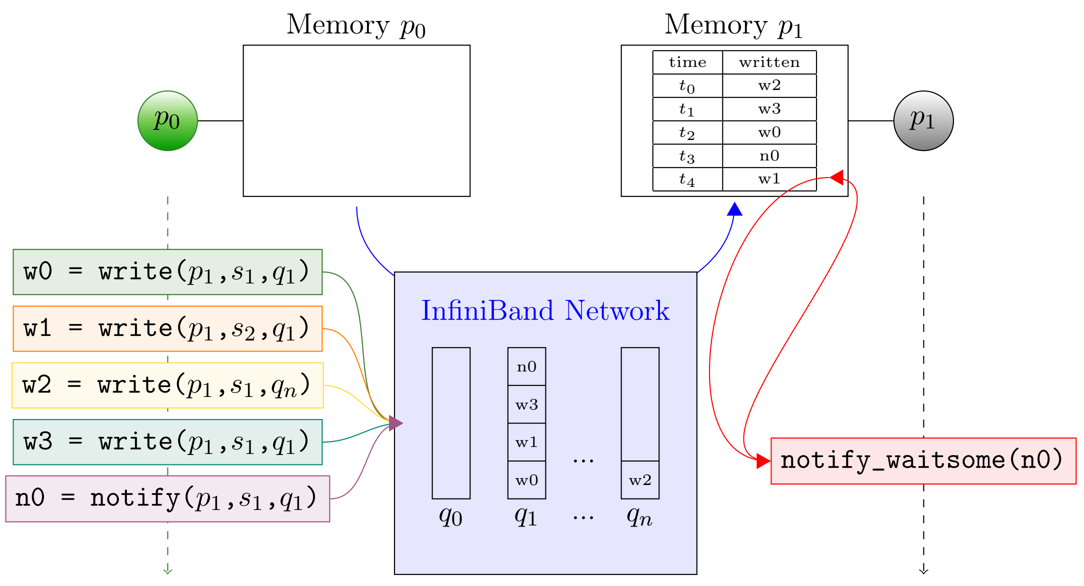


Figure 3.5.: Write order of different write requests from process 0 to process 1. In the table in the memory of process 1 the times $t_k, \forall i,j \in \mathbb{N} : i < j \Rightarrow t_i < t_j$ of the write completion of the different work requests are shown.

been set. Latter can either be done in a blocking or a non-blocking manner, depending on the timeout set by the user in `gaspi_notify_waitsome`.

While a call to `gaspi_notify` is guaranteed to not overtake any previously issued writes to the same queue, rank and segment, the `gaspi_write_notify` routine offers this concept for single messages. The notification set with this routine is guaranteed to be set only after the data of the coupled write has been written and has no implication on any other writes. This distinction has been included in the standard after the Forum's meeting in June 2016, based on a proposal by C. Simmendinger and me [94].

Besides these one-sided communication routines, the GASPI specification also defines collective communication routines. The proximity to the GPI specification becomes apparent, because also GASPI only few collective communication routines, described in the next section.

3.5.3. Collective Communication

Similar to GPI (see Sec. 3.4) and contrary to MPI (Chap. 3.3), GASPI includes only a barrier and an allreduce to keep the specification slim. The main functionalities of the two routines are described in Sec. 2.3.2. Both are issued group-wise and necessitate all ranks in the group to invoke the routine, which directly implies that both are non-local and thus equipped with a timeout. But there are some fundamental differences between the usage of collective communication routines and other communication routines in GASPI. First of all, the collective communication routines do not take a queue ID as an argument. The communication (and

possible computation) is completely handled by the GASPI implementation. The successful return of a collective communication routine means that all necessary data has not only been transferred but also processed, if applicable, and it is safe to reuse all associated buffers.
Another aspect of collective communication completely handled by the implementation is the memory management. While the one-sided communication routines of GASPI take source and destination memory segments and offsets as input, the allreduce takes pointers to source and destination buffers, i.e., the data for an allreduce may also reside in the local, unregistered memory segment. The source buffer needs to carry the data used for the allreduce operation and the destination buffer needs to be large enough to hold the result of the allreduce operation. Since there are no limitations posed to the location of these buffers, the implementation will need to make copies of the source data for the internal RDMA communication.
Especially in comparison to MPI, the collective communication routines defined by the GASPI specification present only a small fraction of possible collective routines. This decision was explicitly made to keep the specification slim while at the same time including the most important collectives for synchronization and global operations. All other collective routines need to be either implemented by every user herself or provided by some external library. One possible implementation of such a library is presented in Chap. 5.
With the description of collective communication routines in GASPI, the introduction of HPC communication APIs and languages relevant to this thesis is concluded. The next section will sum up the information of this chapter, before Chap. 4 will pass into my own research and contributions.

3.6. Summary

In this chapter, different HPC-relevant communication libraries and APIs have been introduced and described. The low-level communication libraries ibverbs and GASNet have been introduced, as they are both likely candidates to build GASPI on. GASNet offers some functionalities, which are also designated by the GASPI standard [33]. Implementing GASPI on top of GASNet would immediately offer the wanted portability through the given conduits and many of the asynchronous features through the Extended API. In addition, there is almost a one-to-one correspondence between several GASPI communication routines and GASNet communication routines. Despite the benefits GASNet offers for GASPI, there are also many challenges considering an implementation of GASPI over GASNet, including:

1. Dynamic Segments
   The GASPI standard demands the dynamic allocation of segments during runtime. GASNet can not offer this. The only possibility would be to first allocate a large global segment via GASNet and then manage the access of the different nodes on top of this segment. But this would be exactly the opposite, of what the GASPI standard meant to

achieve through the dynamic allocation of segments: resource savings.

2. Dynamic Infrastructure
   Another demand of the GASPI standard is the dynamic and possibly sparse communication infrastructure. As GASNet builds up the whole communication network at initialization, the sparse infrastructure, which is intended to save memory resources, can not be set up. All GASPI operations concerning the dynamic of the infrastructure, like `gaspi_connect` or `gaspi_disconnect` would thus turn no-ops.
3. Protocol Overhead
   Every wrapper around an existing API and every layer between hardware and user creates some amount of overhead. In the simplest case, a GASPI function calls a GASNet Core function, which might call a conduit specific hardware layer function. In more complex cases though, there will have to be several different `if`-clauses per GASPI call to choose the right GASNet function, due to the great variety of GASNet calls.
4. Failure Tolerance
   One of the main goals of the GASPI standard is to achieve failure tolerance, even if one of the compute nodes fail. But since every GASNet error kills the running job, failure tolerance is not achievable by implementing GASPI over GASNet.

Adding all these points up, an implementation of GASPI over GASNet did not make sense in the scope of the project. Instead, the reference implementation of GASPI is built directly on low-level communication APIs for IB, the Aries network or Ethernet.
MPI and OpenMP have been introduced as de-facto standards with which GASPI has to be interoperable. MPI is a distributed memory communication paradigm and hence a direct competitor of the GASPI specification. Since many HPC applications already use MPI communication, it is necessary for a GASPI implementation to be interoperable with MPI to ease the porting of the code to a new paradigm. Because GASPI is intended to be used in HPC environments with a hybrid memory architecture, it needs to be interoperable with threading libraries like OpenMP. The combination of GASPI with OpenMP or with MPI is used in benchmarks in Chap. 6.
GASPI is not the first communication library designed for the PGAS, nor will it be the last. In comparison to other presented PGAS APIs, GASPI does not oblige the application programmer to completely rewrite an existing application. Instead it is interoperable with the most commonly used communication libraries and the porting can be done step by step. Additionally, GASPI does not hide the communication steps from the user. While it might be somewhat more comfortable to access any remote or local data in the same manner, e.g., by accessing an element of an array, the user might not be aware of the communication necessary in the background. This may lead to a serious decrease of efficiency and performance and the user has little chance to optimize this. So even though it might be counter-intuitive to expose communication in a PGAS interface, it is necessary for maximizing the communication

optimization.
With the end of this chapter, the context of this thesis has been thoroughly established. The following chapters will now deal with my own contributions to the GASPI communication universe. The next chapter will deal with an adaption of the previously presented n-way dissemination algorithm for usage in a GASPI split-phase allreduce. Chapter 5 will then present a possible implementation of several collective communication routines with GASPI routines in the form of a collective library. Chapter 6 describes the new GASPI functionality `gaspi_read_notify` and two use-cases for this routine, before Chap. 7 will give an overall summary of this thesis together with an outlook on future research.

4. Adaption of the n-way Dissemination Algorithm

After having presented several different algorithms for collective communication routines in Sec. 2.3.3, this chapter will focus on the n-way dissemination algorithm (p. 28). The algorithm is very useful for barrier operations shown in the broad usage of the original algorithm in different APIs [32, 6]. But, in its original version, it is not usable for allreduce operations. The following sections will elaborate on the reasons, present an adaption of the algorithm to resolve these issues and also present experimental results. The chapter also gives a comparison to Bruck's algorithm (p. 29), as these algorithms are very similar. The work presented in this chapter has been published in a conference article [27].

4.1. Problem Statement

When using the n-way dissemination algorithm for an allreduce, the information received in every round is the partial result the source rank has computed in the round before. The receiving rank then computes a new local partial result from the received data and the local partial results already at hand.

Again, P is the number of involved ranks. Let S_l^p be the partial result rank p has computed in round l, $\circ$ be the reduction operation used and x_p be the rank's initial data. Then rank p receives n partial results $S_{l-1}^{r_{l,i}}$ in round l and computes

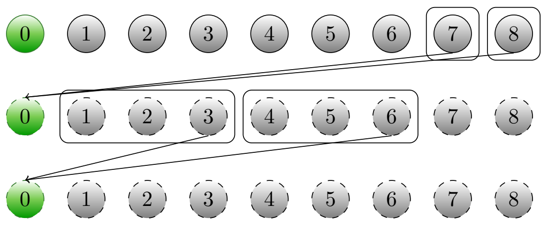
$$S_l^p = S_{l-1}^p \circ S_{l-1}^{r_{l,1}} \circ S_{l-1}^{r_{l,2}} \circ \cdots \circ S_{l-1}^{r_{l,n}}, \tag{4.1}$$

which it transfers to its peers $s_{l+1,i}$ in the next round. Here, $r_{l,i}$ are the i source ranks of round l and $s_{l+1,i}$ the i destination ranks of round $l+1$, as described on p. 28.

In the following example, this transfer of the partial results is explained in more detail for a 2-way dissemination algorithm with $P = 9 = (2+1)^2$ and $P = 8 = (2+1)^1 + 4 \neq 3^k$. In the example, it will also become clear, why the n-way dissemination algorithm needs some adaption for usage in an allreduce.

Example 4.1

The first example will deal with $P = 9$. In Fig. 4.1a, the communication scheme is depicted with a focus on rank 0 and incorporating the partial results transferred. The boxes surround



(a) Graphical representation of data movement from rank 0's view.

rank	round 0	round 1	round 2
0	x_0	$S_1^0 = x_0 \circ x_8 \circ x_7$	$S_2^0 = S_1^0 \circ S_1^6 \circ S_1^3$
1	x_1	$S_1^1 = x_1 \circ x_0 \circ x_8$	$S_2^1 = S_1^1 \circ S_1^7 \circ S_1^4$
2	x_2	$S_1^2 = x_2 \circ x_1 \circ x_0$	$S_2^2 = S_1^2 \circ S_1^8 \circ S_1^5$
3	x_3	$S_1^3 = x_3 \circ x_2 \circ x_1$	$S_2^3 = S_1^3 \circ S_1^0 \circ S_1^6$
4	x_4	$S_1^4 = x_4 \circ x_3 \circ x_2$	$S_2^4 = S_1^4 \circ S_1^1 \circ S_1^7$
5	x_5	$S_1^5 = x_5 \circ x_4 \circ x_3$	$S_2^5 = S_1^5 \circ S_1^2 \circ S_1^8$
6	x_6	$S_1^6 = x_6 \circ x_5 \circ x_4$	$S_2^6 = S_1^6 \circ S_1^3 \circ S_1^0$
7	x_7	$S_1^7 = x_7 \circ x_6 \circ x_5$	$S_2^7 = S_1^7 \circ S_1^4 \circ S_1^1$
8	x_8	$S_1^8 = x_8 \circ x_7 \circ x_6$	$S_2^8 = S_1^8 \circ S_1^5 \circ S_1^2$

(b) Round-wise computation of partial results and data movement for all ranks.

Figure 4.1.: The 2-way dissemination algorithm communication and computation scheme for $P = 9$.

those ranks, whose initial data is included in the partial result transferred: rank 0 receives x_7 and x_8 in the first round and can thus compute $S_1^0 = x_7 \circ x_8 \circ x_0$. In the second communication round, rank 0 receives $S_1^6 = x_4 \circ x_5 \circ x_6$ and $S_1^3 = x_1 \circ x_2 \circ x_3$, then being able to compute

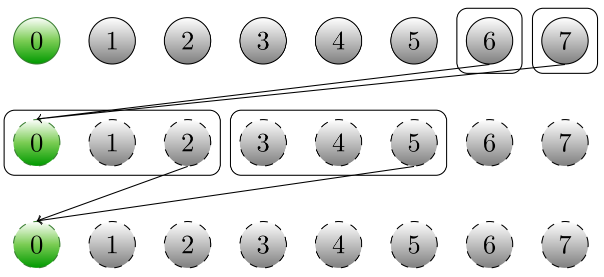
$$\begin{aligned} S_2^0 &= S_1^0 \circ S_1^6 \circ S_1^3 \\ &= x_7 \circ x_8 \circ x_0 \circ (x_4 \circ x_5 \circ x_6) \circ (x_1 \circ x_2 \circ x_3). \end{aligned} \tag{4.2}$$

This final reduction result includes every data item exactly once. The partial results computed by the different ranks and their final results are listed in Fig. 4.1b. One main issue all butterfly-like algorithms suffer from, is their applicability only to associative reduction operations. A comparison of the partial results in Fig. 4.1b shows, that even though all ranks will have a final result including all initial data items exactly once, the reduction operation has been applied to the initial data in different orders.

The second example shows the same steps for a 2-way dissemination algorithm and $P = 8 = (2+1)^1 + 5 \neq (2+1)^k$ in Fig. 4.2. Rank 0 receives x_6 and x_7 in the first round and can thus compute $S_1^0 = x_6 \circ x_7 \circ x_0$. In the second communication round, rank 0 receives $S_1^5 = x_3 \circ x_4 \circ x_5$ and $S_1^2 = x_0 \circ x_1 \circ x_2$, then being able to compute

$$\begin{aligned} S_2^0 &= S_1^0 \circ S_1^5 \circ S_1^2 \\ &= x_6 \circ x_7 \circ \boxed{x_0} \circ (x_3 \circ x_4 \circ x_5) \circ (\boxed{x_0} \circ x_1 \circ x_2). \end{aligned} \tag{4.3}$$

Differing from the first case with $P = 9$, where every initial data element was included once in the final result, here, the initial data from rank 0 is included twice in the final result of rank 0. Expansion of the final results in Fig. 4.2b shows, that each rank will have included its own initial data twice in its final result.



(a) Graphical representation of data movement from rank 0's view.

rank	round 0	round 1	round 2
0	x_0	$S_1^0 = x_0 \circ x_7 \circ x_6$	$S_2^0 = S_1^0 \circ S_1^5 \circ S_1^2$
1	x_1	$S_1^1 = x_1 \circ x_0 \circ x_7$	$S_2^1 = S_1^1 \circ S_1^6 \circ S_1^3$
2	x_2	$S_1^2 = x_2 \circ x_1 \circ x_0$	$S_2^2 = S_1^2 \circ S_1^7 \circ S_1^4$
3	x_3	$S_1^3 = x_3 \circ x_2 \circ x_1$	$S_2^3 = S_1^3 \circ S_1^0 \circ S_1^5$
4	x_4	$S_1^4 = x_4 \circ x_3 \circ x_2$	$S_2^4 = S_1^4 \circ S_1^1 \circ S_1^6$
5	x_5	$S_1^5 = x_5 \circ x_4 \circ x_3$	$S_2^5 = S_1^5 \circ S_1^2 \circ S_1^7$
6	x_6	$S_1^6 = x_6 \circ x_5 \circ x_4$	$S_2^6 = S_1^6 \circ S_1^3 \circ S_1^0$
7	x_7	$S_1^7 = x_7 \circ x_6 \circ x_5$	$S_2^7 = S_1^7 \circ S_1^4 \circ S_1^1$

(b) Round-wise computation of partial results and data movement for all ranks.

Figure 4.2.: The 2-way dissemination algorithm communication and computation scheme for $P = 8$ in the first two out of three rounds.

●

Example 4.1 shows, that the 2-way dissemination algorithm will lead to a wrong computation of an allreduce result, if the number of participating ranks is $P = 10 \neq (2+1)^k$. The following lemma states this in a more general fashion for n-way dissemination algorithms and $P = (n+1)^k$ or $P \neq (n+1)^k$.

Lemma 4.2
If $P \neq (n+1)^k$, the final result of an allreduce implemented with an n-way dissemination algorithm will include data of at least one rank twice. If $P = (n+1)^k$, the n-way dissemination algorithm can be used for the allreduce operation without adaption.

Proof. In every communication round l, each rank receives n partial results each of which is the composition of the initial data of its $(n+1)^{l-1}$ left-hand neighbors. Thus the number of included initial data elements is described through

$$\sum_{i=1}^{l} n(n+1)^{i-1} + 1 \tag{4.4}$$

for every round l. As long as $\sum_{i=1}^{l} n(n+1)^{i-1} + 1 \leq P$, no data item is included twice in the partial result. After the last communication round k, the receiving rank p has information from a total of

$$\sum_{i=1}^{k} n(n+1)^{i-1} + 1 = (n+1)^k \tag{4.5}$$

ranks. Thus if $P = \sum_{i=1}^{k} n(n+1)^{i-1} + 1 = (n+1)^k$, with $k = \lceil \log_{n+1}(P) \rceil$, no initial data item x_i is included more than once in the final result.

Now let $P = (n+1)^{k_0} + q$, $q < n(n+1)^{k_0} \in \mathbb{N}\backslash\{0\}$. Then the number of communication rounds to be absolved is $k = k_0 + 1$ and in the end each rank will have information from a total of

$$\sum_{i=1}^{k_0+1} n(n+1)^{i-1} + 1 = (n+1)^{k_0+1} \tag{4.6}$$

ranks. Thus, the final result will contain

$$\begin{aligned} & (n+1)^{k_0+1} - P \\ = \quad & (n+1)^{k_0+1} - ((n+1)^{k_0} + q) \\ = \quad & (n+1)^{k_0+1} - (n+1)^{k_0} - q \\ = \quad & n(n+1)^{k_0} - q \end{aligned} \tag{4.7}$$

data items more than once. Because we chose $0 < q < n(n+1)^{k_0}$, this number will always be larger than zero.

$\square$

The multiple inclusion of the same initial data elements will not change the result for so called *idempotent* operations, like the maximum or minimum operation. In the case of a summation, the final result will be erroneous.

Having located the problems that arise when using the n-way dissemination algorithm with a number of participants $P \neq (n+1)^k$ and non-idempotent functions, the next section will describe the adaptions that can be made to the n-way dissemination algorithm to resolve these issues.

4.2. Adaption

The adaption of the n-way dissemination algorithm is mainly based on these two properties:

1. in every round l, p receives n new partial results, and
2. these partial results are the result of the combination of the data of the next $\sum_{i=0}^{l-1} n(n+1)^{i-1} + 1$ left-hand neighbors of the sender.

This is depicted in Fig. 4.3 through boxes. Highlighted in green are those ranks, whose data view is represented, that is rank 0's in the first row and rank 2's in the second row. Each box encloses those ranks, whose initial data is included in the partial result the right most rank in the box has transferred in a given round. This means for rank 0, it has its own data, received S_0^6 and S_0^7 in the first round (gray boxes) and will receive S_1^5 and S_1^2 from ranks 2 and 5 in round 2 (white boxes).

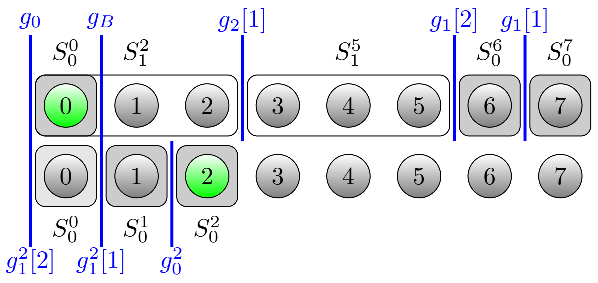


Figure 4.3.: The data boundaries g and received partial results $S_i^{r_{l,j}}$ of ranks 0 and 2.

As each of the boxes describes one of the partial results received, the included initial data items can not be retrieved by the destination rank. The change from one box to the next is thus defined as a *data boundary*. The main idea of the adaption is to find data boundaries in the data of the last round's source ranks, which coincide with data boundaries in the destination rank's data. If such a correspondence is found, the data sent in the last round is reduced accordingly. To be able to do so, it is necessary to describe these boundaries in a mathematical manner. Considering the data elements included in each partial result received, the data boundaries of the receiver p can be described as:

$$g_{l_{\text{rcv}}}[j_{\text{rcv}}] = p - n \sum_{i=0}^{l_{\text{rcv}}-2} (n+1)^i - j_{\text{rcv}}(n+1)^{l_{\text{rcv}}-1} \bmod P, \tag{4.8}$$

where $j_{\text{rcv}}(n+1)^{l_{\text{rcv}}-1}$ describes the boundary created through the data transferred by rank $r_{l_{\text{rcv}},j_{\text{rcv}}}$ in round l_{rcv}.

Also, the sending ranks have received partial results in the preceding rounds, which are marked through corresponding boundaries. From the view of rank p in the last round k, these boundaries are then described through

$$g^s_{l_{\text{snd}}}[j_{\text{snd}}] = p - s(n+1)^{k-1} - n \sum_{i=0}^{l_{\text{snd}}-2} (n+1)^i - j_{\text{snd}}(n+1)^{l_{\text{snd}}-1} \bmod P, \tag{4.9}$$

with $s \in \{1, \dots, n\}$ distinguishing the n senders and j_{snd}, l_{snd} corresponding to the above j_{rcv}, l_{rcv} for the sending rank. To also consider those cases, where only the initial data of the sending or the receiving rank is included more than once in the final result, we let $l_{\text{snd}}, l_{\text{rcv}} \in \{0, \dots, k-1\}$ and introduce an additional *base border* g_B in the destination rank's data.

These boundaries are also depicted in Fig. 4.3 for the given example of a 2-way dissemination algorithm with 8 ranks. The boundaries g_B, g_0, $g_1[1]$ and $g_1[2]$ on rank 0 and g_0^2, $g_1^2[1]$ and $g_1^2[2]$ on rank 2 always reside between two boxes representing the partial results sent in each communication rank. Since the boundaries g_B and $g_1^2[1]$ coincide, the first sender in the last round, that is rank 5, transfers its partial result but rank 2 only transfers a reduction $S' = x_2 \circ x_1$

instead of $x_2 \circ x_1 \circ x_0$.
More generally speaking, the algorithm is adaptable, if there are boundaries on the source rank that coincide with boundaries on the destination rank, i.e.,

$$g^s_{l_{\text{snd}}}[j_{\text{snd}}] = g_{l_{\text{rcv}}}[j_{\text{rcv}}] \tag{4.10}$$

or $g^s_{l_{\text{snd}}}[j_{\text{snd}}] = g_B$. To be able to precalculate these boundaries, Eq. 4.10 needs to be changed:

$$g_{l_{\text{rcv}}}[j_{\text{rcv}}] - g^s_{l_{\text{snd}}}[j_{\text{snd}}] \equiv 0 \bmod P. \tag{4.11}$$

From here, the numbers P, for which the algorithm is adaptable, can be determined:

$$P = s(n+1)^{k-1} + n \sum_{i=0}^{l_{\text{snd}}-2} (n+1)^i + j_{\text{snd}}(n+1)^{l_{\text{snd}}-1} - n \sum_{i=0}^{l_{\text{rcv}}-2} (n+1)^i - j_{\text{rcv}}(n+1)^{l_{\text{rcv}}-1}, \tag{4.12}$$

where

$$\begin{aligned} k &= \lceil \log_{n+1}(P) \rceil \\ s, j_{\text{snd}}, j_{\text{rcv}} &\in \{1, \ldots, n\} \\ l_{\text{rcv}}, l_{\text{snd}} &\in \{0, \ldots, k-1\}. \end{aligned}$$

Then the last source rank, defined through s, transfers only the data up to the given boundary and the receiving rank takes the partial result up to its given boundary out of the final result. Taking out the partial result in this context means: if the given operation has an inverse $\circ^{-1}$, apply this to the final result and the partial result defined through $g_{l_{\text{rcv}}}[j_{\text{rcv}}]$. If the operation does not have an inverse, recalculate the final result, hereby omitting the partial result defined through $g_{l_{\text{rcv}}}[j_{\text{rcv}}]$. Since this boundary is known from the very beginning, it is possible to store this partial result in the round it is created, thus saving additional computation time at the end.
For given P, a 5-tuple $(s, l_{\text{snd}}, l_{\text{rcv}}, j_{\text{snd}}, j_{\text{rcv}})$ can be precalculated for different n. Then this 5-tuple also describes the adaption of the algorithm:

Theorem 4.3
Given the 5-tuple $(s, l_{\text{snd}}, l_{\text{rcv}}, j_{\text{snd}}, j_{\text{rcv}})$, the last round of the n-way dissemination algorithm is adapted through one of the following cases:

1. $l_{\text{rcv}}, l_{\text{snd}} > 0$
   The sender $p - s(n+1)^{k-1}$ sends its partial result up to $g^s_{l_{\text{snd}}}[j_{\text{snd}}]$ and the receiver takes out its partial result up to the boundary $g_{l_{\text{rcv}}}[j_{\text{rcv}}]$.

2. $l_{\mathrm{rcv}} > 0, l_{\mathrm{snd}} = 0$
   The sender $p - s(n+1)^{k-1}$ sends its own data and the receiver takes out its partial result up to the boundary $g_{l_{\mathrm{rcv}}}[j_{\mathrm{rcv}}]$.

3. $l_{\mathrm{rcv}} = 0, l_{\mathrm{snd}} = 0$
   The sender $p - (s-1)(n+1)^{k-1}$ sends its last calculated partial result. If $s = 1$ the algorithm ends after $k-1$ rounds.

4. $l_{\mathrm{rcv}} = 0, l_{\mathrm{snd}} = 1$
   The sender $p - s(n+1)^{k-1}$ sends its partial result up to $g^s_{l_{\mathrm{snd}}}[j_{\mathrm{snd}} - 1]$. If $j_{\mathrm{snd}} = 1$, the sender only sends its initial data.

5. $l_{\mathrm{rcv}} = 0, l_{\mathrm{snd}} > 1$
   The sender $p - s(n+1)^{k-1}$ sends its partial result up to $g^s_{l_{\mathrm{snd}}}[j_{\mathrm{snd}}]$ and the receiver takes out its initial data from the final result.

Proof. We show the correctness of the above theorem by using that at the end each process will have to calculate the final result from P different data elements. We therefore look at (4.12) and how the given 5-tuple changes the terms of relevance. We will again need the fact, that the received partial results are always a composition of the initial data of neighboring elements.

1. $l_{\mathrm{rcv}}, l_{\mathrm{snd}} > 0$

$$\begin{aligned} P &= s\,(n+1)^{k-1} + n \sum_{i=0}^{l_{\mathrm{snd}}-2} (n+1)^i + j_{\mathrm{snd}}\,(n+1)^{l_{\mathrm{snd}}-1} \\ &\quad -n \sum_{i=0}^{l_{\mathrm{rcv}}-2} (n+1)^i - j_{\mathrm{rcv}}\,(n+1)^{l_{\mathrm{rcv}}-1} \\ &= g^s_{l_{\mathrm{snd}}}[j_{\mathrm{snd}}] - g_{l_{\mathrm{rcv}}}[j_{\mathrm{rcv}}] \ . \end{aligned} \tag{4.13}$$

   In order to have the result of P elements the sender must thus transfer the partial result including the data up to $g^s_{l_{\mathrm{snd}}}[j_{\mathrm{snd}}]$ and the receiver takes out the elements up to $g_{l_{\mathrm{rcv}}}[j_{\mathrm{rcv}}]$.

2. $l_{\mathrm{rcv}} > 0, l_{\mathrm{snd}} = 0$

$$\begin{aligned} P &= s\,(n+1)^{k-1} - n \sum_{i=0}^{l_{\mathrm{rcv}}-2} (n+1)^i - j_{\mathrm{rcv}}\,(n+1)^{l_{\mathrm{rcv}}-1} \\ &= s\,(n+1)^{k-1} - g_{l_{\mathrm{rcv}}}[j_{\mathrm{rcv}}] \end{aligned} \tag{4.14}$$

   and thus we see that the sender must send only its own data, while the receiver takes out data up to $g_{l_{\mathrm{rcv}}}[j_{\mathrm{rcv}}]$.

3. $l_{\text{rcv}} = 0, l_{\text{snd}} = 0$
$$P = s\left(n+1\right)^{k-1} \quad . \tag{4.15}$$
In the first $k-1$ rounds the receiving rank will already have the partial result of $n\sum_{i=1}^{k-1}\left(n+1\right)^{i} = \left(n+1\right)^{k-1} - 1$ elements. In the last round it then receives the partial sums of $\left(s-1\right)\left(n+1\right)^{k-1}$ further elements by the first $s-1$ senders and can thus compute the partial result from a total of $\left(s-1\right)\left(n+1\right)^{k-1} + \left(n+1\right)^{k-1} = s\left(n+1\right)^{k-1} - 1$ elements. Including its own data makes the final result of $s\left(n+1\right)^{k-1} = P$ elements.
If $s = 1$ the algorithm is done after $k-1$ rounds.

4. $l_{\text{rcv}} = 0, l_{\text{snd}} = 1$
$$P = s\left(n+1\right)^{k-1} + j_{\text{snd}} \tag{4.16}$$
Following the same argumentation as above, the receiving rank will have the partial result of $s\left(n+1\right)^{k-1} - 1$ elements. It thus still needs
$$\begin{aligned} & P - \left(s\left(n+1\right)^{k-1} - 1\right) \\ = \;& s\left(n+1\right)^{k-1} + j_{\text{snd}} - s\left(n+1\right)^{k-1} + 1 \\ = \;& j_{\text{snd}} + 1 \end{aligned} \tag{4.17}$$
elements. Now taking into account its own data it still needs j_{snd} data elements. The data boundary $g_1\left[j_{\text{snd}}\right]$ of the sender includes j_{snd} elements plus its own data, i.e. $j_{\text{snd}}+1$ elements. The $j_{\text{snd}}^{\text{th}}$ element will then be the receiving ranks data, thus it suffices to send up to $g_1\left[j_{\text{snd}} - 1\right]$.

5. $l_{\text{rcv}} = 0, l_{\text{snd}} > 1$
$$P = s\left(n+1\right)^{k-1} + n\sum_{i=0}^{l_{\text{snd}}-2}\left(n+1\right)^{i} + j_{\text{snd}}\left(n+1\right)^{l_{\text{snd}}-1} \tag{4.18}$$
In this case the sender sends a partial result which necessarily includes the initial data of the receiving rank. This means that the receiving rank has to take out its own initial data from the final result. Due to $l_{\text{snd}} > 1$ the sender will not be able to take a single initial data element out of the partial result to be transferred.

$\square$

Note that the case where a data boundary on the sending side corresponds to the base border on the receiving side, i.e., $g_{l_{\text{snd}}}^{s}\left[j_{\text{snd}}\right] = g_B$, has not been covered above. In this case, there is no 5-tuple like above, but rather $P - 1 = g_{l_{\text{snd}}}^{s}\left[j_{\text{snd}}\right]$ and the adaption and reasoning complies to case 4 in the above theorem. How such an adaption of the algorithm is done for the example of a 2-way dissemination algorithm and 8 ranks is shown in the next example.

Example 4.4
Again, 8 ranks are participating in the allreduce. As shown in Ex. 4.1 and Fig. 4.3, boundary $g_1^2[1]$ of the source rank 2 and g_B of the destination rank coincide. Thus, rank 2 will only transfer data up to this boundary, namely $S' = S_0^1 \circ S_0^2 = x_1 \circ x_2$. Rank 0 will then compute

$$\begin{aligned} S_2^0 &= S_1^0 \circ S_1^5 \circ S' \\ &= x_6 \circ x_7 \circ x_0 \circ (x_3 \circ x_4 \circ x_5) \circ (x_1 \circ x_2), \end{aligned} \tag{4.19}$$

its final result, including each initial data element only once.

●

4.2.1. Cost Model for the Adapted n-way Dissemination Algorithm

To really compare the adapted n-way dissemination algorithm to other algorithms, that could be taken as a basis for an allreduce operation, the theoretical runtimes of the n-way dissemination algorithm will be described in this section. The cost of every algorithm for allreduce operations can be split into two different parts: (1) The accumulated message transfer times (T_{comm}) and (2) the accumulated computation times (T_{comp}). The sum of these two will give the worst case total theoretical runtime for each algorithm.
Let T_m be the time needed for transferring one message m over the network and $T_\circ$ be the time needed for the computation of one partial result. The message transfer time of the algorithm incorporates several stages of the message transfer: an sending overhead σ_s, the time spent in the network and the receiving overhead σ_r. Even though we have implemented the above algorithm in an RDMA fashion, the sending and receiving overheads still need to be considered to account for host channel adapter processing times on the communicating nodes. The receiving node will also have to do some polling to check if the data has been written to its memory. This time is included in the overall time with σ_p. The time spent in the network will again be modeled by the ratio of message size and bandwidth plus the latency: $\frac{M}{\beta} + \lambda$ as on p. 11. When transferring more than one message over any given network, contention arises. This contention factor γ can be included in the accumulated message transfer times T_{comm} just like the possible implicit parallelism of a network may be included through a factor $\frac{1}{\phi}$. Both factors will depend on the actual number of messages sent through the network, but the total message transfer time can then be described through

$$T_m = \sigma_s + \frac{M \cdot \gamma}{\beta \cdot \phi} + \lambda + \sigma_r + \sigma_p \tag{4.20}$$

for each message. The importance of including latency times can be seen from Tab. 2.1 on p. 16, where it is shown that latency can immensely reduce in newer interconnects. Similarly, the relevance of the implicit parallelism of a network will become clear throughout the thesis. The second part of the cost model is dependent of the time needed to execute the reduction

operation $\circ$ on two operands. Since the GASPI specification allows arrays as arguments for an allreduce and also defines that in that case the operation will be performed element-wise, the number of elements used in an allreduce will also impact the total runtime of the computation. Let e be the number of elements in the array, then the computation time in each communication round will be $n \cdot e \cdot T_\circ$.
With the above considerations, the maximum theoretical runtime of the n-way dissemination algorithm is

$$T_{n-\text{way}} = n\lceil \log_{n+1}(P) \rceil (T_m + e \cdot T_\circ), \tag{4.21}$$

because in each communication round, each rank writes a maximum of n messages and also computes a maximum of n partial results. As stated in the description of the algorithm, the number of communication rounds is $\lceil \log_{n+1}(P) \rceil$.
In Chap. 5, this cost model will receive further attention, when the adapted n-way dissemination algorithm is compared to other potential candidates for a library allreduce routine. The next section will concentrate on a comparison between Bruck's n-port algorithm and the adapted n-way dissemination algorithm.

4.2.2. Comparison with Bruck's Algorithm

The communication schemes of Bruck's algorithm and the (adapted) n-way dissemination algorithm are very similar, as can be seen when comparing the schemes depicted in Fig. 2.15 on p. 30 and Fig. 2.14 on p. 28 respectively. Both will transfer a maximum of n messages per communication round and will have $k = \lceil \log_{n+1}(P) \rceil$ communication rounds to go through. And both can only be used for associative and commutative reduce operations - like all algorithms with a butterfly-like communication scheme. Nonetheless, there are several differences, on which this section will concentrate.
While the adapted n-way dissemination algorithm presented in this chapter is an adaption to an already existing algorithm, Bruck's algorithm was a completely new algorithm. Bruck's goal was do design a new allreduce algorithm that was able to efficiently use the available n ports of his message-passing system, e.g., the Connection Machines CM-2 or the CM-5 [9]. The dissemination algorithm on the other hand was originally designed as a barrier algorithm and then adapted to be usable as an algorithm for allreduce operations. The number of messages n to be transferred per round is subordinate in this adaption, because this adaption does not aim at fully loading n ports but rather at finding a performant n for a given network, if possible. This results in Bruck's algorithm being applicable for all pairs (n, P), while the n-way dissemination algorithm is not adaptable for all of these pairs.
Another difference between the two is the messages transferred in each communication round. In the n-way dissemination algorithm, there is only one transferred message that is different from the rest. In the first $k-1$ rounds, every message transferred from one arbitrary but fixed

process to its n peers is identical, i.e., the newest partial result. Only in the last round, one of the partial results might be different. As this partial result is known from the very beginning, it may be stored during the round in which it is computed, thus not introducing additional computation cost in the last round. As stated in the last section, the maximum number of computations will thus be $n \cdot k \cdot e$. In Bruck's algorithm on the other hand, two different partial results are computed in each communication round. This will result in a total maximum of $(n+1) \cdot k$ computations in Bruck's algorithm, i.e., k additional computations. Applying this to a GASPI allreduce, where a group may have at most 65535 members and an allreduce array may have 255 elements, this may lead to a worst-case additional $4080 = 16 \cdot 255$ computations, when $n = 1 \Rightarrow k = \lceil \log_2(65535) \rceil = 16$. This might negatively impact the overall runtime, especially with a very time intensive, user-defined reduction operation. In the field of HPC, this can already make an important difference.

The next section will show experimental results of the adapted n-way dissemination algorithm in comparison to the native GPI2-1.0.1 algorithms and native MPI implementations. A direct comparison between the n-way dissemination algorithm and Bruck's algorithm as the basis for allreduce operations will be made in Chap. 5, where several more algorithms have been implemented in the scope of a GASPI library for collective operations.

4.3. Experimental Results

For first results regarding the usability of the adapted n-way dissemination algorithm for an allreduce, the n-way dissemination algorithm was implemented as a GPI2 routine within version 1.0.1. The runtime of the algorithm was measured by one measurement right before calling the allreduce and one measurement right after the return of the routine.

The experiments were conducted on different machines. The first and smallest test system was the Aenigma, equipped with dual-socket 6-core Intel Westmere X5670 @2.93 GHz nodes which are connected through an IB Quad Data Rate (QDR) interconnect in a fat tree topology. The second system the algorithm was tested on was CASE, a system with dual-socket 12-core Ivy Bridge E5-2695 v2 @2.4 GHz nodes, connected through an IB Fourteen Data Rate (FDR) network, also with a fat tree topology.

Different aspects have to be considered when using the n-way dissemination algorithm. For example, the choice of n not only depends on P, but also on the underlying network. As described in the previous section, the runtime of the algorithm depends on the message transfer times, which again are influenced through bandwidth and implicit parallelism of the network. Thus, the choice of n was investigated in preliminary tests on Aenigma. For this test, the n-way dissemination algorithm was implemented with the sum as the reduction operation with one integer as initial data of each node. For each number of ranks, the algorithm was run 10^6 times for each possible n. Figure 4.4 shows the importance of a good choice of n, i.e., the one that will result in the fastest allreduce algorithm.

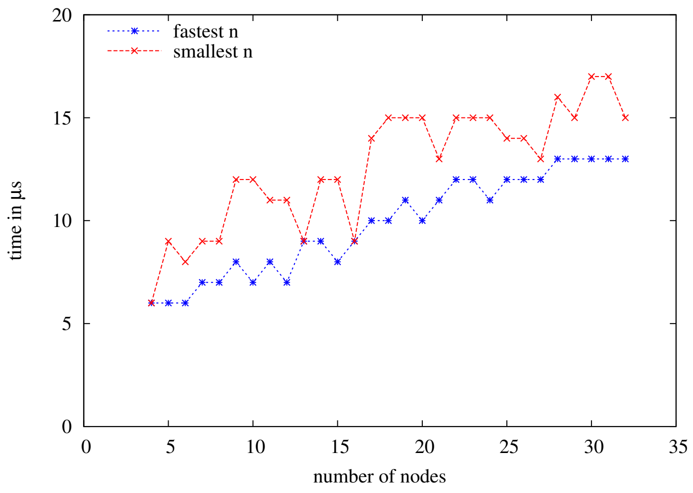


Figure 4.4.: Comparison of n-way dissemination algorithm average runtimes with smallest possible n and fastest n on Aenigma.

The average runtimes of the n-way dissemination algorithm with the smallest possible n for the adaption are compared to those of the fastest possible n on Aenigma. In the worst cases, i.e., for 8 and 10 nodes, the runtime increases by more than 57%, showing the importance of a well chosen n. Thus, the next step to investigate is the question of how to choose n. In a first approach the choice of n was implemented within the allreduce. In most applications, an allreduce will be used several times during the runtime for the same group. It is hence possible to test the allreduce with different possible n, time these runs and take the n with the lowest average runtime for the following allreduces. In the first run of an allreduce, all possible n were run 10 times and the one with the lowest average runtime was then chosen for the following allreduces.

This introduces a high overhead for the first usage of the allreduce, but as seen in Fig. 4.5 for Aenigma, the average runtime of the allreduce after 10^5 tries is still much better than those of the native GPI2-1.0.1 allreduce and even the allreduces of the two available MPI implementations MVAPICH 2.2.0 and OpenMPI 1.6.5. The native GPI2 allreduce is implemented as a binomial spanning tree. The figure shows the results for an allreduce with 255 doubles as the workload. This is the maximum defined by the GASPI specification and thus used as the maximum message size for experiments with collective operations.

Figure B.1a in App. B show the results for the same test with only one integer as workload. Also there, the n-way dissemination algorithm shows faster runtimes than the native GPI2 algorithm, but the MPI implementations are approximately on the same level as the n-way allreduce. This comparison between the different message sizes emphasizes the relevance of the results with the n-way dissemination algorithm, because congestion of the network is to be expected for algorithms with butterfly-like communication schemes and large messages.

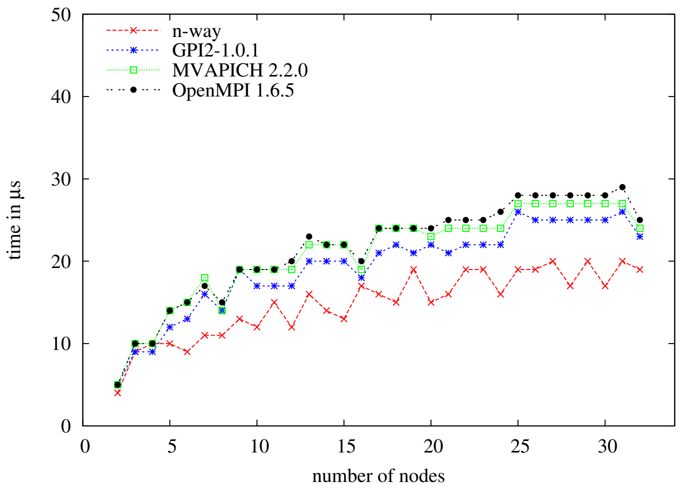


Figure 4.5.: Comparison of different allreduces on Aenigma with 255 doubles and sum as reduce operation.

Figure B.1b in App. B additionally shows that the usage of an adapted n-way dissemination algorithm also makes sense for barriers, because the n-way dissemination barrier is faster than the other three barriers (GPI2-1.0.1, MVAPICH 2.2.0 and OpenMPI 1.6.5).

The same tests were conducted on CASE, where a faster interconnect and newer processors are installed. Instead of the QDR interconnect connecting the compute nodes of Aenigma, CASE is equipped with a FDR interconnect. This has a higher bandwidth and a lower latency (see Tab. 2.1 on p. 16), which is expected to be seen in the results. Even though all algorithms benefit from the newer interconnect, the n-way dissemination algorithm should profit more from the higher bandwidth, because it sends more messages per communication round through the network than, e.g., the native binomial spanning tree or dissemination algorithm of GPI2. On this cluster, the choice of n was done in the same way as described above, i.e., in the first executed allreduce. Different from the tests on Aenigma, the allreduces were not run 10^5 times but only 10^3 times, because of time constraints. Also different from the tests on Aenigma is the MPI implementation available. On this system, IntelMPI 4.1.3 was the fastest available MPI implementation and thus the n-way dissemination algorithm was compared against it.

Figure 4.6 shows the results of the allreduce tests with one integer as workload and a summation as the reduction operation. The `MPI_Allreduce` shows similar runtime results as the native GPI2 implementation. The implementation of `gaspi_allreduce` with the n-way dissemination algorithm on the other hand is significantly faster. For 8 nodes and more, the n-way dissemination algorithm is continuously faster, reaching a peak at an speedup of over 46 % for 48 nodes. In most other cases, the n-way dissemination algorithm is between 25 % and 37 % faster than the MPI implementation.

The difference between the MPI barrier and an n-way barrier is even more significant, reaching over 58 % faster runtimes for 64 nodes, as seen in Fig. 4.7. Overall, the n-way dissemination

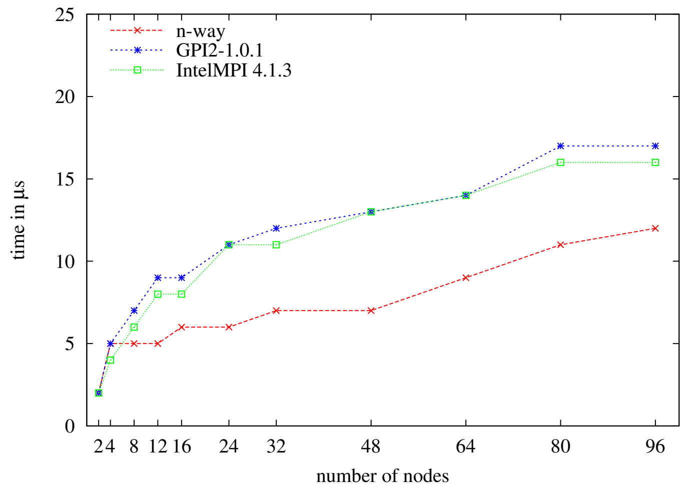


Figure 4.6.: Comparison of average runtimes of the allreduce operation with 1 integer and sum on CASE.

algorithm is faster by continuously more than 42 % when using 8 nodes or more. Different from the allreduce, also the GPI2 barrier is much faster than the MPI barrier with 11 % to over 41 % of performance increase.

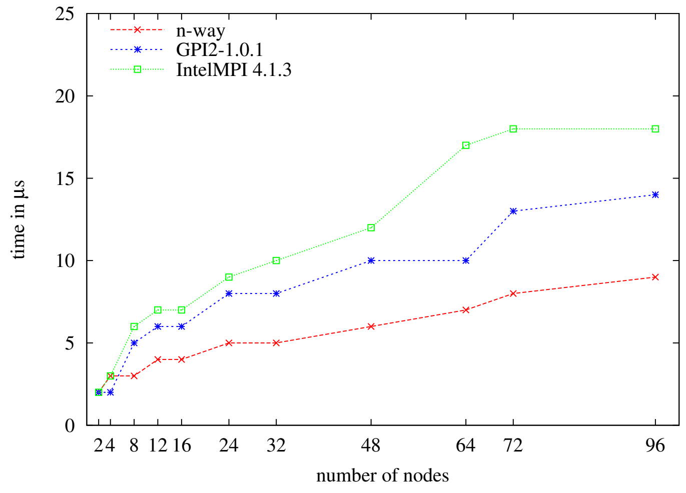


Figure 4.7.: Comparison of average runtimes of the barrier operation on CASE.

The results confirmed the expectation, that a network with a higher bandwidth will have an immense impact on the runtimes of the n-way dissemination algorithm in comparison to the binomial tree algorithm or the original dissemination algorithm, which are used to implement the GPI2 allreduce, respectively the barrier.

4.4. Discussion

This chapter has introduced an adaption to the n-way dissemination algorithm, which enables the use of this algorithm for the allreduce operation. The algorithm is well suited for a split-phase allreduce, as defined in the GASPI specification, due to its low number of communication rounds, especially in comparison to tree-based algorithms. In addition to that, it involves all ranks in the computation of partial results in each communication round. This reduces the imbalance introduced in tree-based algorithms, where all ranks enter the routine but most ranks will not have any computation or communication to be done and thus idle.

In Sec. 4.3, the experimental results show that the performance of the algorithm is highly dependent on the underlying network and the general system configuration. On Aenigma, the system with a lower bandwidth, higher latency interconnect, the benefits from transferring multiple messages per communication round are not reflected in the runtime results. On CASE on the other hand, these effects are very well visible. Considering the development of networks and the further increase of available bandwidth in near future, the transferal of multiple or larger messages per communication round will be playing a more important role. Algorithms with symmetric communication schemes, like the butterfly algorithm, that have congested the network with fairly small messages in the past, will have to be reconsidered for future implementations of collective communication routines.

Another interesting observation to make is the change of differences in runtimes from the barrier to the allreduce operation. Again these observations are only considering runtimes for more than eight nodes, because there is no meaningful observation to be made for smaller numbers of nodes. While the n-way dissemination algorithm is at least 42 % faster than the MPI barrier on CASE, the difference between the MPI allreduce and the n-way allreduce is only at 25 % even though the algorithm has not changed. This change might be due to a change in algorithms in the IntelMPI implementation, i.e., a different algorithm might be implemented in the barrier than in the allreduce operation. It is a usual practice to use several algorithms, depending on message and group sizes [98], so it is permissible to assume the same for a highly optimized MPI implementation as IntelMPI 4.1.3. Another possibility for the reduction of runtime differences might be a different approach to computing the partial results. Both possibilities should be taken into consideration for future research on this topic.

In the experiments in this chapter, the choice of n was made by running the allreduce multiple times with all possible n and then choosing the one with the lowest runtime. While this procedure works well for a limited number of n to test and applications with intensive use of the allreduce operation, it will not be applicable on systems with very high bandwidth, where potentially ten or more different n can be tested. The first allreduce will have a considerably higher runtime than the following allreduces, which cannot be balanced in applications, where only few allreduces are needed. In addition, this procedure is prone to errors through jitter in the network or single higher runtimes due to congestion in the network or contention on the resources. Instead, other options have to be investigated for the choice of n. This could either be some runtime checks during the installation and configuration of GASPI or some environment variables given by the user at the time of installation. In dependence of variables like topology, interconnect type, bandwidth or latency, static but network-specific lookup tables could be generated for the choice of n.

Another point for future research might be the restriction imposed by all butterfly-like algorithms when used for an allreduce operation: they may only be used for associative and commutative reduction operations, if the result needs to be identical on all participating nodes. This limits the usage of algorithms like the adapted n-way algorithm or Bruck's algorithm to maximum or minimum operations as well as sums or products of integers. This issue can be resolved by using multiple underlying algorithms for an allreduce, that are chosen based on not only message size and group size, but also based on the datatype and operation. Another option would be to further adapt these algorithms to internally order the computation of the partial results. This adaption would also imply increasing message sizes in the communication rounds and thus a further parameter to consider when choosing the number of messages to transfer per round.

Further on, different network topologies and interconnects can have a significant impact on the runtime of algorithms used for collective communication. Thus, future work should deal with investigating this impact within GASPI applications. To do so, further implementations of GASPI will have to be implemented and made accessible on different systems.

The following chapter will include the results presented in this chapter in the implementation of an allreduce routine for a GASPI library for collective communication routines. In addition to the comparisons made in this chapter, the next chapter will compare the n-way dissemination algorithm to other allreduce implementations. Apart from the allreduce, also reduce and broadcast routines have been implemented for the collective library and will hence be presented in the following chapter.

5. GASPI_COLL - Collective Communication Routines for GASPI

As described in Sec. 2.3.2, many different collective communication operations exist. While many of these are defined and implemented in the MPI standard, GASPI only defines the barrier and the allreduce operation. As other collective routines are also demanded and in use by application programmers, a collective communication library on top of GASPI has been designed: GASPI_COLL. This library extends the GASPI standard with several additional collective communication routines but also offers an allreduce with potentially different algorithms than used in the implementation of the specification. As this library is designed on top of GASPI, it is portable to every machine or system with a GASPI installation.
All algorithms used have been introduced in Sec. 2.3.3. Here, only additional adaptions to the existing algorithms and the reasons for the choice of the given algorithms will be explained. The semantic of GASPI_COLL closely follows that of GASPI, e.g., a call to `gaspi_coll_allreduce` will take the same arguments as a call to `gaspi_allreduce`. In addition the limits imposed by the GASPI specification are adopted, i.e., the maximum number of elements an array in an allreduce may have and the maximum number of members in a group. Special emphasis needs to be put on the restriction imposed by the specification that no two collective routines of the same type may run at the same time. This transfers much management overhead from the library to the application programmer.
The following sections will describe the group management routines in Sec. 5.1 and the memory management in Sec. 5.2. The implemented collective routines are described in Sec. 5.3. For each collective routine, runtime comparisons with given MPI implementations have been made. Because the GASPI implementation GPI2 does not include all collective routines implemented in GASPI_COLL, runtime comparisons were only made where applicable. The results are presented in Sec. 5.4, before Sec. 5.5 discusses the results and summarizes this chapter.

5.1. Group Management

Because GASPI_COLL is implemented on top of GASPI, only some parts of the provided group structure can be re-used while other parts need to be newly implemented. Thus each group used in GASPI will also have to be created in GASPI_COLL if any of the routines provided in GASPI_COLL are to be used in that group. This will be done through `gaspi_coll_group_create`.

```
typedef struct{

  gaspi_number_t groupSize;
  gaspi_rank_t *groupMembers;
  gaspi_rank_t myID;

  gaspi_pointer_t segmentPtr;
  gaspi_segment_id_t segmentID;

  int allreduce_ctr;
  int allreduce_status;
  int allreduce_start_data_offset[2];
  nway_struct nway_allreduce;
  brucks_struct brucks_allreduce;
  pairwise_struct pw_allreduce;
  tree_struct binomial_allreduce;

  int reduce_ctr;
  int reduce_status;
  int reduce_start_data_offset[2];
  tree_struct binomial_reduce;

  int broadcast_ctr;
  int broadcast_status;
  int broadcast_start_data_offset[2];
  tree_struct binomial_broadcast;

  int alltoall_ctr;
  int alltoall_status;

}gaspi_coll_group;
```

Listing 5.1: Definition of group structure in GASPI_COLL.

When the group is not needed anymore, `gaspi_coll_group_delete` will free all previously allocated resources. This means that the application programmer is in full control of the overhead introduced by GASPI_COLL.
The group structure of GASPI_COLL is defined as shown in List. 5.1. The internal usage of such a newly defined structure mainly serves the purpose of not repeatedly calling the same GASPI routines from within the library. In addition, application programmers will most likely repeatedly use collective routines in the same group, thus, information necessary for the implementation of the different algorithms is stored in further structs, listed in App. C. This circumvents the re-calculation of, e.g., communication peers in each call of the collective routine. Each rank of the group has its own copy of the group struct and the included algorithm structs.
The `group_size`, `*group_members` and `myID` are mainly needed for the calculation of the communication peers in the initialization phase of the collective routines and for the communication within the routines. Each group will create own segments, to and from which the communication within the routine will be done through the GASPI communication routines. To access the

segment and to hand the correct segment ID to the GASPI communication routines, `segmentID` and `*segmentPtr` are needed.
The different counters for the collective routines are especially needed for determining the correct communication buffers. Even though the library implementation sticks to the restriction that two collective routines of the same type may not be used concurrently, two succeeding collective routines may very well have some overlap in communication, as Fig. 5.1 shows.

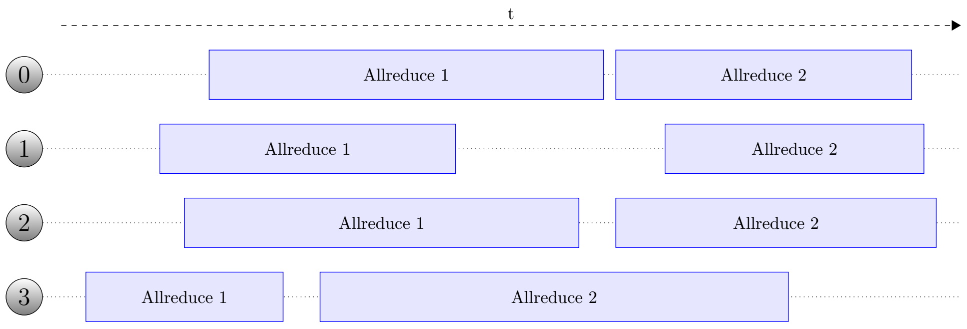


Figure 5.1.: Two succeeding allreduces, where the second allreduce on rank 3 overlaps with the first allreduce on ranks 0-2.

To ensure that two succeeding collective routines do not overwrite each others communicated data, collective routines with an odd counter write into a different buffer than those with an even counter. If the library did not take care of this, the user would be forced to use a barrier between every two uses of the same collective routine, which would not conform with the goals of GASPI and a performant, scalable application. Accordingly the values in `*_start_data_offset` give the offsets, where the odd and even data buffers start.A more detailed description of the memory partitioning is found in Sec. 5.2.

5.2. Memory Management

The implementation of a library for collective communication routines as GASPI applications introduces memory specific management overhead. The used memory resources will be shared with the actual application, thus it is also necessary to keep the memory requirements of a collective communication library as low as possible while still employing the GASPI-defined semantic. A GASPI-based communication library will use the communication routines defined by the GASPI specification to ensure compatibility with all GASPI implementations, especially one-sided communication routines with weak synchronization primitives. Using one-sided communication routines within the GASPI_COLL routines, makes it necessary to manage not only memory accesses but also the notification buffers.
The semantic of collective routines in GASPI can only be deduced from the definition of the allreduce operation. Different from other communication routines, where an offset on an already

registered memory segment is handed to the routine, the allreduce takes a pointer to the actual data to be reduced as an input argument, as well as a pointer to the location where the result should be stored. This directly implies a copy of the data to an internal memory segment which is registered for one-sided communication. In a library, which will use GASPI routines, this means a GASPI memory segment of sufficient size needs to be allocated and registered internally. This segment is also necessary for the weak synchronization of the communication routines which also involves the internal handling of message notifications.
Another important property of collective communication routines imposed by the GASPI specification is the limitation that no two collective routines of the same type may be run concurrently within the same group. As described through Fig. 5.1, this does not guarantee, that write accesses of succeeding collective routines do not overlap. The allocated memory segment for collective communication routines is thus structured such that for all routines, two buffers are available: one for the routines started with an odd counter and one for the routines started with an even counter, as depicted in Fig. 5.2. This prevents the overwriting of data from two succeeding collective operations.

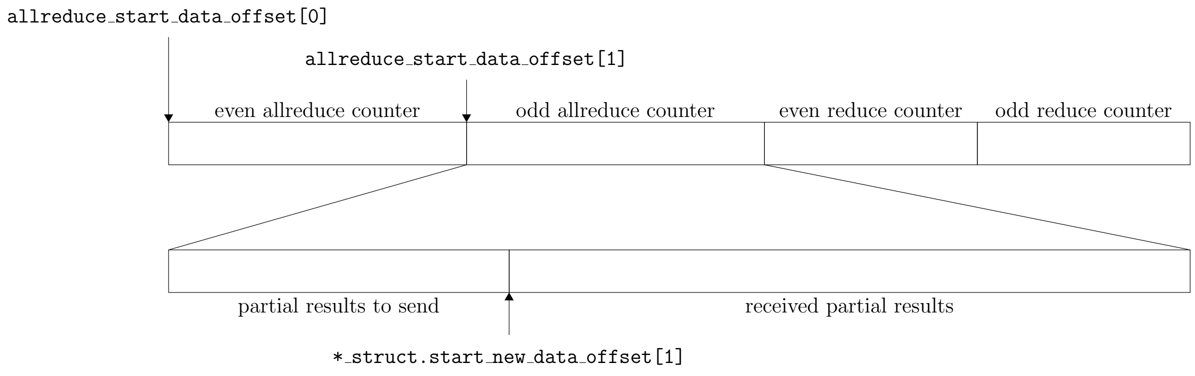
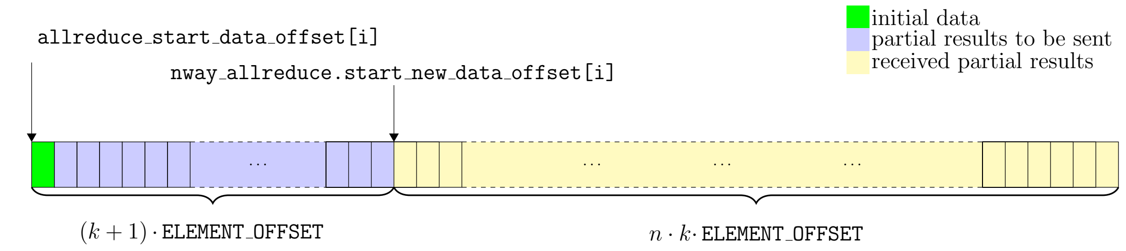


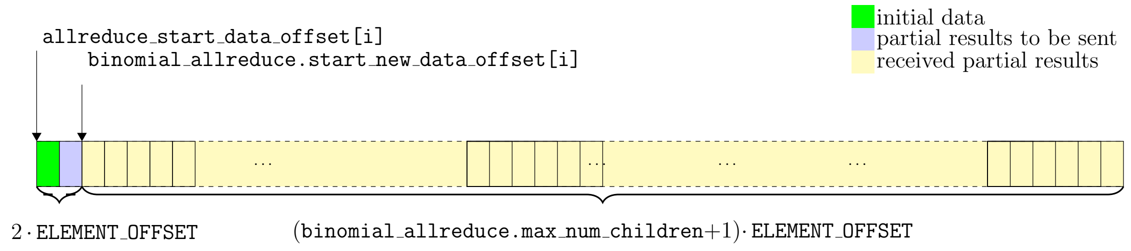
Figure 5.2.: Partitioning of the collective segment for a group in GASPI_COLL.

Each collective routine in GASPI_COLL will have its own memory partition and according offsets, which need to be used in the one-sided communication operations. The offsets are always calculated in dependence of the maximum possible buffer size used in collective communication: 255 doubles. This limit has been adopted from the GASPI specification of the allreduce and will also hold for the reduce and the broadcast operation. Through this, a fixed limit to the maximum size of an internal memory segment per group is given. In addition to the message size, the number of communication rounds has to be considered for the allocation of a sufficiently large internal segment.
Exemplarily, Fig. 5.3 shows the partitioning of the allreduce segment for the n-way dissemination algorithm (Fig. 5.3a) and for the BST (Fig. 5.3b). Let n be the number of messages sent per round and the number of rounds is $k = \lceil \log_{n+1} P \rceil$. The portion of the segment dedicated to allreduces is then used by the n-way dissemination algorithm as follows: The initial data of the rank is copied to the very beginning of the segment (green). In every round,

the received partial results are written to the end of the segment, starting at offset $(k+1)\cdot$ `ELEMENT_OFFSET`. The total memory requirement for Bruck's algorithm with n messages per round is thus $(\lceil \log_{n+1} P\rceil \cdot (n+2)+2)\cdot$ `ELEMENT_OFFSET`.


initial data
partial results to be sent
received partial results
allreduce_start_data_offset[i]
nway_allreduce.start_new_data_offset[i]
$(k+1)\cdot$ ELEMENT_OFFSET
$n\cdot k\cdot$ ELEMENT_OFFSET

(a) n-way dissemination algorithm


initial data
partial results to be sent
received partial results
allreduce_start_data_offset[i]
binomial_allreduce.start_new_data_offset[i]
$2\cdot$ ELEMENT_OFFSET
(binomial_allreduce.max_num_children$+1)\cdot$ ELEMENT_OFFSET

(b) BST

Figure 5.3.: Partitioning of the allreduce segment for different algorithms.

The binomial spanning tree has much smaller memory requirements than the n-way dissemination algorithm or Bruck's algorithm. The array needs to hold the initial data, the partial results received by the children, the newly calculated partial result and the final result received by the parent. The classical binary tree has at most 2 children and a binomial spanning tree at most $\lceil \log_2(P)\rceil$ children. The total memory requirement is $(\lceil \log_2(P)\rceil+3)\cdot$ `ELEMENT_OFFSET` bytes.

The notification buffer of every GASPI segment is limited by the implementation, in case of the GPI2 implementation, this limit is set to 65535 notifications. This number is high enough for all implemented collective communication routines at the moment. To be used with other GASPI implementations in the future, a query within the group creation will have to be done to check the number of available notifications per segment. In dependence of the retrieved number, different steps will have to be taken, including the exclusion of certain algorithms that need too many notifications, or the allocation of multiple internal segments instead of only one per group.

5.3. Collective Routines

This section will shortly describe the routines implemented in GASPI_COLL and the underlying algorithms, before the next section will show experimental results with these algorithms.

The broadcast routine also sticks to the semantic and limitations given by the GASPI specification, as far as applicable. The initial data of the participating ranks are handed to the routine via pointers and are internally copied for further use within the collective routine. The same is true for the result buffer: the address is handed to the routine and the final result will be written into this location by the reduce routine. Another limit imposed by the GASPI specification is the size of the message buffers.

Allreduce

The allreduce has been implemented with different underlying communication algorithms, all of which have been described in Sec. 2.3.3 and Chap. 4. Depending on message sizes, data types, reduction operation and group size, the algorithm to be used can be chosen. The BST algorithm (p. 25) can be used for any kind of reduction operation and data type. The experiments in the next section will show, for which message sizes and group sizes the algorithm is most performant. The PE algorithm (p. 27), the adapted n-way dissemination algorithm (Chap. 4) and Bruck's algorithm (p. 29) can not be used for non-associative routines, but may show better results when used, e.g., for maximum or minimum operations or the barrier.
As already mentioned above, the GASPI_COLL allreduce will take the same arguments as the original GASPI allreduce but will have the GASPI_COLL prefix `gaspi_coll_`:

```
gaspi_coll_allreduce(gaspi_pointer_t buffer_send,
                     gaspi_pointer_t buffer_receive,
                     gaspi_number_t num,
                     gaspi_operation_t operation,
                     gaspi_datatype_t datatype,
                     gaspi_group_t groupID,
                     gaspi_timeout_t timeout);
```

Listing 5.2: GASPI_COLL allreduce routine.

This leads to an easy adaption of an application code and less hassle for the programmer when testing the library allreduce instead of the native GASPI allreduce.

Reduce

Unlike the allreduce, where all participating ranks have the final result upon successful return of the operation, this only holds true for the root rank in the reduce operation. All ranks contribute their own data to be reduced into a final result, which the root rank will then have. Because the root rank also contributes data, a source buffer and a destination buffer are necessary for the reduce routine, both of which do not have to lie within a registered segment.

```
gaspi_coll_reduce(gaspi_rank_t root,
                  gaspi_pointer_t buffer_send,
                  gaspi_pointer_t buffer_receive,
                  gaspi_number_t num,
                    gaspi_operation_t operation,
                    gaspi_datatype_t datatype,
                    gaspi_group_t groupID,
                    gaspi_timeout_t timeout);
```

Listing 5.3: GASPI_COLL reduce routine.

To complete the reduction, the user needs to specify which reduction operation is to be used. The predefined reduction operations are the same that are specified in the GASPI specification: sum, minimum and maximum. Additionally, the user needs to specify the datatype and number of elements to be reduced. From these two arguments the message size will be internally calculated for the transferal of data. If the number of elements to be reduced is larger than 1, the reduction operation will be applied element-wise.
A return of the operation with `GASPI_SUCCESS` on any non-root rank implies that the work to be done by this rank has been completed and the local buffers may be reused. If the rank is a leaf node in the underlying binomial spanning tree, the work consists of posting a write request to the internal queue. If the the rank is an inner node, the work consists of waiting on the data to be received from the child nodes, computing a partial result and transferring this partial result to the parent node. For the root rank, the successful return implies not only, that all ranks have finished their work in the routine, but also that the final result will be available in the receive buffer. That the successful return of the routine only makes implications on the local progress is the typical GASPI semantic, posing a problem together with the limitation of the GASPI specification that two collectives of the same type may not run at the same time. This limitation has also been posed on the allreduce and barrier routines and can there be solved through internal double buffering. As shown in Fig. 5.1, two succeeding allreduces may very well overlap, because the successful return of one allreduce only makes implications on the local status of the allreduce - other ranks may still be involved in the communication or computation of some result. It is not feasible for a third allreduce to start before the first one has been completed though - always assuming that the user does not call the allreduce before ensuring that the previous ones have locally completed their work and communication. The second allreduce will stall as long as the first allreduce has not finished on all nodes. The successful return of the second allreduce will thus not only make implication on the local status of this allreduce, but will also imply that all ranks have completed the first allreduce and thus the internal buffers of the first allreduce can be reused without overwriting data that is still needed. For the reduce routine, this is no longer true, because the step disseminating the final result is missing. Several reduces could thus interfere internally and overwrite data that is still needed by a previous reduce.
To encompass this, GASPI_COLL will offer a second routine with an acknowledgment mechanism additionally implemented: `gaspi_coll_reduce_ack`.

```
gaspi_coll_reduce_ack(gaspi_rank_t root,
                      gaspi_pointer_t buffer_send,
                                 gaspi_pointer_t buffer_receive,
                                 gaspi_number_t num,
                                 gaspi_operation_t operation,
                                 gaspi_datatype_t datatype,
                                 gaspi_group_t groupID,
                                 gaspi_timeout_t timeout);
```

Listing 5.4: GASPI_COLL reduce routine with acknowledgment.

Not only does this routine notify the other ranks of the group when a reduce is completed, but it also checks whether the last reduce with an odd respectively even counter has finished on all nodes before starting the new reduce with an odd/even counter. This does not comply with the usual GASPI semantic because the routine introduces a synchronization point, but enables the programmer to use the reduce operation without needing to introduce many barriers.
The GASPI_COLL reduce operation has been implemented with the BST algorithm. The other algorithms discussed for an implementation of the allreduce operation are not suitable for a reduce, because not all participating nodes will need the final result. This would automatically be the case when using the n-way dissemination algorithm, Bruck's algorithm or the PE algorithm and introduce additional, unnecessary communication and computation. These algorithms might very well be suitable for implementing the acknowledged version of the reduce operation though.

Broadcast

The GASPI_COLL broadcast routine follows the GASPI specification in that the source and destination buffers do not have to lie within a registered segment and that the element size may be at most the number retrieved by `gaspi_allreduce_elem_max` times the size of a `double`. A limitation of the buffer size is necessary for the internal memory management. Using the chosen message size keeps the maximum message size for all collective communication routines identical.

```
gaspi_coll_broadcast(gaspi_rank_t root,
                     gaspi_pointer_t buffer,
                     const gaspi_size_t size,
                     const gaspi_group_t groupID,
                     const gaspi_timeout_t timeout);
```

Listing 5.5: GASPI_COLL broadcast routine

The root rank has some data in its buffer to be distributed, for all other ranks in the group, this buffer will be the location where the received data is stored when the broadcast is finished. Different from MPI, only one buffer address is given to the routine instead of a source and a destination buffer, because only one buffer is necessary on all ranks. The buffer will need to have the size given to the routine on all ranks, i.e., enough space to hold all data transferred. If

the buffer allocated by the user is too small, the routine will overwrite data that overlaps with the buffer. The routine will return with `GASPI_SUCCESS` when the data to be broadcast has been queued for transferal and copied into the buffer. A successful return makes no implication on the status of other ranks involved in the communication or on the remote write.
Through the last property, which is strongly encouraged by the GASPI semantic, the user will run into additional necessary synchronization points within his application to ensure that no two broadcasts run at the same time and start overwriting the data from previous broadcasts, as already described for the reduce above. To encompass this, GASPI_COLL will offer a second broadcast routine with additional acknowledgment of receiving nodes. All ranks of the broadcast will be notified by the leaf nodes of the underlying binomial spanning tree, when they have received the data. This acknowledgment also implies that all other ranks have finished the broadcast and the next broadcast can safely be started.

```
gaspi_coll_broadcast_ack(gaspi_rank_t root,
                         gaspi_pointer_t buffer,
                         const gaspi_size_t size,
                         const gaspi_group_t groupID,
                         const gaspi_timeout_t timeout);
```

Listing 5.6: GASPI_COLL broadcast routine with acknowledgment to the root process.

Successful return of this routine will mean the same as a successful return of the standard broadcast, but on the root rank it will additionally mean that all ranks have finished the broadcast. This convenience function is included as a compromise between the notification of all ranks, comparable to a weak barrier, and the notification of no rank at all, forcing the user to use a barrier.
Like the reduce operation, the broadcast was implemented with a BST algorithm to circumvent additional communication and computation overhead that would be introduced by the other algorithms that are used for an implementation of the allreduce routine. Similar to the reduce case above, the usage of these algorithms disseminating information among all participating processes, might be very useful for the implementation of the acknowledged version of the broadcast routine.

Barrier, Scatter and Gather

The barrier, scatter and gather operations have not been implemented in the GASPI_COLL library. The internal barrier of the GPI2 can not be beaten by an externally implemented barrier. It might be worthwhile implementing the barrier with a different algorithm, e.g., the n-way dissemination algorithm in future GASPI implementations.
The GASPI specification defines the `gaspi_read_list` and `gaspi_write_list` routines. These routines can easily be used as substitutes for a scatter or gather, are already asynchronous operations and implemented closer to the hardware than a library can get. Thus, scatter and

gather routines have not been implemented so far but it should be investigated, whether it does make sense to explicitly implement these routines on other systems than IB-based networks.

5.4. Experimental Results

The routines described in Sec. 5.3 have been implemented and tested on Aenigma (see p. 71 for further details on the system). In the scope of [26], the allreduce routine has also been tested on MareNostrum III, a system with two sockets nodes of with 8-core Sandy Bridge E5-2670/1600 @2.6GHz processors and an IB FDR-10 network in fat tree configuration. On this system, the algorithm was compared to the allreduce routines of Intel MPI 4.1.3.049. These results are also presented in this section.

The section will show runtime comparisons for the smallest possible message size (one integer) and the largest possible message size (255 doubles) of the implemented GASPI_COLL routines, GPI2 routines, where applicable, and MPI routines. The runtimes shown are average times from 10^4 runs to balance single higher runtimes which may be caused through different deterministically irreproducible aspects like jitter, contention in the network and similar. Timings were taken right before the call and then again immediately after the call returned. Between two calls of an allreduce, a barrier was called to eliminate caching effects. One GASPI process was started per node or per NUMA socket, where the latter is the maximum number of GASPI processes that can be started per node. On both systems, GPI2-1.2.0 was the GASPI implementation used for benchmarking.

To ease the navigation through the experimental results, this section will be divided following the implemented GASPI_COLL routines. First, the results of the allreduce experiments will be shown, followed by the reduce and broadcast results.

Allreduce

To convey an idea of the overhead induced through the implementation of the allreduce as a GASPI library routine instead of implementing the allreduce directly with ibverbs, this overhead is depicted in figure 5.4. The runtime for the allreduce with one integer increases by a factor of up to 1.84 and with 255 doubles, it even increases by a factor of up to 2.18. This will have to be kept in mind, when regarding the following results.

While the BST and the PE transfer a fixed number of messages per communication round, the n-way dissemination algorithm and Bruck's algorithm may transfer different numbers of messages per communication round. Since Bruck's algorithm works for all combinations of (n, P), $n = 5$ was fixed for these experiments. For the n-way dissemination algorithm the n is chosen in the first call of the allreduce routine and the smallest n possible is chosen. This procedure differs from the procedure in Sec. 4.3, where a number of allreduces was started in the first call and the fastest n was chosen. Further research has shown, that the overhead

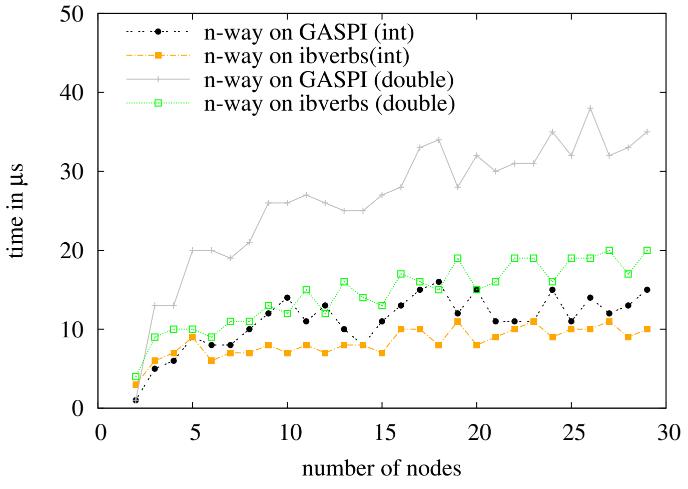
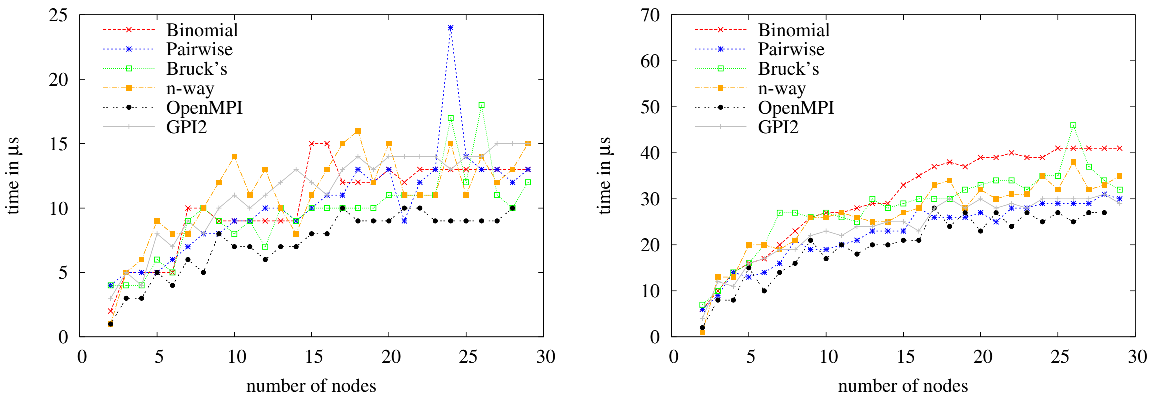


Figure 5.4.: Comparison of average runtimes of the allreduce with sum on Aenigma. Implemented with ibverbs (green and orange) and as a GASPI library routine (black and gray). One GASPI process was started per node.

induced by calling a sufficiently high number of allreduces to chose a n in this first call is not necessarily compensated through the potentially faster following allreduces.

Figure 5.5 shows the comparison of runtimes on Aenigma with the sum as reduction operation, with one integer in Fig. 5.5a and 255 doubles in Fig. 5.5b. In both cases, the Intel MPI allreduce shows the fastest runtimes, but only with large messages the GPI2 implementation is also faster than most library implementations. For small messages, the allreduce runtimes are very unstable, showing that one algorithm may be very suitable for a given number of participating processes but not so much for others. For example, Bruck's algorithm is the fastest library implementation for 15 to 20 processes but the PE is much faster for 21 processes, even beating the Intel MPI implementation.



(a) 1 integer **(b)** 255 doubles

Figure 5.5.: Comparison of allreduce implementations and the sum as reduction operation on Aenigma. One GASPI process was started per node.

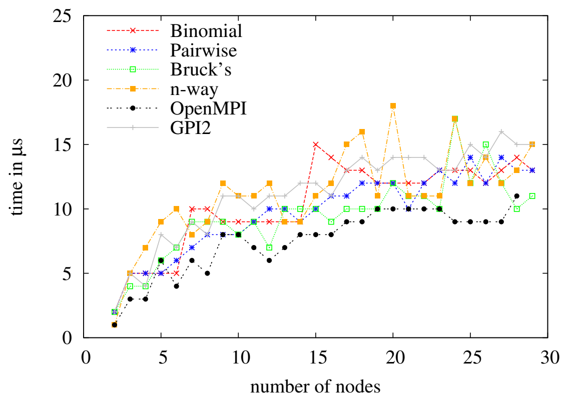
Very interesting is the peak in runtimes that the PE, Bruck's and the n-way algorithm have when used with 24 processes. While the PE algorithm has the highest peak, also the peak in Bruck's algorithm is remarkable because both algorithms had only few outliers when used with smaller node numbers. For the n-way dissemination algorithm, it is not clear whether this peak is one of the many leaps in the runtime plot or if there is really a correlation to the 24 processes as would be expected for Bruck's and the PE algorithm. All in all, Bruck's algorithm and the PE implementation show the best results, such that a combination of these two algorithms should be considered for global sums with small messages.

Additionally notable is the difference in runtime between the GASPI_COLL BST allreduce implementation and the native GPI2 allreduce implementation, which is also implemented with an BST. The GASPI_COLL implementation is in parts significantly faster than the native implementation. On the one hand this reflects the different implementations of the same communication scheme but might also imply a difference in management overhead. To verify the latter point, further tests with a BST implementation, identical to the current GASPI_COLL implementation, within the GPI2 will have to be made.

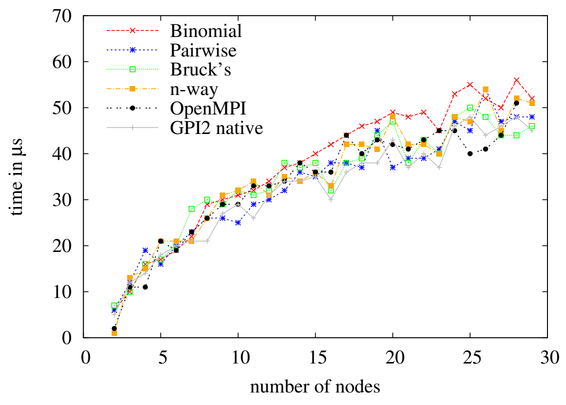
When using larger messages, i.e., 255 doubles in this benchmark, all algorithms show fewer jumps in the runtime plots (Fig. 5.5b). The native GPI2 allreduce implementation is now faster than most library implementations, which is to be expected, considering that this implementation has less function call overhead. The PE implementation shows the fastest average runtimes for almost all group sizes, ranging below the GPI2 runtimes but still above the MPI runtimes. For 15 processes and more, the BST algorithm performs worst of all algorithms, except for a peak of Bruck's algorithm at 26 processes. The peak at 24 processes that was seen in Fig. 5.5a, does not appear again when using large messages. Since Aenigma was used exclusively for this test, an interference of other applications running at the same time could be ruled out. Repetition of the benchmark also showed the same peak every time.

Figure 5.6 shows the averaged runtimes of the same experiments with the maximum used as the reduction operation. Again the tests were conducted for small messages (Fig. 5.6a) and for large messages (Fig. 5.6b). Especially for small messages the observations to be made are very similar to those with a global sum. The MPI allreduce shows the best runtimes and the GASPI_COLL implementations are better than the GPI2 native allreduce. Also the jumps in the runtimes are again very large when using the n-way dissemination algorithm and the runtimes of the PE algorithm and Bruck's algorithm are the overall fastest. Different from the previous experiment, the peak at 24 processes is only visible for the adapted n-way dissemination algorithm and for Bruck's algorithm. The PE algorithm does not show this sudden increase this time. Overall the runtimes dither in the same range as the runtimes of the global sum for small messages.

In much contrast, the runtimes of the global maximum for large messages are considerably higher than those of the global sum. While the maximum runtimes are in the range of 25 to



(a) 1 integer



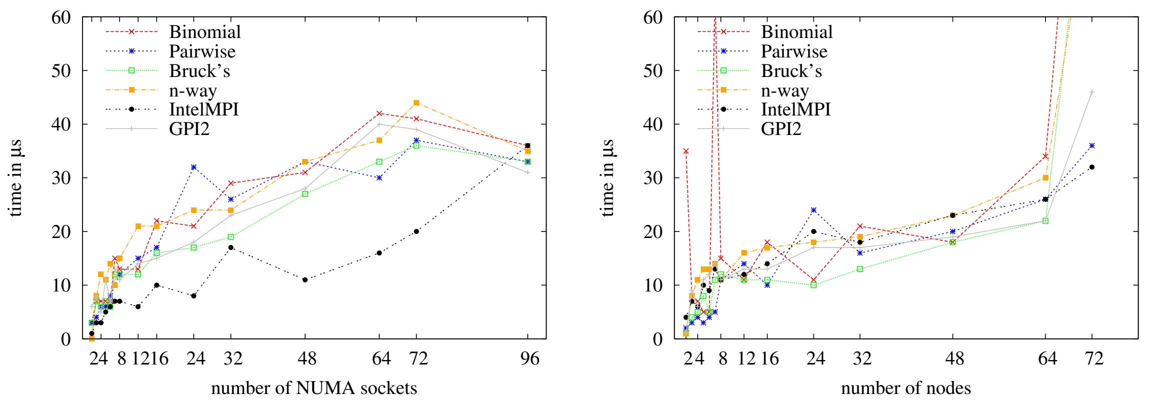
(b) 255 doubles

Figure 5.6.: Comparison of allreduce implementations with the maximum as reduction operation on Aenigma. One GASPI process was started per node.

45 μs for an allreduce with sum, this range moves up to 45 to almost 60 μs for the maximum as reduction operation. Since this is true for all tested allreduce implementations, this increase in runtimes surely comes from the higher runtimes of the maximum operation. For the first time, the native GPI2 implementation is one of the best implementations, while at the same time it is necessary to note that all runtimes are much closer together than in the previous experiments. The MPI runtimes range somewhere in the middle, only showing the fastest runtimes for 25 and 26 processes. Of the GASPI_COLL implementations, no algorithm continuously shows the best runtimes. Apart from the BST algorithm, which is the overall slowest algorithm, the fastest algorithm alternates often. When using the minimum as reduction operation, the results are almost identical to the results presented here for the maximum operation. These results are depicted in Fig. C.1 in App. C.

The same tests were conducted on MareNostrum III, which has a faster IB interconnect and newer, more powerful processors than Aenigma. It is to be expected, that these differences will also be reflected in the results of the same experiments. An important difference to the experiments conducted on Aenigma is the group sizes chosen for the experiments. While it was possible to test every group size on Aenigma, this was not possible on MareNostrum III. This means, the plots shown will show an overall picture of the runtime development for increasing numbers of nodes, but it is not possible to make such fine-grained remarks on possibly erroneous runtimes. The following figures show the differences in runtimes whether one process is started per node or per socket for one integer as payload.

Figure 5.7 shows the averaged runtimes of the allreduce with one integer and the sum as the reduction operation. In Fig. 5.7b the averaged runtimes for one process started per node are shown. There is a sudden and high increase in runtimes of the GPI2, BST, Bruck's and the n-way algorithm at 72 nodes, and additional two peaks for the BST at 2 and 7 nodes. Further tests were not possible in the given timeframe to further investigate on this, so all statements are made with this consideration in mind. Figure 5.7a only shows runtimes up to 60 μs to ease the comparison to the other figures and to enhance the readability of the different plots for node numbers up to 64 nodes, which are very close together. An unenlarged version of this figure can be found in App. C, Fig. C.2.



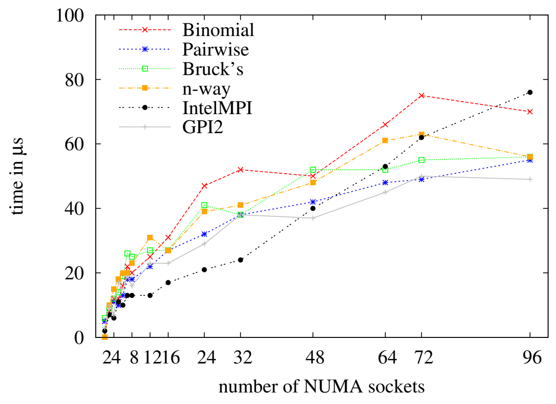
(a) 1 process per socket **(b)** 1 process per node

Figure 5.7.: Comparison of allreduce implementations with sum as reduction operation for one integer on MareNostrum III.

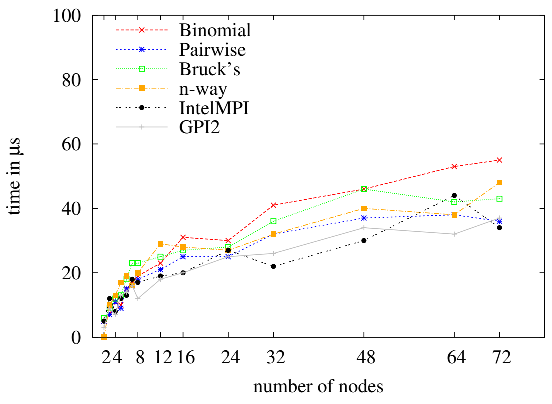
On this system, the advantage of a high-bandwidth interconnect can already be seen for small messages. As Fig. 5.7 shows, the MPI implementation has higher runtimes than the GASPI_COLL implementations with Bruck's algorithm or PE in most cases. Also the native GPI2 implementation shows faster runtimes than MPI but is not as fast as the library implementation with Bruck's algorithm. Again, the PE shows a peak at 24 processes, not only here but also in Fig. 5.7a. The overall runtime trends of the GASPI related algorithms is the same when using one process per socket as when using one process per node: Bruck's algorithm shows the best results, the BST and n-way runtimes form the upper limit and the PE and GPI2 implementations are somewhere in between. Compared to the runtimes of one process per node, they are all slower though. In the first case, it took 64 processes a bit more than 20 μs to complete the allreduce with Bruck's algorithm. In the second case, this runtime increases to almost 35 μs. This stands in heavy contrast to the behavior of the MPI implementation, whose runtimes are much faster when using more processes per node. Here the runtimes seem more dependent on the number of nodes used than on the number of processes started per node.
In App. C, the runtime results for the minimum and maximum operations are shown. Figures C.3a and C.4a show the results with maximum operation with one integer with one process per socket, respectively with one process per node started. The runtimes of GASPI_COLL allreduce routines are nowhere close to the MPI implementation when starting one process per socket, but clearly overtake the MPI and the GPI2 implementation when started with one process per node. Again there is a sudden jump in runtimes for 72 nodes, even though it is not as extreme as when using the sum as reduction operation. Overall, the same can be said for the

experimental results from the minimum operation, shown in Figs. C.3c and C.4c, emphasizing the importance to know whether to start one process per node or per socket when using certain communication libraries.

Figure 5.8 shows the results of the experiment with 255 doubles instead of one integer as payload and the sum as reduction operation. Again, the runtimes of all allreduce implementations are much faster, when starting one process per node instead of starting one process per socket. Looking at the case with one process started per socket, there is a significant difference to be observed from the above case: the MPI implementation does not show significantly better runtimes for all node numbers in Fig. 5.8a. For large messages, the PE and Bruck's allreduce implementations in GASPI_COLL show better runtimes than the MPI implementation from 64 processes on. The PE allreduce implementation is overall the fastest implementation of the GASPI_COLL allreduces, but another important observation to make is the much better performance of the adapted n-way dissemination algorithm, when compared to its performance with small messages. The runtimes are constantly better than those of the BST allreduce, which shows the overall worst runtimes for 16 processes and more. A small exception to this is seen for 48 processes, where Bruck's algorithm performs slightly worse.



(a) 1 process per socket



(b) 1 process per node

Figure 5.8.: Comparison of allreduce implementations with sum as reduction operation for 255 doubles on MareNostrum III.

Figure 5.8b shows the runtimes of the same experiments with one process started per node. For higher process numbers, all implementations are again faster than when started with one process per socket. Significantly, this is not true for small node counts and the MPI implementation. The other observations made for the comparison of the different implementations in Fig. 5.8a hold true for this case. Especially, there is no jump in the runtimes between 64 and 72 nodes, as was previously observed for small messages in Fig. 5.7b. Again, these results translate directly into a recommendation of GASPI application programmers to start one process per node instead of one process per socket.

The results of the allreduce operation with 255 doubles for the maximum and minimum can be found in App. C in Figs. C.3b, C.4b, C.3d and C.4d. The first two figures show the results of the maximum operation with one process per socket, respectively one process per node and the latter two figure accordingly show these results for the minimum as reduction operation. In general, the same statements just presented for the sum operation can be made for the maximum and minimum operation as well. Different from the results on Aenigma, the maximum and minimum operations do not induce a significantly higher runtime, which is probably due to the newer processors available on MareNostrum III. The results of the minimum operation show some significant irregularities, which need to researched further. For example, the MPI implementation showed a significant peak at 48 nodes, when using the minimum operation and starting one process over socket (Fig. C.3d). This peak is not seen in the GASPI-based implementations, indicating a sudden and temporary contention in the system. Similarly, the minimum operation showed worse runtimes than the maximum operation for more than 32 nodes - seen in the comparison of Fig. C.4b with Fig. C.4d. Considering the similarity of the runtimes of these comparing routines in the other experiments, again some kind of hardware contention is implied. These results thus not only show the relevance of the system configuration and hardware on applications, but also the influence of load on the system on each and every application.

The following experimental results of the reduce and the broadcast operations were only conducted on Aenigma, because MareNostrum III was not available for these experiments.

Reduce

As seen through the allreduce experiments, the main difference between the runtimes of maximum, minimum and sum operations are operation specific, i.e., do not change the overall performance observations made between the different implementations. Thus, `gaspi_coll_reduce` was tested with the sum as the reduction operation on Aenigma. Again, the routine was started 10^5 times and the average was calculated. Different from above, the average was calculated on all nodes and maximum of the averaged runtime results are used for the comparison of GASPI_COLL implementations MPI implementation shown in Fig. 5.9. This different view on the averaged runtimes is necessary because of the different work on the single nodes. While the leaf nodes only send off their data and are then immediately finished with their work in the reduce, the other nodes are dependent on the other participating nodes and have to wait on some data before being able to complete their own work.

There is no comparison to a native GPI2 implementation of a reduce, as the GASPI specification does not define a reduce and this none is implemented. In this case, the barrier between to call to the testes routine not only ensures that later runs profit from some caching effects of former runs but also ensures that the different routines do not have concurrent write accesses to the internal memory segment, possibly falsifying the results. This issue is related to the semantic suggested by the GASPI specification.

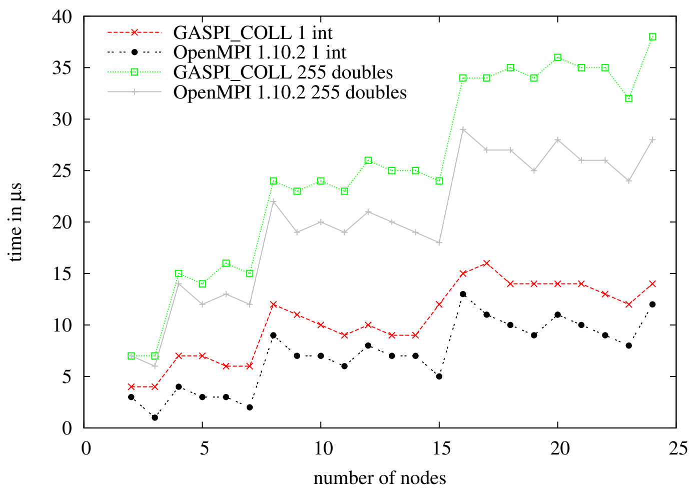


Figure 5.9.: Reduction with sum on Aenigma. Both with one integer and with 255 doubles.

For both small messages as well as for large messages, the MPI implementation of the reduce routine shows significantly faster averaged runtimes than the GASPI_COLL implementation. Especially for large messages, the runtime plots of the GASPI library routine and the MPI routine run almost in parallel, making jumps at 4, 8 and 16 nodes. With every one of the jumps, the gap between the MPI implementation and the GASPI library implementation increases, showing that the GASPI_COLL reduce implementation does not deal with increasing group sizes quite as well as the MPI implementation when reducing large arrays. For small messages, the gap between the MPI implementation and the GASPI_COLL implementation stays almost constant, with slight deviations at 3, 15, 16 and 19 nodes.

It was to be expected that the GASPI_COLL implementation does not show faster runtimes than the MPI implementation, due to the additional overhead induced by using the GASPI routines within the GASPI_COLL communication routines. The MPI implementation does not have to cope with this additional overhead but may instead use the network dependent communication API directly. The impact of this overhead could already be observed in the comparison of the adapted n-way dissemination algorithm implemented as a GASPI_COLL library routine or implemented in the scope of the GPI2 (Fig. 5.4, p. 87). Nonetheless it is very encouraging that the GASPI_COLL implementation runtimes stay within a difference of 15 μs of the MPI implementation, a range in which the GASPI induced overhead may very well fall.

The same expectations are in place for the GASPI_COLL broadcast implementation, with which the same experiments were conducted. These results are presented next.

Broadcast

The setting for the broadcast experiments were the same as for the reduce experiments, especially the timings shown are the maxima of all averaged runtimes on each node. Again, it was only compared to the MPI broadcast routine, as GASPI does not define a broadcast. Figure 5.10 shows the results of the broadcast experiments with small message sizes of one integer and large message sizes of 255 doubles.

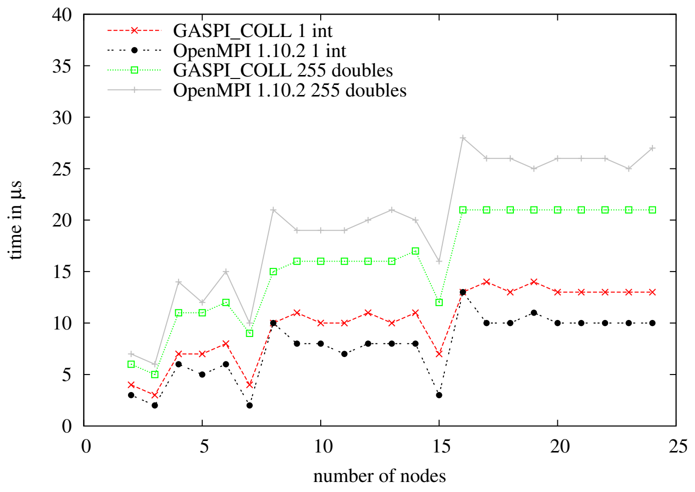


Figure 5.10.: Broadcast on Aenigma. Both with one integer and with 255 doubles.

Just like in the comparison of the reduce runtimes of MPI and GASPI_COLL, also the plots of the averaged runtimes of the broadcast run almost in parallel. Different from before, the runtimes of small messages here also show this behavior. The GASPI_COLL implementation is constantly slightly slower than the MPI implementation when using small messages. For large messages on the other hand, the GASPI_COLL implementation is faster than the MPI implementation. Just like in the reduce experiments, the tree-characteristic leaps in runtimes can be seen at 4, 8 and 16 nodes.

5.5. Discussion

This chapter introduced the collective communication library GASPI_COLL, complementing the GASPI specification with alternative algorithms for the allreduce operation and additional reduce and broadcast routines. The two main reasons for the implementation of this library are the exploration of different algorithms for the allreduce routine and the addition of collective routines that make the step of using GASPI for new applications and of porting existing applications to GASPI easier. With the design of the library and its implementation, obstacles concerning collective communication routines as GASPI applications have been encountered, which will be recapitulated and discussed here. Whenever the work in this chapter opened

questions or possibilities for future research, these will be described as well.
One of the goals in the design of the GASPI_COLL was to have a semantic and user interface that is consistent with the GASPI specification. Thus, certain design decisions were heavily influenced by the restraints and limitations imposed by the specification. Since the GASPI specification allows the creation of many groups, this was adopted in GASPI_COLL. This made the implementation of a distinct group structure and according group management routines necessary, given through `gaspi_coll_group_create` and `gaspi_coll_group_delete`. The first of these routines will prepare everything necessary for the later calls to GASPI_COLL communication routines, while the deletion routine will free the allocated resources again. For future implementations, it will be worthwhile investigating additional configuration possibilities for the group creation to enable a more resource saving, sparse group structure.
When creating a group, a segment will be allocated and registered for the one-sided GASPI communication routines used in the implementation of the GASPI_COLL collective routines. This is done to comply with the specification, which states:

> [[32], p. 102] `gaspi_allreduce` copies the send buffer into an internal buffer at the first invocation. The result is copied from an internal buffer into the receive buffer immediately before the procedure returns successfully.

Since the allreduce operation is the only collective communication defined by the specification, all restrictions defined for the allreduce will also be adopted in GASPI_COLL. The message size limit imposed by the GASPI implementation will be adopted by GASPI_COLL, directly leading to the necessary size of the internal communication segment for this group. One difficulty with this procedure is the number of segment IDs needed by the library. Each segment created by the library will need an unique ID which can not be used by the application any more. The number of creatable groups will thus depend on the number of creatable segments, being subject to the GASPI implementation at hand. Depending on future GASPI implementations, this behavior might have to be adapted. A different approach, not complying with the GASPI semantic, could be an implementation of collective routines working directly on memory segments allocated and registered by the user. This would lead to issues regarding the additional memory internally needed for, e.g., computing the partial results in the allreduce, receiving more than one message per round in most algorithms and double buffering. In addition to the access control this necessitates, additional care will have to be taken considering the notifications needed for weak synchronization within the collective routines. If the segment is shared by the user and the library, so are the notifications. It hence is more feasible to create library-owned segments, which the user should not access. A compromise could be to let the user define the segment ID to be used, which can easily be done through the addition of one argument in the group creation routine.
After Secs. 5.1 and 5.2 have introduced the group and memory management of GASPI_COLL, Sec. 5.3 described the routines implemented in GASPI_COLL. The allreduce operation, even

though defined and thus implemented in GASPI, is also implemented in the library to investigate the behavior of different algorithms in a one-sided, asynchronous setting. The adapted n-way dissemination algorithm, Bruck's algorithm, the PE algorithm and the BST algorithm were chosen for this. The first three were chosen because of their few communication rounds needed even for large groups and the BST algorithm was chosen as a representative of tree-based algorithms. Even though the BST algorithm showed the worst runtime results in the experiments, its implementation is necessary for all non-associative reduction operations. Although all algorithms were implemented with GASPI communication routines, adding a further level of overhead, the runtimes showed promising results on both clusters. Considering the overhead introduced through the usage of GASPI routines instead of directly implementing the routines with IB Verbs, especially Bruck's algorithm and the PE algorithm should be considered for implementing a native allreduce. A further point to note is the influence of different interconnects and the number of processes started per node on the runtimes of the collective communication routines. It was possible to test the allreduce implementations not only on Aenigma but also on MareNostrum III. The second cluster has a newer IB interconnect, with which the GASPI_COLL allreduce was faster than the local MPI implementation even for small messages. On the smaller cluster, this was only seen for large messages.

The reduce and broadcast operations could only be tested on the smaller cluster, Aenigma. In three experiments, the GASPI_COLL implementation was slower than the MPI implementation of the same routine, which was to be expected. Nonetheless, the difference in runtimes is so small, that they will most likely vanish or even reverse if implemented with IB Verbs instead of GASPI routines. In one experiment, the GASPI_COLL runtimes were even better than the MPI runtimes: the broadcast with large messages.

Apart from these promising runtime results of the reduce and the broadcast operation, the implementation of these two with respect to the GASPI paradigm and semantic introduces new fields of investigation, namely the successful return of the operation. The GASPI specification defined the return value `GASPI_SUCCESS` as that it "implies that the procedure has completed successfully." ([32], p. 14). For all communication routines, this return value only makes implications for the local status of the communication, i.e., there are no guarantees that all other involved processes have also completed the routine. This leads to the problem depicted in Fig. 5.1 on p. 79. Even though the routine is locally completed and another collective of the same type may be started, the internal communication might interfere with the previous collective on other processes. This problem is even increased in the implementation of one-to-all and all-to-one collective routines. Considering for example a sequence of broadcast calls with the same root, the asynchronism of the one-sided communication routines used could lead to a situation where the root has started n broadcasts, overwriting data in the receiving peer's memory. If the receiving peer has (in the worst case) not even entered the first broadcast, the data further disseminated by this process will not be the correct data. To prevent this, some kind of acknowledgment mechanism has to be implemented. All participating nodes will have to be notified that the broadcast or reduce routine has finished on all other nodes to safely start a succeeding collective of the same type. This does not comply with the semantic defined through the GASPI specification but is required for a more user-friendly interface, where the user is not constrained to make excessive use of the barrier operation only to make sure that a reduce or broadcast does not overwrite data of the preceding reduce or broadcast. The implementation of such an acknowledgment mechanism needs to be investigated in more detail in the future. One possible way to implement this kind of acknowledgment mechanism could be through the usage of one of the allreduce algorithms.

These obstacles do not only concern an implementation of collective routines in the form of a library, but also the design of according collectives within a GASPI implementation will need the same attention. Nonetheless, a native implementation of collective routines should always be preferred over an external library, which will always need additional resources. One issue, that would be immediately resolved through the native implementation is the access control of segments and notifications. While a native implementation of collective routines is to be favored, GASPI_COLL will complement GASPI implementations as long as further collective routines are not defined by the specification. Even though the inclusion of additional collective routines is repeatedly discussed in the GASPI Forum, an inclusion of further collectives into the GASPI specification is currently not planned, to keep the specification slim and the number of defined routines manageable.

The next chapter on the other hand, will describe a routine, that will be included in the next GASPI specification: the `gaspi_read_notify` routine.

6. Notified Read for GASPI

The GASPI specification has introduced several one-sided communication routines and weak synchronization possibilities for write operations, as presented in Sec. 3.5. The concept of weak synchronization can also be extended to the read routines, which has been done by the inclusion of a `gaspi_read_notify` routine [95] and the change of semantic of `gaspi_write_notify` as described on page 56 and in [94]. This chapter will go into more detail on the semantic and possible implementation of the `gaspi_read_notify` routine for IB networks. It will first present some background information on the motivation behind implementing and including `gaspi_read_notify` in the GASPI specification in Sec. 6.1, before Sec. 6.2 will describe the new routine in more detail, explaining why certain implementation and design choices have been made. Basic benchmarks and tests are described in Sec. 6.3. The two driving use-cases, a graph exploration and a pipelined matrix transpose, are described in Sec. 6.4 and Sec. 6.5. The graph exploration is inspired by the Cray Urika-GD (see p. 109), and is a proof of concept that graph exploration on distributed systems can easily be implemented with the `gaspi_read_notify` routine, while the graph traversal is a study on the feasibility to implement an alltoall or alltoallv with `gaspi_read_notify` in the scope of GASPI_COLL. Section 6.6 will recap the chapter and point out further research questions.

6.1. Introduction and Motivation

The GASPI specification has a variety of communication routines to offer (see Sec. 3.5), where a special focus lies on RDMA or one-sided communication. Two very basic one-sided communication routines are `gaspi_write` and `gaspi_read`. Since in RDMA communication only the initiating process is active, the GASPI specification also offers weak synchronization routines which enable the application programmer to notify the passive rank of written data. For this weak synchronization, each process has a notification array within a local segment. Notifications written to this array are guaranteed to not overtake previous writes from the initiating rank to the same destination rank and segment in the same queue. For example, if process p_0 issues multiple writes to rank p_1 and segment s_1 in queue q_1 and afterwards issues a notification to the destination tuple $< p_1, s_1 >$ in queue q_1, this notification will be written after all previous writes have been successfully completed. The remote rank, which has so long been totally passive, can issue a call to `gaspi_notify_waitsome` to check whether the needed data is already available.

Figure 3.5 on p. 56 already depicted this process, exemplarily showing, that the order of the issuing of the write requests does not necessarily influence the order of writing of the data. Yet, weak synchronization as described in Sec. 3.5 is not possible for the `gaspi_read` routine. A process can issue multiple read requests to the underlying network, but in contrast to the principle of writes and notifications, a following notification and the possibly successful return of `gaspi_notify_waitsome` on a given notification will not imply anything about the status of the previously issued read requests. Accordingly, no `gaspi_read_notify` routine is defined in the GASPI specification. But there are several use-cases, where this `gaspi_read_notify` could come in handy, e.g., in the pipelined matrix transpose kernel (Sec. 6.5) or in a graph traversal application (Sec. 6.4).
Following the above arguments, a proposal was submitted to the GASPI-Forum's meeting on January 20th 2016 to include `gaspi_read_notify` to the GASPI specification [95]. On June 22nd 2016 this proposal was accepted and will be included in the next version of the GASPI specification. This section will discuss the semantics and a possible ibverbs-based implementation of `gaspi_read_notify` in GPI2-1.1.1. The actual source code can be found in App. D.1.

6.2. Semantic and Implementation

The `gaspi_read_notify` routine interface has been designed to fit the rest of the GASPI API and especially close to the `gaspi_write_notify` routine:

```
gaspi_return_t
gaspi_read_notify(gaspi_segment_id_t segment_id_local
                 , gaspi_offset_t offset_local
                 , gaspi_rank_t rank
                 , gaspi_segment_id_t segment_id_remote
                 , gaspi_offset_t offset_remote
                 , gaspi_size_t size
                 , gaspi_notification_id_t notification_id
                 , gaspi_queue_id_t queue
                 , gaspi_timeout_t timeout )
```

Listing 6.1: `gaspi_read_notify` interface for GASPI.

Like the `gaspi_write_notify` routine, also the `gaspi_read_notify` routine needs information about the source location of the data, given through the remote rank, the remote segment ID and the remote offset of the data. The local destination location of the data, given through `<segment_id_local, offset_local>` is also necessary for a successful transfer of the data. The queue to post the communication request to is given by `queue` and the notification to be set through `notification_id`. A successfully returning call to `gaspi_notify_waitsome` on this notification on the local segment `segment_id_local`, indicates that the associated data will also have been written to the local segment.

In June 2016, a proposal was submitted to the GASPI Forum, requesting the change of the notification semantic of `gaspi_write_notify` [94]. With this change, the notification semantic of `gaspi_write_notify` and `gaspi_read_notify` will be the same in that the arrival of the notification only implies the arrival of the data in this notified communication call and makes no implications on any other write or read operations. In contrast to the `gaspi_write_notify` routine, the notified read does not take a notification value as an input parameter. Instead, the value is predefined as 1 for all notified reads. A timeout, as described on p. 52, may be given to use this non-local routine in a non-blocking manner.

The semantic of the `gaspi_read_notify` routine will be defined in the GASPI specification, but the implementation of the routine may differ significantly on different systems, depending on the underlying network and message transferal systems, i.e., the low-level communication APIs. The following implementation of `gaspi_read_notify` is specialized on IB networks and correspondingly makes use of ibverbs.

The `gaspi_read_notify` routine was implemented in a local copy of the GASPI reference implementation GPI2-1.1.1. The open source version of this GASPI implementation is implemented in ibverbs, as delivered in the OFED stack and described in Sec. 3.2.1. The general form of a ibverbs work request is presented on p. 36, but what is most important for the implementation of a `gaspi_read_notify` is the second entry in the `ibv_send_wr` struct: `ibv_send_wr *next` and the guaranteed ordering of messages as described on p. 37. This makes it possible to first read the requested data from the remote segment and afterwards read a notification value, written into the local notification buffer. This second read is necessary for the caller of the `gaspi_read_notify` to check the local notification buffer with `gaspi_notify_waitsome`.

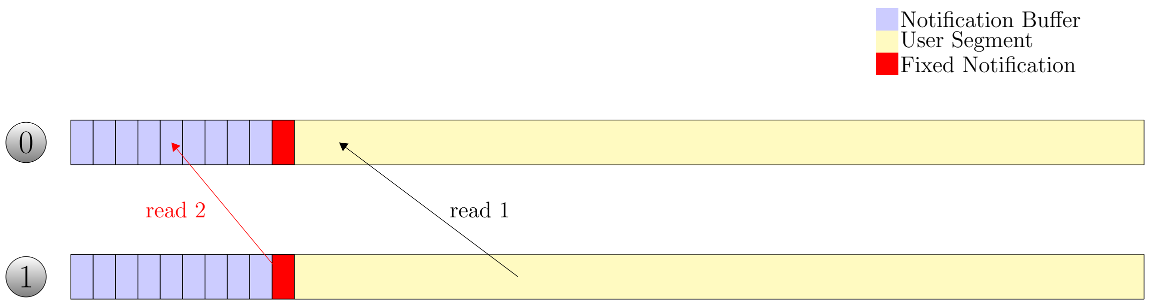


Figure 6.1.: A notified read initiated from rank 0, reading data from rank 1's global segment (read 1) and then reading the fixed notification (read 2).

To implement a `gaspi_read_notify` in GPI2-1.1.1, a fixed notification was included in each segment. Figure 6.1 shows a schematic sketch of a general GASPI segment. The portion of the segment available to the user is colored in yellow and the reserved and locked notification buffer is depicted in blue. In the discussed implementation of the `gaspi_read_notify`, an additional fixed notification is included at the end of the notification buffer. This fixed notification is depicted in red. The user does not have access to this memory location, as it is locked by the

implementation. The second read in the `gaspi_read_notify` implementation will read this fixed notification on the remote side and write it into the local notification buffer at offset `notification_ID`.
Before testing the routine in a matrix transpose application (Sec. 6.5) and a graph exploration application (Sec. 6.4), the following section will show basic performance tests and results.

6.3. Experimental Results

In a first benchmark, the native implemented notified read was compared to two emulations on top of the GPI2-1.1.1 implementation of the GASPI standard. The three tested versions are:

1. the native implemented `gaspi_read_notify`,
2. a GASPI application, which reads from the remote rank, waits on the queue and then notifies itself of the arrival of the data (read, wait, notify (RWN)), and
3. a GASPI application which first notifies the remote rank, and then the remote rank issues a `gaspi_write_notify` to the local rank (notify, write_notify (NWN)).

The benchmark measures the time until the local rank knows that remote data has arrived, which is tested through `gaspi_notify_waitsome`, i.e., a read based ping pong benchmark. The times were taken for different message sizes 2^i B with $i \in \{1, 2, \ldots, 19\}$, according to the micro-benchmarks distributed with the GPI2. Because not all message sizes are relevant to the research covered in the following sections, the figures in this section will only show the median runtimes for $i \in \{1, 2, \ldots, 12\}$. Supplementary figures can be found in App. D.2. Each version was run 1000 times and timed with the help of `gaspi_cycles`, a routine exclusive to GPI2 and used for all delivered micro-benchmarks to get the median of these timings.
Since many real-world applications are hybrid applications, i.e., using distributed and shared memory communication, it is especially important to know the performance of communication routines in a multithreaded environment. An important aspect of threaded applications is the sharing of resources, which can be influenced through the pinning of threads to cores. By pinning a thread to a given core, the OS or threading library may not move the thread to a different core during execution. This pinning will introduce additional management overhead, which might be balanced through the exclusive access of a thread to the core, i.e., reducing the contention on a resource. To evaluate this, different pinning models have been tested: unpinned, pinned to one core, pinned to the first <number of threads> cores or pinned to all even numbered cores. The tests were conducted on the Aenigma (p. 71) and on MareNostrum III (p. 86) and the results of the tests will also be discussed in this order. The threading was achieved through the usage of OpenMP pragmas, where the OpenMP version on Aenigma was 2.5 and on MareNostrum III it was 3.1.
In general, this section will concentrate on the discussion of the results of `gaspi_read_notify` and the unpinned version of the benchmark. The above mentioned, supplementary figures in

App. D.2 show the results for all message sizes and (emulated) notified read implementations. The results are grouped according to the different pinning models: Fig. D.1 shows the complete results of the benchmark with unpinned threads. Figure D.2 then shows the results of the benchmark with all threads pinned to core 0 and Fig. D.3 the results with the threads pinned to the first <number of threads> cores. These results are discussed together in this section, because for these pinning versions, the results and conclusions are very similar. When the threads are pinned to all even core numbers the results were significantly different and are shown in Fig. D.5. The runtime plots are very erratic for larger numbers of threads and show much higher runtime results than the other pinning models, disqualifying such a setting for HPC applications.

Figure 6.2 shows the median runtimes of the unpinned `gaspi_read_notify` benchmark on Aenigma for different numbers of threads. Figures D.2c and D.3c show the results of the `gaspi_read_notify` benchmark when pinned to core 0 and when pinned to the first <number of threads> cores. These show slight differences in single runtime measurements, but these do not influence the following statements.

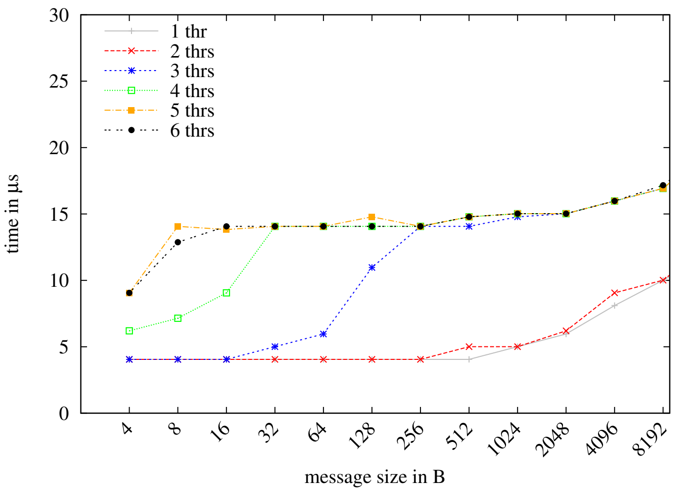


Figure 6.2.: Median runtimes of `gaspi_read_notify` ping pong benchmark with different message sizes and unpinned threads on Aenigma.

For message sizes of up to 8 kB, it does not make a significant difference, whether one or two threads are used. Very much alike, there is no significant difference in runtime at all between the usage of five or six threads. For message sizes up to 16 B one to three threads show almost the same runtimes, but from 32 B on, the contention caused by a third thread can be seen in the runtimes. The median runtime with three threads converges towards that of four to six threads and levels with these at a message size of 256 B. When using four threads, the development of the runtimes is similar. The runtimes are already higher for small messaged than when using fewer threads, and they converge towards the runtimes with more threads, already leveling at 32 B messages. The main reason for the runtime increase between the different number of threads used is contention on the hardware resources and management overheads. If the difference in runtime for reading a 8 B message with two or four threads was solely due to the increase of the overall transferred data size, it could be expected, that one thread needs approximately the same time for reading 32 B as four threads need to read 8 B. This is obviously not the case, but rather the extra time for handling competing accesses of the threads to some resource are reasons for the increase in runtime.

Figure 6.3 shows the median runtimes of the `gaspi_read_notify` implementation for different thread pinning options. The pinning of all threads on one core is labeled as *pinned 0*, the pinning of the threads to single consecutive cores is labeled *pinned 0-5* and the unpinned version is labeled as *unpinned* in the key. For better readability, only the runtimes of one thread and six threads are shown in this figure. There are no meaningful differences in the runtimes between the different pinning models for messages up to 8 kB. Only for very small messages of up to 8 B, the unpinned run shows slightly faster runtimes than the other two versions when using 6 threads. Again for 64 B and 512 B messages there are slight runtime differences when using 6 threads, but these are only differences of approximately 1 μs and hence neglectable in the overall interpretation of the results. This underlines the previously made statement, that the pinning model does not influence the runtimes significantly.

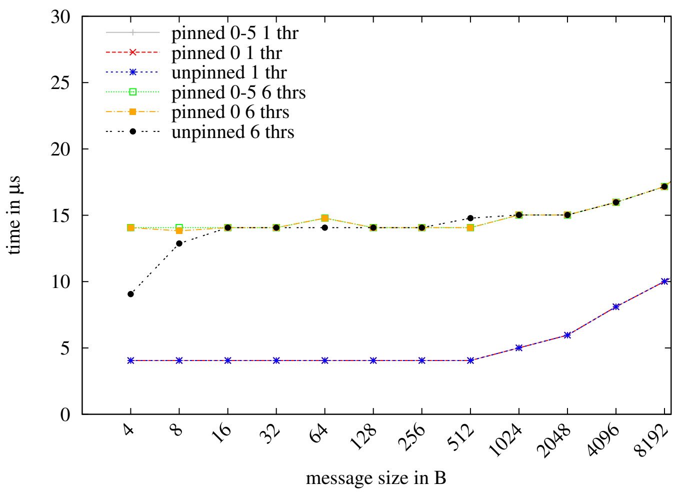


Figure 6.3.: `gaspi_read_notify` with different pinning on Aenigma.

Through the comparison of Figs. D.1a to D.3a, respectively Figs. D.1b to D.3b, these statements can also be found true for NWN and RWN implementations.

Figure 6.4 shows a direct comparison of the runtimes of the different (emulated) notified read implementations in the unpinned case. The native implementation of `gaspi_read_notify` actually shows better runtimes with 6 threads than the two emulated versions. This difference is significant for message sizes of up to 256 B, but dissolves afterwards. Fig. D.4 shows this comparison for all pinning models. The NWN implementation has better runtimes for very small message sizes when a pinning model is applied. These are only significant for message sizes of 4 B and afterwards neglectable.

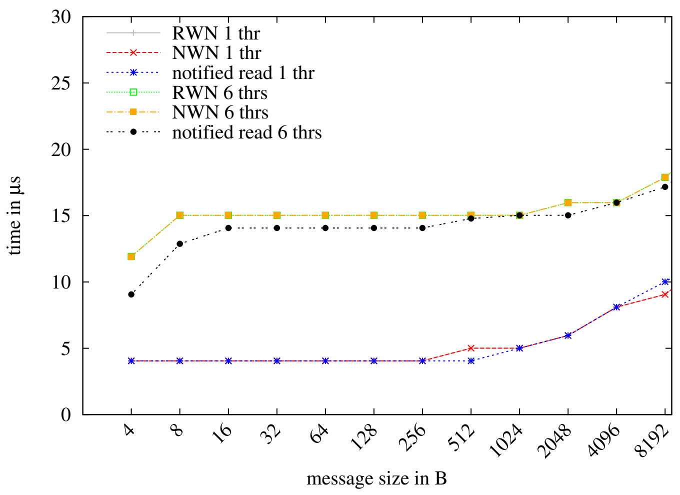


Figure 6.4.: Comparison of the different read implementations without pinning on Aenigma. For higher readability, only results for one and six threads are shown.

An important observation to be made in all cases is that the increase of the message size by 2 does not increase the median runtime of the fetching of the data and subsequent notification by the same factor for messages of up to 8 kB. As can be seen in the figures in App. D.2, this behavior gravely changes for messages of 32768 B and more, where a doubling of the message size also leads to a high increase of the runtime (note the logarithmic y-axis in the plots).

The same experiments were conducted on MareNostrum III, a system with more cores per socket and a different interconnect (see p. 86). While Aenigma has 6 cores per socket, MareNostrum III has 8 cores per socket and accordingly the number of threads used for the test was raised. While the interconnect family and topology of the two systems is the same, i.e., IB networks in a fat tree configuration, the IB type of MareNostrum III is FDR-10 in comparison to the QDR type in Aenigma. This different type of IB network has a higher theoretical message rate than QDR networks. This, together with the higher number of cores per socket, was expected to lead to somewhat different results on MareNostrum III. Due to the very erratic results of the benchmarks with threads pinned to every second core (see Fig. D.5), this specific benchmark was left out on MareNostrum III. As before, additional figures and the data sets regarding the benchmarks on MareNostrum III can be found in App. D.2.

Figure 6.5 shows the results of the same experiment on MareNostrum III that Fig. 6.2 showed for Aenigma: the comparison of the median runtimes for the process to read remote data of different sizes and notify itself of the arrival of the data for different numbers of unpinned threads. Similar to the results from Aenigma, there is almost no difference in runtime to be observed for three threads or less. On the other hand, there is not such a clear upper limit to the runtimes as it had been posed by the runtimes of five or six threads on Aenigma. Using four or five threads always adds a bit to the runtime that was needed for the same message size by three, respectively four threads. The same behavior is seen when using six threads, but surprisingly, the benchmark delivers approximately the same runtimes up to a message size of 256 B when using eight threads. When using seven threads, the runtime is higher than using eight threads for messages up to a size of 1024 B.

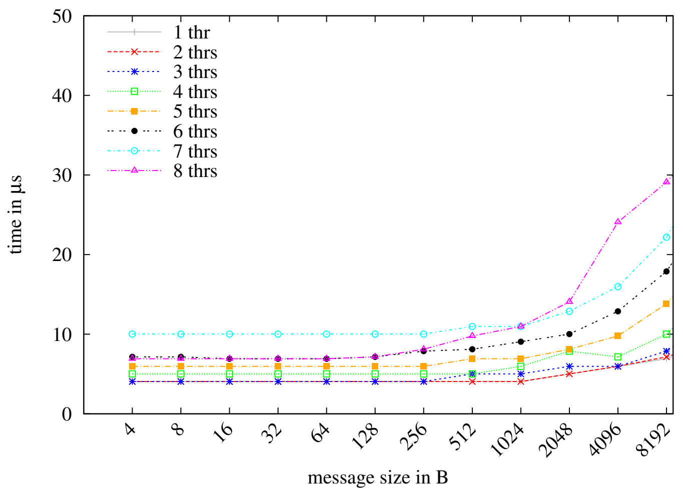


Figure 6.5.: Median runtimes of `gaspi_read_notify` ping pong benchmark with different message sizes and unpinned threads on MareNostrum III.

In comparison to the runtimes on Aenigma, a marginal decrease is visible for low thread numbers and small message sizes. With higher thread numbers the lower runtimes in comparison to the same thread numbers and message sizes on Aenigma become more evident. This observation is actually turned over when looking at the runtimes for larger message sizes of 8 kB and more. While all depicted runtimes on Aenigma stay below 20 μs even for 8 kB messages, the median runtime for this message size with seven threads surpasses the 20 μs boundary and is even close to 30 μs when using eight threads. This is clear evidence of the dependence of communication routines on underlying hardware.

Figure 6.6 shows the results of the `gaspi_read_notify` benchmark and the influence of the different pinning schemes on the runtime of on MareNostrum III. While there was merely a marginal difference between the runtimes in the different pinning schemes on Aenigma, this is significantly different on MareNostrum III for higher thread numbers. Without a predefined pinning scheme or when pinning all threads to only one core, the runtimes for messages up to a size of 2048 B are relevantly faster than when pinning the threads to eight consecutive cores. For messages of 4 kB or 8 kB in size, this is turned around again. A comparison of Figs. D.6 to D.8 shows the according results for the emulated reads, where RWN shows a similar but not as significant behavior and NWN shows very different runtimes for messages up to 8 kB depending on the pinning model.

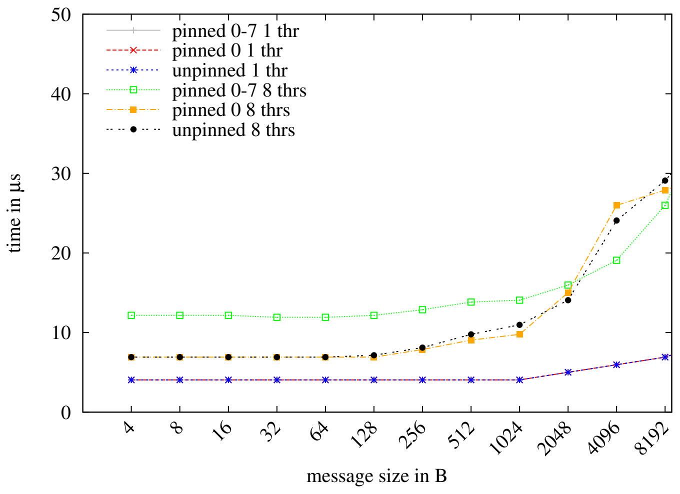
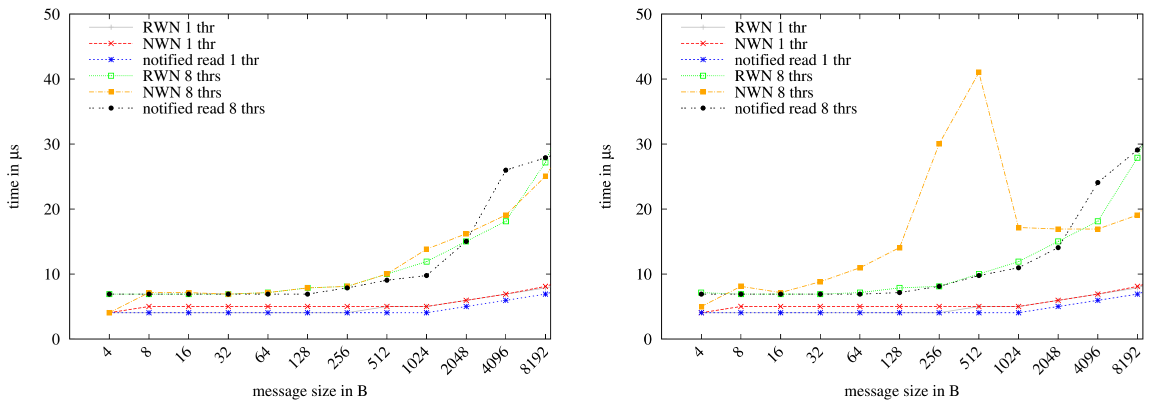


Figure 6.6.: `gaspi_read_notify` with different pinning on MareNostrum III.

Figure 6.7 shows a comparison of the different implementations with the three pinning models. While there is not a large difference between the runtimes of the implementations when all threads are pinned to one core (Fig. 6.7a), the results for the other pinning schemes are more diverse. Figure D.9c shows the median runtimes of the three implementations when the threads are pinned to eight consecutive cores, i.e., on one socket. The RWN implementation with eight threads has runtimes close to the runtimes when only one thread is active for message sizes up to 1024 B, while the other two implementations are much slower. The NWN implementation and the `gaspi_read_notify` implementation's runtimes are relatively close together, differing almost constantly by only approximately 2.5 μs. All three runtimes converge for large messages from 2048 B on.



(a) pinned to one core **(b)** unpinned

Figure 6.7.: Comparison of the different read implementations with different pinning on MareNostrum III. For higher readability, only results for one and eight threads are shown.

Last but not least, Fig. 6.7b shows the runtimes with no pinning restrictions applied. The two read-based implementation show a very steady increase in the median runtimes, but the NWN implementation has an erratic increase for 256 B and 512 B messages, before decreasing to a more expected level again. This behavior could not be further investigated during the time of granted access to this machine, but considering that the same benchmark with fewer threads does not show the same behavior (see Fig. D.6a), there is a strong suspicion that there has been high congestion in the network.

Even though the runtimes of the `gaspi_read_notify` experiments range in the same runtime area as the two emulated GASPI implementations, the native implementation has some important advantages over the other two implementations. While the NWN implementation needs both involved processes to be active in the communication, the native implemented notified read only needs one active component. In addition, the process waiting on the data will not be affected through the progress of the remote rank. If the remote rank is stuck in some computation, it will not be able to check on its notifications and issue a write. Thus the local rank might be idling until the remote rank has spare resources to transfer the data.

The RWN emulation does not suffer from this drawback. Considering that the process always needs to wait on a queue until it can issue the next read, the amount of reads in flight is limited to the number of available queues. In a real world application, this difference will have an immense effect on the scalability of the application.

To further investigate on the usability of `gaspi_read_notify` in real-world applications, the notified read was tested in two use-cases closer to real-world scenarios: a graph exploration and a matrix transpose. The graph exploration, described in the following section, can be seen as an extended ping pong benchmark with multiple communicating nodes. Only through a native implementation of the notified read, this kind of distributed graph exploration is made possible. The matrix transpose benchmark will then be described in Sec. 6.5 and is not only relevant for the implementation of an actual distributed matrix transpose, but also delivers valuable insights on possible implementations of a PGAS alltoall.

6.4. Use-Case: Graph Exploration

In the age of *Big Data*, new approaches are needed in every area working with large amounts of data to cope with the immense rise of data. In many fields, the data is collected and then analyzed with respect to previously unknown connections between the single data items. Example use-cases for such graph analytic methods are news [16], health care [15] or earth sciences [20]. For example, finding connections between the symptoms of different patients can lead to new insights on new diseases and their therapy or the analysis of similarities in genome sequences may lead to a better understanding of hereditary illnesses. Similarly, the analysis of journalistic data can be sped up by graph analytic applications to reveal connections between

seemingly unrelated facts belonging to a bigger story. Researchers can also be supported to find related work in the obscure amount of available resources, transcending linguistic barriers as well as semantic changes over the course of time.
The data, distributed among the memory in a cluster, can be described as the nodes of a graph. The connection one data item u has to a second data item v can then be considered as an edge of a graph. In this manner, data analysis is directly translated into graph analysis and, because the connections are unknown in advance, into a graph discovery or exploration application. In the following, connected nodes will be called neighbors.
Not only the is the amount of data to be analyzed today so large that a human will not be able to analyze it thoroughly in one's lifetime, but in addition, it will not fit into a single computer's memory nor does the computing power of a single computer suffice to analyze the data in an acceptable time frame. This timeframe varies from use-case to use-case but in the fast moving times today, an acceptable time frame becomes smaller every minute. While the necessary compute power may be accessible in clusters or in the cloud, the problem of data locality remains. Commodity, shared memory architectures are not large enough to cope with Petabytes of data and thus, the data will most likely be distributed among different compute nodes. Cray has chosen to address this problem with a single graph discovery appliance: The Urika-GD Graph Analytics Appliance [17]. In this scope, Cray names challenges of big data graph analysis:

- Partitioning of the graph:
  To cope with large amounts of data, the Urika-GD is equipped with a large shared memory of up to 512 TB.
- Unknown connections between the vertices:
  The large shared memory already deals with one of the issues arising through not knowing the connections between the vertices: it yields prefetching and predictions unnecessary. In addition, it might be necessary to analyze different edges of the graph at the same time. This results in a lot of data loading into random access memory (RAM) and accesses to main memory. Relative to the processors computing speed, these accesses are extremely slow. To keep the processor from idling, the Urika-GD is equipped with up to 8192 graph accelerator processors (Threadstorm), each capable of dealing with 128 threads.
- Dynamic of today's data:
  Since the data set may grow during run-time and needs to be included in the graph database, the Urika-GD has a fast I/O connection of up to 350 TB/hour.

The Urika-GD is purpose-built and tackles the problems of immense amounts of data with a equivalently immense shared memory, which is at the same time their solution for mitigating communication overhead. Yet, not every researcher is able to work with a Urika-GD but must instead cope with a distributed memory system at hand. The above addressed communication overhead then has to be taken into account. The goal needs to be, that the computational power of all available nodes is bundled to find the connections between different, distributed data items while at the same time retrieving remote data without involvement of the remote node. The last aspect is especially important to keep all nodes autonomous in their computation and independent of the status of other nodes. Because RDMA operations allow this kind of communication without involvement of remote CPUs, a PGAS-based implementation of a graph discovery algorithm seems predestined.

The following considerations are very theoretical, because no genuine data analysis problem was available, but they strongly underline the possibilities offered by a notified read routine: with the new `gaspi_read_notify` routine, it is possible to implement a distributed graph traversal in the PGAS, which is arbitrarily expandable and thus capable of holding even the largest of data sets. While classical graph traversal algorithms, like breadth first search or depth first search, require a very sequential approach to traversing the nodes, a GASPI-based graph traversal approach is massively parallel and asynchronous. On each compute node a graph exploration is started by multiple threads as described in List. 6.2.

```
get_root_node(&this_node);
for(i=0; i < this_node.num_neighbors; i++){
  notified_read(this_node.neighbor[i];
}
while(!complete){
  wait_for_node(&this_node);
  for(i=0; i < this_node.num_neighbors; i++){
    notified_read(this_node.neighbor[i]);
  }
  do_work(&this_node);
}
```

Listing 6.2: Schematic pseudo code of a notified read based graph traversal.

After having read a local root node v_0 to begin the exploration at, the neighbors $v_1, \ldots, v_n$, i.e., the nodes this root node is connected to, are known. These neighbors are then visited, i.e., the information is read from (remote) memory, to reveal the connections of these nodes. With `gaspi_read_notify`, the reads from remote memory can be offloaded to the network, freeing CPU resources for further necessary analysis of the data connected to the previously read nodes. While this analysis reveals the connections of the nodes in a real-world application, the nodes in the kernel application implemented with `gaspi_read_notify` carry the addresses of the neighboring nodes. The nodes in the mock-up model are thus designed as shown in List. 6.3. In addition to the addresses of the neighbors, the struct also carries the number of neighbors, necessary to know how many read requests are to be issued, and a dummy payload.

```
typedef struct{
  gaspi_rank_t rank:
  gaspi_segment_id_t seg_id;
  gaspi_offset_t offset;
} node_address_t;

typedef struct{
  int num_neighbors;
  node_address_t neighbors[MAX_NUM_NEIGHBORS];
  double payload[SIZE];
} node_t;
```

Listing 6.3: Structures necessary for GASPI graph traversal.

Some GASPI-specific preparatory steps are needed to set up the environment. For one-sided communication, a PGAS has to be set up and segmented. For first experiments, one source segment s_i per compute node was allocated. This source segment is filled with random graph nodes. In addition to the source segment, target segments are needed, where the (remotely) read information is written to. In a first setup, one segment was created per thread to decrease the number of potentially shared variables (e.g., read counters or offset counters) and at the same time increase the number of available notifications. Each notified read will use up one notification, thus the goal is the maximization of available notifications. The possibility to add notification buffers through additional segments facilitates the scalability of a GASPI-based graph exploration because it increases the number of possible read requests in-flight. The only limiting factor will then be the number of available queues and their capacity.
In a first step towards any real-world scenario, a graph exploration with `gaspi_read_notify` was implemented as depicted in List. 6.2, without a work part. Depending on the cluster and the number of compute cores per node, a fitting number of threads was started. Fitting here means, that at most one thread per core is started, i.e., no potential hyper-threading was used. For threading the application, OpenMP was used in the version available on Aenigma and MareNostrum III (versions 2.5 and 3.1 respectively).

6.4.1. Experimental Results

One GASPI process was started per node for this experiment. Each process then again started 6, respectively 8 threads with an OpenMP pragma. The number of threads were chosen with respect to the results of the tests in Sec. 6.3, which implied that starting more threads will have drastic effects on the runtime. No pinning model was applied when starting the application, also as a direct consequence from the previous tests. Each thread had its own segment and queue to work with. This worked well for this experiment, because the number of started threads did not exceed the number of available queues or segments in this GASPI implementation and should also be considered for future genuine implementations to reduce internal management overhead

and support scalability. Additional management will be necessary if this is not possible in future settings.
The impact of possible payload was investigated by using 180 doubles as additional payload. The total amount of data transferred was hence 160 B without additional payload and 1600 B with payload. The number of neighbors per node was fixed to 10 to be able to reproduce the test results on other systems. As every thread had its own segment to read the data to, each thread also had 65535 notifications at hand. After having received 65535 notifications, the program was stopped and timings were taken with `gettimeofday` out of the sys/time.h header. Both seconds and microseconds were taken from the value returned by `gettimeofday`. On Aenigma, the GPI2-1.1.1 implementation including the `gaspi_read_notify` implementation was used and cross compiled with OpenMPI 1.10.2 and multithreaded support. In Fig. 6.8 the average time it takes for a `gaspi_read_notify` to be posted and the according `gaspi_notify_waitsome` to see the notification is shown, with a comparison between the graph traversal with no payload, i.e., only the neighbors are read, and the graph traversal with payload, i.e., additional 180 doubles are transferred.

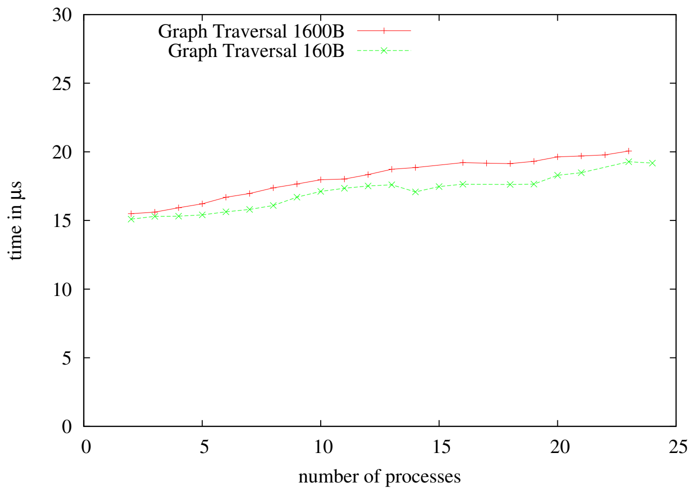


Figure 6.8.: Graph traversal results on Aenigma with (red) and without (green) payload, 65535 reads per thread and 6 threads per rank.

When comparing this to Fig. 6.2, where the results of a `gaspi_read_notify`-based ping pong benchmark were shown, the impact of multiple communicating nodes can be derived from this figure. The ping pong benchmark was conducted between two nodes, transferring messages of increasing sizes. At 128 B and 256 B this benchmark needed 14.066696 μs and at 1024 B and 2048 B, the benchmark needed 15.020370 μs. These values are merely increased to 15.09824 μs and 15.49730 μs when testing on two nodes in this graph traversal kernel, i.e., the additional work of retrieving the addresses of the neighbors merely increased the runtime by less than 1 μs. The increase of runtime through the addition of multiple compute nodes is also of high interest. Even though the load on the network is 12 times higher when using 24 nodes than using 2 nodes, the runtime of the graph exploration kernel only increases by approximately 4.1 μs, respectively 4.6 μs, i.e., an increase by a factor of 1.27, respectively 1.29.

On MareNostrum III, the same GPI2-1.1.1 version was used and cross compiled with IntelMPI 4.1.3.049 and multithreaded support. The most interesting parts of the graph traversal kernel were the behavior with larger numbers of nodes and accordingly the benchmark was only run with certain higher node numbers. Figure 6.9 shows these averaged runtime results of the graph traversal kernel on MareNostrum III. Again, the difference in runtimes between the version with payload and the version without payload stays almost constant up to 64 nodes, but with 128 nodes, the gap increases significantly - but still not linearly with the rise of data transferals.

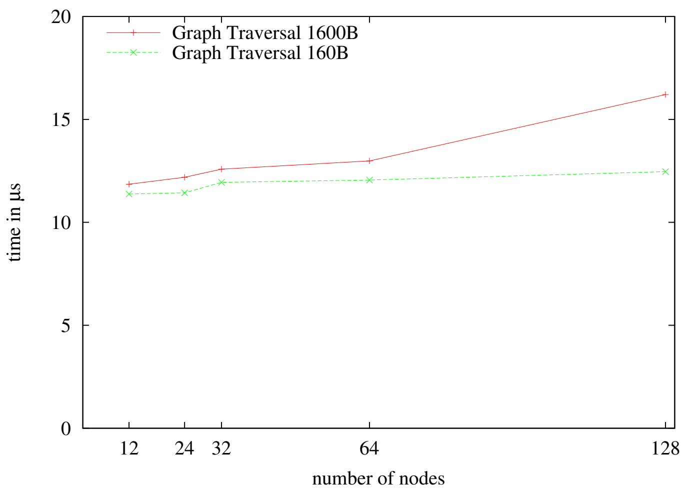


Figure 6.9.: Graph traversal results on MareNostrum III with (red) and without (green) payload, 65535 reads per thread and 8 threads per rank.

The results on both Aenigma and MareNostrum III show, that the `gaspi_read_notify` routine can be used as a basis for graph traversal use case scenarios. Despite increasing network load and message sizes, the runtimes of the graph exploration kernel do not show devastating results. Nonetheless, it is clear that the experiments shown in this section can only be seen as the very basis of future research on graph exploration with notified read. A genuine use-case needs to be investigated, such that the overlap of computation and communication can be explored in a real-world scenario. Additionally, the scalability will have to be tested on larger systems, which may show the need for additional scheduling mechanisms within the application to deal with high numbers of nodes.

Another possible use-case for the notified read are the alltoall and alltoallv routines, which could be implemented in a partially evaluable fashion with the producer-consumer model enabled by the notified read routine. To probe the usability or `gaspi_read_notify` in an alltoallv, the PGAS community benchmark - implementing a matrix transpose - was used as a baseline.

6.5. Use-Case: Pipelined Matrix Transpose

The transpose of a large matrix is part of many high-performance applications or libraries, e.g., the fast Fourier transform or BLAS. The challenge for an efficient program today is the size of the matrices, because one matrix will no longer fit into the memory of only one process. Thus, the matrix is distributed among the processes. A matrix transpose has the same communication scheme like an alltoall routine (see p. 22) and is thus investigated with regard to a possible implementation of a GASPI_COLL alltoall routine.

The pipelined transpose kernel in this section performs a multithreaded, distributed matrix transpose in blocks, as shown in Fig. 6.10. Each of the processes 0, 1 and 2 have columns of a large matrix in their memory. These are represented as sub-matrices A_i, B_i and C_i. In the communication step, every process transfers 2 sub-matrices to the other processes. The transpose of the sub-matrices is then done locally. For small matrices, both steps might be combined in one single communication step, but for large matrices, this is not possible due to the constraints of given communication routines.

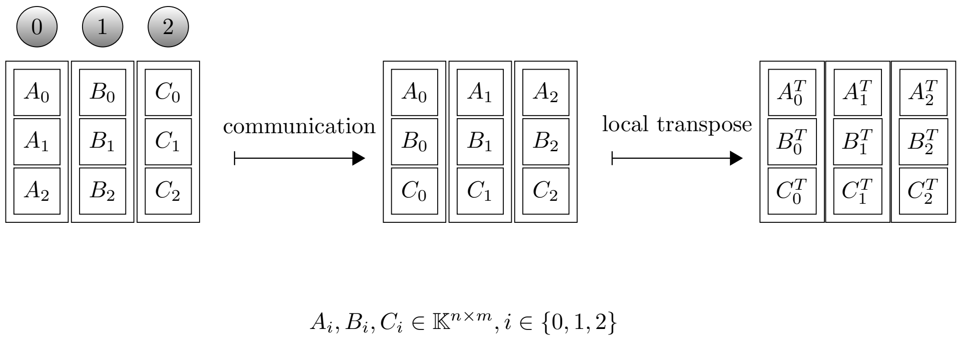


Figure 6.10.: A distributed matrix transpose with three processes. The transpose is divided into a communication step and a local transpose step.

In this benchmark, the local transpose step is always conducted in a threaded manner with OpenMP for shared memory communication, whereas the interesting part for this benchmark is the distributed communication. There are several ways to tackle the communication step of the matrix transpose. The most straightforward implementation might be the usage of an alltoall function (see p. 22), like it is implemented in MPI. This scheme can also be found in the PGAS community benchmark, available at GitHub [92]. Seeking an asynchronous, highly parallel, one-sided implementation attempt, there are several issues with this implementation. For example, only one thread issues the allreduce operation and in addition to that, a barrier is necessary for synchronizing the threads. Only the thread issuing the blocking alltoall routine will know when the data from the other processes has arrived and needs to inform the other threads. Another issue is the missing overlap of computation and communication because all communication needs to be finished before the transpose of the sub-matrices can be started.

Another implementation at [92] is a GASPI based implementation with `gaspi_write_notify`, schematically shown in List. 6.4. In this implementation, all write requests are started by one thread in one loop. This results in a simultaneous initiation of $P \cdot (P-1)$ write requests - this number becoming arbitrarily large with rising process numbers and possibly congesting the network right from the beginning of the execution. After having issued the write requests, each thread runs through a loop of block-indices checking whether the required block has been received or not. If the block has been received or is a local block, the sub-matrix is transposed. In case the data is not available yet, the thread will come back to this block in a later loop iteration, until all blocks have been transposed.

```
#pragma omp parallel{
  if(this_is_first_thread()){
    for(i = 0; i < P-1; i++){
      gaspi_write_notify(&submatrix[i]);
    }
  }
  while(!complete){
   for(j = 0; j < my_num_submatrices; j++){
      if(gaspi_notify_waitsome(&j) == GASPI_SUCCESS){
        transpose(&submatrix[j]);
      }
    }
  }
}
```

Listing 6.4: Schematic implementation of the pipelined matrix transpose with `gaspi_write_notify`.

Even though the communication is overlapped with the transpose a bit better in the write-based implementation than in the alltoall version, all communication is still done in one part of the code. All data is spilled into the network at once, by one thread. In addition to this, the work load is partitioned statically. Thus, the first thread, doing all the communication in the loop, will start the transposing with some delay in relation to the other threads. Because this thread has the same number of submatrices to be transposed like the other threads, these other threads will most likely be done and idling until the first thread has finished its transposes.

To overcome these issues, a pipelined transpose with an emulated notified read has been implemented [92]. The emulated notified read is implemented as a GASPI application, like the RWN emulated read in Sec. 6.3. Each read is issued in a new queue and before issuing the next read to the same queue, the application waits on the queue. When the wait request returns successfully, the read data has arrived in the local memory and the process issues a local notification. Afterwards, a new read is issued to the queue. The local notification is necessary for the other threads to be able to use the `gaspi_notify_waitsome` routine to check for arrived data. Still, all reads are issued by only one thread in an initial loop but the submatrices are transposed as they arrive and the threads do not necessarily have to idle until the reading thread has transposed all submatrices.

These different implementations were benchmarked on a 7D enhanced FDR IB hypercube and the results can be found on [93]. Additionally, an MPI implementation with non-blocking sends was benchmarked in this setting. On this system, 512 processes with a total of 6144 cores were available for benchmarking. The MPI implementations stop scaling very early, while the two GASPI implementations keep scaling almost identically well up to 256 processes. After this, the emulated read shows better scaling than the implementation with the notified write.

Since this benchmark already presents results for one of the most common hybrid implementation techniques, as well as a GASPI implementation to compare against, this benchmark was chosen to benchmark the native implementation of `gaspi_read_notify`.

6.5.1. Implementation with Notified Read

One of the theoretical main benefits of a pipelined transpose based on notified reads is the idea of triggering a read request, whenever another one has been processed. This should lead to less network congestion when dealing with high numbers of processes. In an implementation where all writes or reads are issued in one loop, an application with P processes will issue $P \cdot (P-1)$ communication requests at once. In this implementation, a fixed number of reads is issued in the beginning and further reads are issued whenever one of the previously read data elements can be processed, i.e., the communication is consumer-driven.

The above scheme was implemented in the scope of the mentioned PGAS community benchmark available at [92]. The kernel is a hybrid implementation of GASPI, MPI and OpenMP, where MPI is used for the initialization of the processes and the MPI barriers are used when taking the timings. One-sided, notified GASPI routines are used for the communication, i.e., `gaspi_write_notify` and `gaspi_read_notify`. OpenMP is used for using multiple threads in the execution and for parallelizing not only the communication with other nodes, but also the work. Accordingly, the `gaspi_read_notify`-based implementation uses MPI, GASPI and OpenMP routines in the same manner and is schematically shown in List. D.2.

In a multithreaded environment, mechanisms are needed to ensure that each matrix block is only read once and to keep all threads busy with work until the whole matrix is transposed. Additionally, in this dynamic reads setting, it is not possible to statically divide the work among the threads before starting the application. Therefore two counters, shared among the threads of one MPI process, are used to guarantee that each block is read and transposed only once, while at the same time enabling the dynamic distribution of work: a read counter and a notification counter.

The read counter is used and incremented by the threads every time they issue a read request. The value retrieved from the read counter is then used to determine from which rank messages shall be read. The atomic incrementation and retrieval of the read counter is ensured through the OpenMP flush pragma and the usage of `__sync_fetch_and_add`. As soon as this counter

reaches a value $> P - 1$, a thread-local switch variable is set and no further accesses to this counter are necessary. If this is already the case in the initial reading loop, the loop is exited. The notification counter tracks whether further calls to `gaspi_notify_waitsome` are necessary. As long as there is still some data in the pipeline - either in the communication channel or already locally available but unprocessed, each thread will call `gaspi_notify_waitsome` on the first P notifications. If a notification is received, all threads will get the ID of the received notification and subsequently make a call to `gaspi_notify_reset` on the received notification ID. Only one of the threads will receive the value of the notification, increment the notification counter, if necessary start a new read request and then start the transposition of the received matrix elements. The other threads will again call `gaspi_notify_waitsome`. The first thread to reach the code for the transposition of the local submatrix will transpose the local portion of the matrix and afterwards join the other threads in locally transposing the remote submatrices.
In comparison to the given implementations, this dynamic notified read implementation has several theoretical benefits. First of all, the local transpose is not dependent of any other processes any more. While in the write- and alltoall-based implementations, each process was dependent of the progress of the other processes, this issue is resolved in the read-based version because only the transposing process itself is active in the communication. Furthermore, the work is distributed dynamically. Not only can each thread work on any block that has arrived, but even more, reads are only issued when a thread will have the capacity to transpose a new block in near future. This should result in a less contented network, because the reads are issued over some time span and not all at once in the beginning of the transpose.
In addition to the here presented implementation with a dynamic work distribution and communication scheme, a second `gaspi_read_notify`-based transpose was implemented with the same scheme that the original implementations of the benchmark have. This implementation will be called the static read implementation in the following. The results of the conducted experiments as well as their setup will be described in the next section.

6.5.2. Experimental Results

The notified read based implementation was compared to the different implementations already available for the PGAS community benchmark at GitHub [92]. The runtime was determined through two time measurements: one directly before entering the barrier in front of the parallel region and one right behind the barrier after the parallel region. The median iteration time of 50 iterations was taken to calculate the transpose rate (in GB/s). The experiments were conducted on Aenigma and MareNostrum III. No change was made to the timing, median determination and rate calculation given in the community benchmark. Nonetheless, several parameters potentially influence the pipelined transpose kernel benchmark:

(i) The number of MPI processes started on each node (ppn).
    Even though the benchmark measures the runtimes of GASPI routines, the runtimes

are compared to a pure MPI implementation and to the benchmarks of the GASPI community benchmark. Both are set up with MPI process management routines and to maintain comparability, also the `gaspi_read_notify` implementation will be started with MPI processes. The experiments will be started with either one or two processes per node to reduce contention on the available resources. The setup of the community benchmark necessitates `M_SZ` to be dividable by the number of processes used, influencing the possibly tested node counts immensely.

(ii) The number of threads used.
The number of threads used per compute node can show significant impact on runtimes. The usage of multiple threads introduces management overhead and sharing of resources. Depending on the architecture of the compute node, the threads need to share resources like compute nodes and of course the NIC.

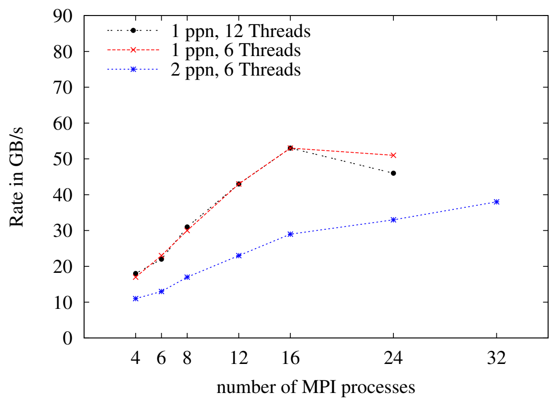
(iii) The size of the matrix to be transposed.
The size of the matrix to be transposed influences the number and size of messages to be transferred over the network. In the GASPI community benchmarks, the original size of the matrix was `M_SZ` = 12288. In addition, multiples of this size are tested to sustain the side conditions of the benchmark.

In the following, the impact of the different parameters on the transpose rate is shown through different experiments. First, experimental results on Aenigma are presented.In dependence of these results, the parameters for the tests on MareNostrum III were chosen. The results on MareNostrum III are explained at the end of the section.
The plots will be showing different matrix transpose implementations in the scope of the GASPI community benchmark. To create a baseline for the benchmarks, all online available implementations of the benchmark have been tested on the different machines. In addition, two implementations with the `gaspi_read_notify` routine have been tested, which totals in five different implementations:

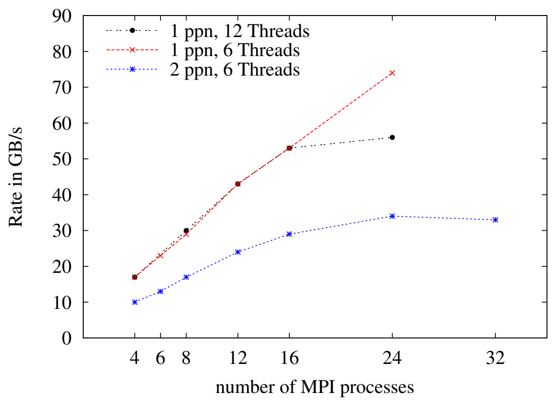
1. The MPI-based alltoall implementation of the benchmark.
2. The `gaspi_write_notify`-based implementation.
3. The emulated read implementation, in which the RWN implementation of the notified emulated read is used (see p. 102 item 2 for details on the emulated read implementation).
4. An own implementation with `gaspi_read_notify` and the static distribution of the original benchmark.
5. An own implementation with `gaspi_read_notify` and a dynamic distribution of work and communication, as described above.

The first parameter tested was the number of processes to start per node in combination with the number of processes started. As described in bullets (i) and (ii) above, these numbers influence the contention on the single resources of each compute node. Since the compute nodes are equipped with two sockets, a maximum of two MPI processes were started per node. In addition, to limit sharing of resources by threads, at most one thread per core was started. Overall, the following setups were benchmarked: one process per node (1ppn) with one thread per core, two processes per node (2ppn) with one thread per core and one process per node with half as many threads as cores started.

Figure 6.11 shows the impact on the transpose rate, induced through the change of number of nodes and threads started. Both figures show the results for a matrix size of 12288 (the initial benchmark value). In Fig. 6.11a the transpose rates of the static `gaspi_read_notify`-based implementation are plotted. The blue line shows the rate for 2 MPI processes started on one compute node. This rate is significantly lower than the rates of any number of threads started with only one process per node. The contention on the NIC is too high when two processes compete for access and decrease the transpose immensely.



(a) M_SZ = 12288, static notified read

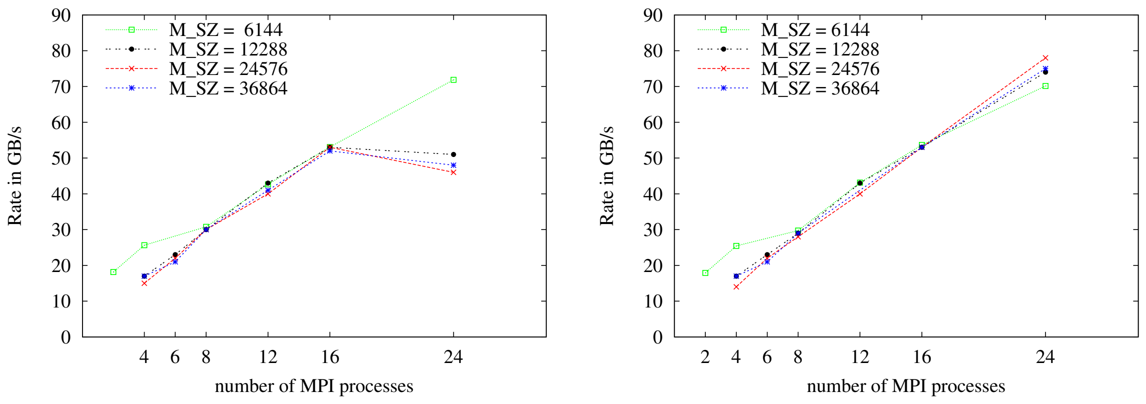


(b) M_SZ = 12288, dynamic notified read

Figure 6.11.: Impact of thread and node count on transpose rates of the `gaspi_read_notify`-based pipelined matrix transpose on Aenigma.

The same can be seen in Fig. 6.11b, where the rates of the `gaspi_read_notify`-based implementation with a dynamic distribution of work is shown. The slope of the rate decreases significantly for more than 16 started nodes for all but the dynamic `gaspi_read_notify` implementation with one process and six threads per node. These first results confirm the presumption that the dynamic work distribution and communication show better scalability than a static distribution of work and communication. In addition, these results show that the sharing of resources when starting two processes per node has an immense impact on the transpose rate. The transpose rate is significantly higher when using only one process per node, while at the same time keeping the total number of threads per node constant. This shows the immense overhead of starting two processes per node. The following experiments were hence restricted to one process per node with 6 threads on Aenigma.

Fig. 6.12 shows the impact of different matrix sizes on the transpose rate of the two notified read based implementations of the pipelined transpose benchmark. Due to the restrictions of the community benchmark, the matrix sizes 6144, 12288, 24576 and 36864 were compared. No significant difference in the transpose rate can be seen - neither in the static nor in the dynamic work distribution implementation - for the original matrix size or larger matrix sizes. The static implementation rates, shown in Fig. 6.12a, all decrease for more than 16 MPI processes, except for the smallest matrix size. In that case, the rate keeps on scaling. The second implementation on the other hand shows a steady increase of the transpose rate for all matrix sizes (Fig. 6.12b). The same is true for the original write-based implementation of the matrix transpose, shown in Fig. D.10.

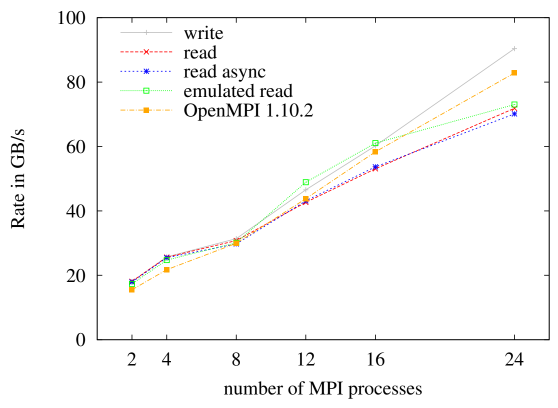


(a) static notified read **(b)** dynamic notified read

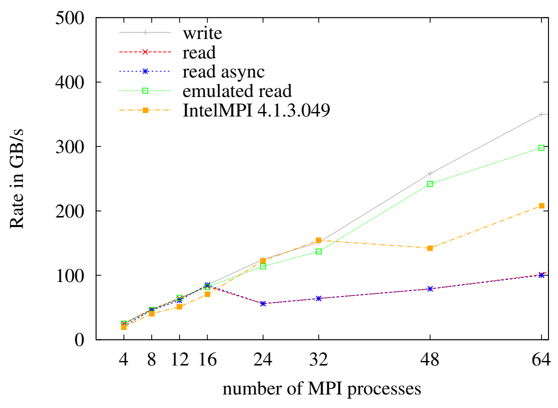
Figure 6.12.: Matrix size impact on transpose rates for the two implementations with `gaspi_write_notify` on Aenigma. Each started with one (1ppn) process per node and 6 threads per process.

After these preparatory experiments, investigating the influence of different parameters on the matrix transpose kernel, the parameters for a direct comparison between all different implementations of the benchmark suite and the read-based implementations can be set. One MPI process per node with six threads each were started on Aenigma. The matrix size chosen is `M_SZ` $= 6144$, because even though the transpose rates did not change too much with different matrix sizes, the runtimes were higher, so the choice fell on the smallest tested matrix size to reduce the actual runtime of the benchmark.

In Fig. 6.13a the results of the comparison on Aenigma are shown. All implementations show a similar transpose rate for small numbers of processes. Starting with 12 processes, differences between the different implementations can be seen. The write-based implementation scales very well and shows a steadily increasing transpose rate. Also the emulated read implementation shows a good scaling behavior up to 16 processes but then the slope decreases down to the two `gaspi_read_notify`-based implementations, which show the worst transpose rates from 12 processes on. The MPI alltoall implementation of the benchmark shows almost as good transpose rates as the write-based implementation, even for 24 processes, where the other implementations have already significantly slowed their scaling.



(a) Aenigma



(b) MareNostrum III

Figure 6.13.: Direct comparison between the different implementations on (a) Aenigma and (b) MareNostrum III.

On MareNostrum III, the results from the experiments on Aenigma were taken into account and thus the following setting was tested: one process per node, 8 threads per process, and `M_SZ` = 12288. The process and thread count was adopted directly from the previous tests on Aenigma. Since MareNostrum III has eight cores per socket, also the thread count was increased to eight. The chosen matrix size was not the smallest one tested on Aenigma, but rather the initial benchmark size. Figure 6.13b shows the transpose rates of the different implementations on MareNostrum III. The transpose rates of the two `gaspi_read_notify`-based implementations are almost indistinguishable and both decline when using more than 16 processes. The MPI alltoall implementation faces a similar problem for more than 32 nodes, while the two implementations presented in the community benchmark keep on scaling. The experiment was not conducted for higher numbers of processes, because the trend of the current implementation seems clear.

Both the experimental results on Aenigma as well as those on MareNostrum III differ greatly from those presented in the PGAS community benchmark, where the system architecture was very different from the two architectures presented here. This underlines the dependence of the performance of communication routines on the underlying hardware. Future research must hence extend to other platforms to gain further understanding of the capabilities of a notified read. Considering only the results gained in these experiments, an implementation of a GASPI_COLL alltoall routine should not be based on `gaspi_read_notify`.

6.6. Discussion

This chapter introduced a new notified read routine, which will be included in the next version of the GASPI specification. Current progress on the specification can be found on the GitHub page of the GASPI Forum [35]. The `gaspi_read_notify` routine extends the idea of a completion notification, already available for write-based GASPI routines, to read-based routines. With this routine, not only the requested data will be read from (remote) memory, but furthermore a local notification will be set. This notification can be queried by the local process with `gaspi_notify_waitsome` and acknowledges that the requested data has been written to the local memory. To enable this fine-grained notification mechanism, i.e., a notification associated with exactly one message, the semantic of `gaspi_write_notify` will also be changed in the next GASPI specification, based on [94].

After introducing the routine, the chapter also presents an IB Verbs specific reference implementation for `gaspi_read_notify` in the scope of GPI2-1.1.1. This implementation relies on the ordering of messages as defined in [61], which is also the basis for the GASPI implementation GPI2, thus maintaining the restrictions already imposed by the implementation. Different experiments and benchmarks were then performed to investigate the potential of this specific notified read implementation.

The first experiment was an adaption of the classical ping pong benchmark, where the time between the issuing of a notified read and the arrival of the data is measured for different messages sizes. The ping pong benchmark delivered with GPI2 was taken as a basis and adapted for the `gaspi_read_notify` routine. In addition to the native implementation of a notified read, also two emulated notified reads were benchmarked: RWN and NWN. In the first version, one read is started per queue and before the next read request is issued in this queue, the calling process waits on the queue and then issues a local notification. The second version needs the active participation of the remote process, first notifying the remote process and then waiting on the data and notification written by the remote process. All three implementations were tested with different numbers of threads and different pinning models.

Even though all three implementations show similar runtime results, certain advantages of the `gaspi_read_notify` routine over the two emulated implementations need to be emphasized. First of all, the RWN implementation has a high demand on queues. Only issuing one read per queue makes this emulated read very purpose specific, as it can not be used as a part of an application where also other communication routines are used. The RWN would block the queues for other communication requests. In addition the number of in-flight read request is limited to the number of queues available in the specific GASPI implementation - limiting it to 16 in GPI2-1.3.0. The second emulated notified read using write operations (NWN), is not limited to the number of available queues or blocks queues in an unfair manner, inapplicable to any communication intense application. Instead, this emulated notified read involves the remote rank in the communication, thus undermining all principles of asynchronism and one-sided communication driving the GASPI communication universe. The completion of this emulated notified read depends on the status of the remote process, and is not suitable for any truly one-sided and asynchronous communication scheme. Thus, the native implementation of `gaspi_read_notify` is the only long-term alternative. Nonetheless, NWN and RWN can be alternatives for application specific communication schemes that are not affected by the above mentioned restrictions.

Apart from the basic ping pong benchmark, the `gaspi_read_notify` implementation was also tested in two use-case scenarios: a graph traversal and a matrix transpose. The graph traversal is motivated by the requirements data analysis faces in the age of big data. In many fields like health care, weather forecast and research in more general, the amount of available data has reached a threshold where thorough analysis needs immense computational power and at the same time has high memory requirements. These prerequisites can be met by purpose built systems like the Urika-GD, but can also be implemented in a PGAS environment. The capability of the PGAS to be expanded almost arbitrarily handles the memory requirements of big data analysis, and the one-sided, asynchronous communication deals with the communication overhead which is usually the problem in distributed memory applications. The applicability of `gaspi_read_notify` to this scenario is demonstrated in a very theoretic manner in this chapter, but first experiments show that GASPI is very well capable of handling numerous `gaspi_read_notify` requests, issued in such a graph exploration scenario. Further research will have to include the investigation of the scalability with larger message sizes and the overall scalability on larger clusters. As the experiments on MareNostrum III have already shown, the runtime significantly increases for large messages on higher number of nodes. Since this is not the case for smaller messages, real-world graph analysis with `gaspi_read_notify` might have to include some scheduling mechanisms. GASPI implementation specific limitations, like the number of queues and the number of segments available for communication, can be handled internally at the cost of additional management overhead. For the systems the graph exploration has been tested on, this was not necessary, but when future processors introduce further parallelism through additionally available threads, either the GASPI implementation can adapt, or these issues will have to be dealt with in the application.

The second use-case, the matrix transpose, was especially chosen with respect to a possible implementation of a partially evaluable alltoallv in GASPI [25] or GASPI_COLL. The communication scheme of a alltoall can be seen as a more general version of the matrix transpose, which was already available as a PGAS community benchmark on GitHub [92]. The pipelined matrix transpose in the community benchmark is available with a `gaspi_write`-based implementation, a RWN-based implementation and two MPI implementations. Thus, the perfect setting to also test a `gaspi_read_notify`-based implementation of a matrix transpose. The matrix transpose was implemented in two different notified read versions: with a static work distribution, taken over from the original benchmark, and a dynamic work distribution. In the experiments, there was no great runtime difference between these two implementations and they will hence be grouped together in the following discussion.

As expected from the original benchmark, the write-based matrix transpose was the fastest implementation on the two available clusters. Very much unlike the original benchmark on a

7D enhanced hypercube with a IB FDR interconnect, the RWN-based implementation does not reach a higher transpose rate than the write-based implementation. Especially the MPI implementations perform much better on Aenigma and MareNostrum III than on the cluster used in the community benchmark, which drastically emphasizes the high impact of underlying hardware on the communication performance. In all cases tested in the scope of this thesis, the `gaspi_read_notify`-based implementations perform significantly worse than the original implementations. Considering the mentioned hardware-dependence, it would be very interesting to see the results of the notified read implementation on a system like the one used in the community benchmark. However, there are also other conclusions to be drawn from this benchmark. For an implementation of alltoall or alltoallv - whether in the scope of GASPI or a library like GASPI_COLL - a write-based implementation is to be preferred over a read-based implementation. Even though the community benchmark shows excellent results for the RWN-based implementation of the matrix transpose, these results could not be totally confirmed. In addition, the restrictions discussed in the scope of the ping pong experiments also apply here, i.e., a standalone matrix transpose kernel can gain from a RWN, but in an application with more communication, the RWN-based implementation will be too resource wasting. The results of this chapter necessarily need to be regarded in the right context: the benchmarks are all standalone communication kernels, from which important conclusions can be drawn for similar applications, but the results can not be directly transferred to more complex applications including more communication.

All in all, the results of the experiments in this chapter especially show one thing: much more research is necessary on notified reads. It has become very clear, that the performance of applications including any kind of communication are extremely hardware dependent. Especially considering notified reads, the native implementation may gain from future hardware developments or alternative implementation methods. One such possible alternative implementation of an IB Verbs-based notified read could be to already internally poll on the completion queue, i.e., take the RWN implementation one level lower. This would not fit the GASPI semantic, where `gaspi_notify_waitsome` is used to query the completion of such a notified read. For a read with internal polling, a different kind of weak synchronization mechanism will have to be implemented, which was outside the scope of this thesis. Another important aspect for future research is the possibility to implement `gaspi_read_notify` on other architectures and systems, like Cray Aries networks.

Another possible use-case for `gaspi_read_notify` could be the implementation of reduce and broadcast routines for GASPI_COLL. The next chapter will conclude the research presented in this thesis and point to further possible research fields in the future.

7. Conclusion

This thesis presented different research areas in the scope of communication routines in GASPI, highlighting the dependencies between HPC applications, especially communication, and the underlying hardware. Chapter 2 introduced all relevant HPC hardware components and their connection to this thesis. This started with different memory architectures and their connection to different communication schemes like message-passing in distributed memory architectures and implicit communication in shared memory architectures. The combination of certain aspects of these two principles and the development of RDMA capable hardware has led to the development of the PGAS. Whether a HPC system is capable of RDMA depends on the underlying network, i.e., the interconnect and the network protocols, where RoCE at least offers a RDMA protocol and IB even enables RDMA communication completely bypassing the CPU. The last section of the chapter then presented an introduction to different communication types, with a special emphasis on collective communication routines and algorithms.
Chapter 3 then established a state of the art overview of HPC communication APIs, libraries and languages. The first section gave a historic overview of the relation between communication development and hardware development. Then, several specifications gained special attention due to their relevance to this thesis. GASNet and IB Verbs were introduced as low-level APIs, which can be used as a basis for high-level libraries like GASPI or some of the other PGAS languages which were described. OpenMP and MPI were described as de-facto standards of shared, respectively distributed memory communication interfaces. Following the description of MPI, also GPI was described as the predecessor of GASPI and last but not least the GASPI specification was described. With these two introductory chapters, the context for the research on different communication schemes in the scope of GASPI was set. Chapters 4 to 6 then dealt with the research conducted for collective communication routines and one-sided communication schemes in the GASPI universe and presented the contributions of this thesis, in the same order as listed in the introduction, i.e.,

1. the adaption of the n-way dissemination algorithm,
2. the collective library GASPI_COLL and
3. the notified read routine for GASPI.

These contributions and all implications arising from them will now be summarized and put into a larger context in the following. In addition, pointers to future research will be given where applicable.

After different algorithms for collective communication routines like the allreduce and especially the barrier had been introduced in Sec. 2.3.3, Chap. 4 presented an adaption to the n-way dissemination algorithm. The algorithm is an extension of the dissemination algorithm which is widely used in barrier operations due to its great performance. Compared to classical tree algorithms, the n-way dissemination algorithm needs much fewer communication rounds - the only important factor when dealing with small messages on today's HPC interconnects - namely $\lceil \log_{n+1}(P) \rceil$ compared to $2 \cdot \lceil \log_2(P) \rceil$ communication rounds. This immense advantage can also be exploited for allreduce routines, but necessitates some adaptions in the communication scheme to ensure the computation of correct reduction results. The adapted n-way dissemination algorithm shows significant performance improvements compared to the native GPI allreduce implementation, which is based on the binomial spanning tree (BST) algorithm. Even though the adapted n-way dissemination algorithm is not suitable for all reduction operations and the adaption presented is not feasible for all group sizes, the runtimes of allreduce and barrier routines can significantly benefit from this algorithm. An allreduce implementation should always be based on several different algorithms to ensure best possible runtimes for all combinations of reduction operations, datatypes and group sizes. Thus, the adapted n-way dissemination algorithm can complement existing implementations.

An algorithm that necessarily needs mentioning in this context is the n-port global combine algorithm developed by Bruck. This algorithm has a very similar communication scheme, is usable for all group sizes and mainly differs from the n-way dissemination algorithm in the computation of the final result of a reduction. The additional computation necessary in Bruck's algorithm might affect the runtime in large scale applications with very computation intense reduction routines, but this could not be shown in the scope of this thesis. A direct comparison of runtimes between Bruck's algorithm and the adapted n-way dissemination algorithm was then shown in the scope of GASPI_COLL in the next chapter.

Chapter 5 then dealt with the implementation of collective communication routines int the scope of an external library, amending the collective routines defined in the GASPI specification. The allreduce was implemented with different algorithms and additionally a broadcast and reduce were implemented. GASPI_COLL is meant to be a portable GASPI library and thus all routines were implemented with one-sided GASPI routines and weak synchronization primitives. The implementation of the collective routines as GASPI applications gave great insight on the obstacles presented by a completely one-sided and asynchronous communication.

First, the allreduce was implemented with the BST, the pairwise exchange (PE), Bruck's and the adapted n-way dissemination algorithm. In implementing the n-way dissemination algorithm as a GASPI application, after having previously implemented it as an ibverbs application, already gave some insight on the overhead caused by necessarily including an additional layer of indirection and emphasizes the importance of including the different algorithms for the allreduce in the native GASPI implementation. The implementation of different algorithms with a butterfly-like communication scheme, i.e., all but the BST algorithm, showed that the development of hardware has made the use of these algorithms plausible again. While former systems suffered from congestion when these algorithms are used, the experiments in this chapter showed that especially for larger messages the butterfly-like communication schemes showed faster runtimes than the tree-based algorithm. The allreduce experiments also showed the impact of different hardware on the runtimes of the algorithms, as well as the importance of knowing whether to start a GASPI application with one process per node or one process per socket, especially in direct comparison to the MPI implementations.

The reduce and broadcast routines were both implemented with a BST algorithm. The difference in runtime to the native MPI implementations is in an acceptable range, considering that the GASPI_COLL routines are implemented with GASPI routines instead of ibverbs. The most important result of the implementation of the reduce and the broadcast was rather the finding of implementation pitfalls with one-sided asynchronous routines. While classical two-sided, message-passing routines could not overwrite the data within the routine, this is dramatically changed in a one-sided communication scheme. Especially in one-to-all and all-to-one collective routines, the obstacles implied through a purely local view on the status of routines and data become very apparent. While the communication scheme of an allreduce automatically notifies all participating ranks that all ranks have entered the routine (otherwise no global result could be computed), this does not hold true for broadcast and reduce operations. To ensure that data from a previous reduce or broadcast is not overwritten before the remote rank has used that data, some additional acknowledgment mechanism will have to be implemented. This does not necessarily match the semantic defined in the GASPI specification, but is necessary for a user-friendly and even usable collective routine. Future research on applicable algorithms and possible acknowledgment implementation will have to be done.

Chapter 6 introduces a completely new routine: a notified read for GASPI. This routine, together with a new notification semantic, has been added to the GASPI specification 16.6 [34] by the time of publication of this thesis. This change in semantic is made to introduce a more fine-grained notification possibility, moving away from the fencing definition in the previous specification to a message-wise notification mechanism. The notification of routines with a combination of data transferal and weak synchronization mechanism, i.e., `gaspi_read_notify` and `gaspi_write_notify`, now only imply the arrival of the data associated with this call and make no further implications on other written or read data. This routine can be used for many different applications, especially in consumer-producer based communication models. With `gaspi_read_notify`, the working nodes are able to transfer remote data into their local memory when they need it. Thus, the dependence on the status of computation of the remote node is mitigated and the load on the network is distributed over the complete runtime of the application instead of being bundled at certain hot spots. This also enables autonomous dynamic work distribution in dependence of the work load on single nodes instead of a static distribution of work, possibly leading to immense idling times of single nodes that are faster in finishing their work. This is not necessarily a consequence of initial non-optimal work partitioning, but may be a result of sudden contention or failure of single nodes.

Two use-cases were described and analyzed in this chapter: a graph exploration and a matrix transpose. The PGAS introduces the possibility of highly scalable applications, able to cope with the high amounts of data being analyzed today. The graph exploration implementation serves as a proof of concept for big data analytical problems, which can be implemented in a completely asynchronous, one-sided and distributed fashion through `gaspi_read_notify`. An implementation of a graph traversal with a notified read eliminates the involvement of remote nodes in the communication and as a direct consequence frees every compute node from the dependence on the computation status of remote nodes. Through the ability of modern networks like IB to bypass the OS in the communication, no CPU resources are necessary for communication and can rather be used for the actual analysis of data. The trend of increasing bandwidth in this sector of HPC interconnects will additionally support this communication strategy. The mere possibility of implementing a graph exploration application with `gaspi_read_notify` liberates the user from purpose built hardware like the UrikaGD.

The second use case was a matrix transpose, which also served as a case study of the usability of a notified read in alltoall implementations. The results of the experiments showed, that an alltoall implementation should rather be based on write operations than on read implementations - at least on the systems available for testing in this thesis. The benchmark used for the matrix transpose kernel was also tested on another system by the developer of the benchmark. The results on that system imply that on other systems, the read-based implementation of a matrix transpose or an alltoall might still make sense. This once again shows the hardware dependence of HPC communication routines and applications. Additional use cases for the notified write may then also be other collective communication routines like the reduce or broadcast. This will have to be investigated in the future.

The dependence of communication routines and algorithms on the underlying hardware was a hot topic throughout the thesis. In every field of research, the results were different on each test system. While this is nothing to be worried about, this underlines the importance of ongoing research in HPC communication. The algorithms and communication schemes will have to be adapted for each architecture and underlying communication protocols. This especially includes the necessity to revisit older algorithms and reevaluate their usability for modern topologies and interconnects, as shown in this thesis. In addition, the expansion of RDMA to other interconnect technologies is to be expected and might unravel new possibilities, especially in the context of notified read or native collective operations.

Glossary

Aenigma Test System, description on p. 71.

API application programming interface

BMBF Bundesministerium für Bildung und Forschung

BST binomial spanning tree

CAF Co-Array Fortran

CASE Test System, description on p. 71.

CPU central processing unit

CQ completion queue

CRAFT Cray Research Adaptive Fortran

DMA direct memory access

DMAPP Distributed Memory Application

FDR InfiniBand Fourteen Data Rate

FDR-10 InfiniBand network with a theoretical data rate approximately ten times as high as the single data rate. This specification is proprietary to Mellanox.

GAS global address space

GASNet Global Address Space Networking

GASPI Global Address Space Programming Interface

GASPI_COLL GASPI library implementing collective communication routines.

GCC GNU Compiler Collection

GPI Global Programming Interface

GPI2 Implementation of the GASPI specification by the Fraunhofer ITWM.

GRH Global Route Header

HPC high performance computing

HPCS High Productivity Computing Systems

HPF High Performance Fortran

IB InfiniBand

IB Verbs General definition of the InfiniBand Verbs.

IBTA InfiniBand Trade Association

ibverbs OFED implementation of the IB Verbs.

IP Internet Protocol

LRH Local Route Header

MareNostrum III Test System, description on p. 86.

MPI Message-Passing Interface

MPIF Message-Passing Interface Forum

NIC network interface controller

NUMA non-uniform memory access

NWN GASPI notified read implementation with `gaspi_notify` and `gaspi_write_notify`

OFED OpenFabrics Enterprise Distribution

OpenMP Open Multi-Processing

OS operating system

OSI Open Systems Interconnection

PE pairwise exchange

PGAS partitioned global address space

QDR InfiniBand Quad Data Rate.

QoS Quality of Service

QP queue pair

RAM random access memory

RDMA remote direct memory access

RMA remote memory access

RoCE RDMA over converged Ethernet

RWN GASPI notified read implementation with `gaspi_read`, `gaspi_wait` and `gaspi_notify`

TCP Transmission Control Protocol

UDP User Datagram Protocol

UMA uniform memory access

UPC Unified Parallel C

List of Figures

Listings

List of Tables

Bibliography

[1] Titanium homepage. http://titanium.cs.berkeley.edu/. retrieved 2016.08.24 at 16:28.

[2] Y. Ajima, Y. Takagi, T. Inoue, S. Hiramoto, and T. Shimizu. The tofu interconnect. In *2011 IEEE 19th Annual Symposium on High Performance Interconnects*, pages 87–94, Aug 2011.

[3] B. Alverson, E. Froese, L. Kaplan, and D. Roweth. Cray XC Series Network. http://www.cray.com/Assets/PDF/products/xc/CrayXC30Networking.pdf, 2012. retrieved 2016.11.07 at 10:54.

[4] ANSI. *American National Standard Programming Language FORTRAN.* American National Standards Institute, 1978.

[5] D. Bonachea. Readme file of gasnet. http://gasnet.lbl.gov/dist/README. retrieved 2016.10.28 at 12:37.

[6] D. Bonachea. GASNet Specification, v1.1. Technical Report UCB/CSD-02-1207, UC Berkeley, Oktober 2002.

[7] E. D. Brooks. The Butterfly Barrier. *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[8] E. D. Brooks, B. C. Gorda, K. H. Warren, and T. S. Welcome. Bbn tc2000 architecture and programming models. In *Compcon Spring '91. Digest of Papers*, pages 46–50, Feb 1991.

[9] J. Bruck and C.-T. Ho. Efficient global combine operations in multi-port message-passing systems. *Parallel Processing Letters*, 3(04):335–346, 1993.

[10] W. Carlson, J. Draper, D. Cullera, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, University of California-Berkeley, 1999.

[11] B. Chamberlain. Chapel: Productive Parallel Programming. http://www.cray.com/blog/chapel-productive-parallel-programming/, May 2013. retrieved 2016.08.24 at 16:21.

[12] C. Clos. A Study of Non-Blocking Switching Networks. *The Bell System Technical Journal*, 32(2):406–424, March 1953.

[13] Coarray* support in gfortran as specified in the fortran 2008 standard. https://gcc.gnu.org/wiki/Coarray. retrieved 2016.08.24 at 16:09.

[14] The C++ programming language - homepage. https://isocpp.org/std/the-standard.

[15] Cray Inc. Cray®Urika®-GX Agile Analytics Platform for Healthcare and the Life Sciences. http://www.cray.com/sites/default/files/Urika-GX-Healthcare-Life-Sciences.pdf. retrieved 2016.11.08 at 09:05.

[16] Cray Inc. News analytics. http://www.cray.com/Assets/PDF/products/urika-gd/Urika-GD-SB-NewsAnalytics.pdf, September 2014. retrieved 2016.11.08 at 09:05.

[17] Cray Inc. Real-Time Discovery in Big Data Using the Urika-GD Appliance. http://www.cray.com/sites/default/files/resources/Urika-GD-WhitePaper.pdf, October 2014. retrieved 2016.11.08 at 09:05.

[18] Cray Inc. Using the GNI and DMAPP APIs, S–2446–52. http://docs.cray.com/books/S-2446-52/S-2446-52.pdf, Feb 2014. retrieved 2016.11.08 at 09:05.

[19] Cray Inc. Chapel language specification version 0.981. http://chapel.cray.com/docs/latest/_downloads/chapelLanguageSpec.pdf, April 2016. retireved 2016.07.24 at 13:40.

[20] Cray Inc. Cray® Urika®-GX Agile Analytics Platform for Earth Sciences. http://www.cray.com/sites/default/files/Urika-GX-Earth-Sciences.pdf, 2016. retrieved 2016.11.08 at 09:05.

[21] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter. Partitioned global address space languages. *ACM Comput. Surv.*, 47(4):62:1–62:27, May 2015.

[22] L. Dekker, W. Smit, and J. Zuidervaart. Massively parallel processing applications and development. In *Proceedings of the 1994 EUROSIM Conference on Massively Parallel Processing Applications and Development*, Delft, The Netherlands, June 1994. Elsevier Science.

[23] J. Dongarra. Report on the sunway taihulight system,. Department of Electrical Engineering and Computer Science Tech Report UT-EECS-16-742, University of Tennessee, June 2016.

[24] S. Ekanayake. Survey on high productivity computing systems (hpcs) languages. http://grids.ucs.indiana.edu/ptliupages/publications/Survey_on_HPCS_Languages_formatted_v2.pdf, 2013. retrieved 2016.09.16 at 15:08.

[25] V. End and C. Simmendinger. Proposal to the GASPI Specification - Inclusion of gaspi_alltoallV. http://www.gaspi.de/readings/alltoall_gaspi_style.pdf, Jan 2016. retrieved 2016.10.25 at 14:55.

[26] V. End, C. Simmendinger, R. Yahyapour, and T. Alrutz. Butterfly-like Algorithms for GASPI Split-Phase Allreduce. *International Journal on Advances in Systems and Measurements*, 9(3&4), Dec 2016.

[27] V. End, R. Yahyapour, C. Simmendinger, and T. Alrutz. Adapting the n-way Dissemination Algorithm for GASPI Split-Phase Allreduce. In *INFOCOMP 2015, The Fifth International Conference on Advanced Communications and Computation*, pages 13 – 19, June 2015.

[28] Fraunhofer ITWM. GPI2 homepage. www.gpi-site.com/gpi2. retrieved 2016.11.08 at 09:07.

[29] E. Freudenthal and A. Gottlieb. Process Coordination through Fetch-And-Increment. In *Proceedings of the ACM Fourth International Converence on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[30] Fujitsu. Innovative "6-Dimensional Mesh/Torus" Topology Network Technology. http://www.fujitsu.com/global/about/businesspolicy/tech/k/whatis/network/. retrieved 2016.11.08 at 09:08.

[31] GASNet website. http://gasnet.cs.berkeley.edu. retrieved 2016.07.01.

[32] GASPI Consortium. GASPI Homepage. http://www.gaspi.de. retrieved 2016.11.08 at 09:14.

[33] GASPI Consortium. GASPI: Global Address Space Programming Interface, Specification of a PGAS API for communication Version 16.1. https://raw.githubusercontent.com/GASPI-Forum/GASPI-Forum.github.io/master/standards/GASPI-16.1.pdf, February 2016. retrieved 2016.11.08 at 09:12.

[34] GASPI Consortium. GASPI: Global Address Space Programming Interface, Specification of a PGAS API for communication Version 16.6. https://raw.githubusercontent.com/GASPI-Forum/GASPI-Forum.github.io/master/standards/GASPI-16.6.pdf, September 2016. online from 2016.11.30; retrieved 2017.01.25 at 10:59.

[35] GASPI Forum. GitHub Repository of the GASPI Standard. https://github.com/GASPI-Forum/GASPI-Standard. retrieved 2016.10.25 at 13:18.

[36] A. Geiger. *Höchstleistungsrechnen in den Ingenieurwissenschaften.* Habilitationsschrift, Univ., Habil.-Schr.–Stuttgart, 1999, Stuttgart, 2000.

[37] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. The Java®Language Specification Java SE 8 Edition. http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf, Feb 2015. retrieved 2016.07.24. at 14:32.

[38] Global Programming Interface (GPI) User Manual. http://www.gpi-site.com/cms/sites/default/files/gpigetstarted.pdf. retrieved 2016.11.08 at 09:15.

[39] P. Grun. *Introduction to InfiniBand™ for End Users.* InfiniBand® Trade Association, 2010.

[40] D. Grunwald and S. Vajracharya. Efficient Barriers for Distributed Shared Memory Computers. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 604–608. IEEE Computer Society, 1994.

[41] R. Gupta and C. R. Hill. A Scalable Implementation of Barrier Synchronization Using An Adaptive Combining Tree. *International Journal of Parallel Programming*, 18(3):161–180, June 1989.

[42] R. Gupta, V. Tipparaju, J. Nieplocha, and D. Panda. Efficient Barrier using Remote Memory Operations on VIA-Based Clusters. In *IEEE Cluster Computing*, page 83ff. IEEE Computer Society, 2002.

[43] M. Heath, O. Section, and S. Mathematics. *Hypercube Multiprocessors, 1987: Proceedings of the Second Conference on Hypercube Multiprocessors, Knoxville, Tennessee, September 29-October 1, 1986.* Proceedings in Applied Mathematics Series. Siam, 1987.

[44] D. Hensgen, R. A. Finkel, and U. Manber. Two Algorithms for Barrier Synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[45] P. N. Hilfinger, D. O. Bonachea, K. Datta, D. Gay, S. L. Graham, B. R. Liblit, G. Pike, J. Z. Su, and K. A. Yelick. Titanium language reference manual, version 2.19. Technical Report UCB/EECS-2005-15, Electrical Engineering and Computer Sciences, University of California at Berkeley, Nov 2005.

[46] T. Hoefler. What are the real differences between RDMA, InfiniBand, RMA, and PGAS? https://htor.inf.ethz.ch/blog/index.php/2016/05/15/what-are-the-real-differences-between-rdma-infiniband-rma-and-pgas/, May 2016. retrieved 2016.07.29 at 13:33.

[47] T. Hoefler, J. Dinan, D.Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur. Leveraging MPI's One-Sided Communication Interface for Shared Memory Programming. In *Proceedings of the 19th European MPI User's Group Meeting (EuroMPI 2012)*, September 2012.

[48] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. Fast Barrier Synchronization for InfiniBand. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 272–272, Washington, DC, USA, 2006. IEEE Computer Society.

[49] T. Hoefler, T. Melan, F. Mietke, and W. Rehm. A Survey of Barrier Algorithms for Coarse Grained Supercomputers. Chemnitzer Informatik Berichte, Volume 4, Numer 3, 2004.

[50] Höchstleistungsrechenzentrum Stuttgart (HLRS). GPI-2 on the HLRS Wiki. https://wickie.hlrs.de/platforms/index.php/GPI-2. retrieved 2016.08.27 at 09:40.

[51] Höchstleistungsrechenzentrum Stuttgart (HLRS). Hazel hen specification. https://www.hlrs.de/de/systems/cray-xc40-hazel-hen/. retrieved 2016.07.31 at 14:15.

[52] IBM. IBM Blue Gene/Q Systemkonfiguration. http://www-03.ibm.com/systems/de/technicalcomputing/solutions/bluegene/#spec. retrieved 2016.08.12 at 13:39.

[53] IEEE P802.3bs 400 Gb/s Ethernet Task Force. IEEE P802.3bs 400GbE Adopted Timeline. http://www.ieee802.org/3/bs/timeline_3bs_0514.pdf, May 2014. retrieved 2016.11.08 at 09:15.

[54] IEEE Standards Association. IEEE 802.3 - IEEE Standard for Ethernet. http://standards.ieee.org/about/get/802/802.3.html. retrieved 2016.11.08 at 09:16.

[55] IEEE Standards Association. Ieee 802.3™ethernet standards milestones. http://standards.ieee.org/events/ethernet/timeline.pdf, 2013. retrieved 2016.07.29 at 17:19.

[56] IEEE Standards Organisation. History of Ethernet. http://standards.ieee.org/events/ethernet/history.html. retrieved 2016.11.08 at 09:17.

[57] InfiniBand Trade Association. InfiniBand Roadmap. http://www.infinibandta.org/content/pages.php?pg=technology_overview. retrieved 2016.05.25 at 15:37.

[58] InfiniBand Trade Association. Infiniband architecture specification volume 1, release 1.2.1, annex a16. https://cw.infinibandta.org/document/dl/7148, 2010. retrieved 2016.11.08 at 09:19.

[59] InfiniBand Trade Association. FDR InfiniBand Fact Sheet. https://cw.infinibandta.org/document/dl/7260, 2011. retrieved 2016.09.13 at 15:02.

[60] InfiniBand Trade Association. Infiniband architecture specification volume 1, release 1.2.1, annex a17. https://cw.infinibandta.org/document/dl/7781, 2014. retrieved 2016.07.31 at 13:39.

[61] InfiniBand Trade Association. Infiniband architecture specification volume 1, release 1.3. https://cw.infinibandta.org/document/dl/7859, March 2015. retrieved 2016.11.08 at 09:19.

[62] Intel. Intel omni-path architecture. http://www.intel.de/content/www/de/de/high-performance-computing-fabrics/omni-path-architecture-fabric-overview.html. retrieved 2016.07.31 at 14:46.

[63] Intel Xeon Processor E5-2699 v4 Specifications. http://ark.intel.com/products/91317/Intel-Xeon-Processor-E5-2699-v4-55M-Cache-2_20-GHz. retrieved 2016.07.25 at 12:50.

[64] Intrepid Technology Inc. GNU UPC. http://www.gccupc.org/. retrieved 2016.11.08 at 13:52.

[65] ISO. Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model. ISO/IEC 7498-1, International Organization for Standardization, 1994.

[66] IT4Innovations. 7d enhanced hypercube. https://docs.it4i.cz/salomon/network-1/7d-enhanced-hypercube. retrieved 2016.11.08 at 13:52.

[67] Fraunhofer ITWM homepage. http://www.itwm.fraunhofer.de/en/fraunhofer-itwm.html. retrieved 2016.11.08 at 13:53.

[68] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[69] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-Driven, Highly-Scalable Dragonfly Topology. *SIGARCH Comput. Archit. News*, 36(3):77–88, June 2008.

[70] libibverbs git repository. http://git.kernel.org/cgit/libs/infiniband/libibverbs.git. retrieved 2016.08.24 at 14:50.

[71] A. M. Mainwaring and D. E. Culler. Active Message Applications Programming Interface and Communication Subsystem Organization. Technical Report UCB/CSD-96-918, EECS Department, University of California, Berkeley, Oct 1996.

[72] Mellanox /Dell. Fdr-10 description. http://www.mellanox.com/related-docs/products/oem/RG_Dell_SKU.pdf. retrieved 2016.09.13 at 15:01.

[73] Mellanox Technologies. Mellanox Technologies Mellanox IB-Verbs API (VAPI). http://nuweb12.neu.edu/rc/wp-content/uploads/2013/09/MellanoxVerbsAPI.pdf, 2001. retrieved 2016.07.24 at 13:36.

[74] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *The International Journal of Supercomputer Applications and High-Performance Computing*, 8(1.3), 1994.

[75] Message-Passing Interface Forum. *MPI: A Message Passing Interface Standard, Version 3.0*. High-Performance Computing Center Stuttgart, 09 2012.

[76] J. Milano and P. Lembke. Ibm system blue gene solution: Blue gene/q hardware overview and installation planning. Technical report, May 2013. retrieved 201.07.31 at 14:25.

[77] Message-Passing Interface Forum homepage. http://www.mpi-forum.org. retrieved 2016.11.08 at 13:53.

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

[79] Open Source Software Solutions, Inc. OpenSHMEM Application Programming Interface 1.3. http://www.openshmem.org/site/sites/default/site_files/OpenSHMEM-1.3.pdf, Feb 2016. retrieved 2016.11.07 at 11:03.

[80] OpenFabrics Alliance. OFED Overview. https://www.openfabrics.org/index.php/openfabrics-software.html. retrieved 2016.07.29 at 15:49.

[81] OpenMP Architecture Review Board. OpenMP Application Program Interface. http://www.openmp.org/mp-documents/openmp-4.5.pdf, November 2015. Version 4.5, retrieved 2016.08.24 at 15:51.

[82] Oracle Inc. Java platform standard edition 8 documentation. http://docs.oracle.com/javase/8/docs/. retireved 2016.07.24 at 14:34.

[83] D. Padua. *Encyclopedia of Parallel Computing*, volume 4 of *Encyclopedia of Parallel Computing*. Springer, 2011.

[84] D. M. Pase, T. MacDonald, and A. Meltzer. The CRAFT Fortran Programming Model. *Scientific Programming*, 3(3):pp. 227–253, 1994.

[85] G. F. Pfister and V. A. Norton. 'Hot-Spot' Contention and Combining in Multistage Intercommection Networks. *IEEE Transactions on Computers*, C-34:943–948, October 1985.

[86] PGAS homepage - links. http://pgas.org/index.php?option=com_weblinks&view=category&id=38&Itemid=67. retrieved 2016.07.21.

[87] P. Pierce. The nx/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications: Architecture, Software, Computer Systems, and General Issues - Volume 1*, C3P, pages 384–390, New York, NY, USA, 1988. ACM.

[88] The python language reference. https://docs.python.org/3/reference/index.html. retrieved 2016.07.24 at 14:38.

[89] The python standard library. https://docs.python.org/3/library/index.html#library-index. retrieved 2016.07.24.

[90] T. Rauber and G. Rünger. *Parallel Programming for Multicore and Cluster Systems*. Springer-Verlag Berlin Heidelberg, 2nd edition, 2013.

[91] C. Rice University. High performance fortran language specification. *SIGPLAN Fortran Forum*, 12(4):1–86, Dec. 1993.

[92] C. Simmendinger. GASPI Pipelined Transpose Community Benchmark Code. https://github.com/PGAS-community-benchmarks/Pipelined-Transpose/blob/master/gaspi/Pipelined_transpose_WN.c, April 2016. retrieved 2016.11.04 at 07:54.

[93] C. Simmendinger. GASPI Pipelined Transpose PGAS Community Benchmark Wiki. https://github.com/PGAS-community-benchmarks/Pipelined-Transpose/wiki, April 2016. retrieved 2016.11.04 at 07:54.

[94] C. Simmendinger and V. End. GASPI Proposal: Clarification of gaspi_write_notify Semantic. http://www.gaspi.de/proposals/2016_06_write_notify_clarification_slides.pdf, June 2016. retrieved 2016.11.04 at 07:54.

[95] C. Simmendinger and V. End. Proposal to the GASPI Specification - Introduction of gaspi_read_notify. http://www.gaspi.de/proposals/read_notify_gaspi.pdf, January 2016. retrieved 2016.11.04 at 07:54.

[96] Z. Tang. Th express-2 reaches new heights for supercomputer interconnects. *National Science Review*, 2016.

[97] M. ten Bruggencate and D. Roweth. DMAPP - An API for One-sided Program Models on Baker Systems. In *Cray User Group 2010 Proceedings*, 2010.

[98] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19:49–66, 2005.

[99] The IEEE and The Open Group. The Open Group Base Specification Issue 6, IEEE Std 1003.1, 2004 Edition: pthread.h. http://pubs.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html, 2004. retrieved 2016.11.08 at 13:54.

[100] TOP500 List Statistics. https://www.top500.org/statistics/list/. retrieved 2016.07.29 at 10:05.

[101] TOP500. Top500 list june 2016. https://www.top500.org/lists/2016/06/, June 2016. retrieved 2016.07.31 at 14:09.

[102] N.-F. Tzeng and A. Kongmunvattana. Distributed Shared Memory Systems with Improved Barrier Synchronization and Data Transfer. In *Proceedings of the 11th International Conference on Supercomputing (ICS)*, pages 148–155, 1997.

[103] J. von Neumann. First draft of a report on the edvac. *IEEE Ann. Hist. Comput.*, 15(4):27–75, Oct. 1993.

[104] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, 36(4):388–395, 1987.

[105] S. Yu. IEEE 802.3 "Standard for Ethernet" marks 30 years of innovation and global market growth. http://standards.ieee.org/news/2013/802.3_30anniv.html. Press release, retrieved 2016.07.29 at 16:47.

A. GASNet Conduits

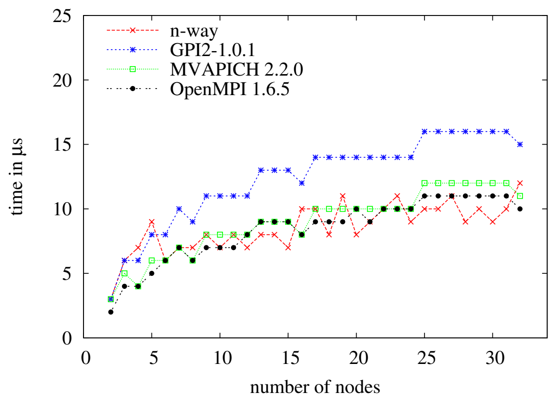
GASNet achieves high portability to different systems through twelve different conduits [5]:

smp	for single shared-memory nodes usage
udp	uses the User Datagram Protocol as a standard component of the TCP/IP suite
mpi	a very general implementation running on all systems with MPI 1.1 and newer
ofi	for OpenFabrics Interfaces
portals4	for systems running Portals 4
pami	for machines using the IBM Parallel Active Messaging Interface, i.e. BlueGene/Q
gemini	for Cray machines with the Gemini interconnect
aries	for Cray machines with the Aries interconnect
ibv	for systems with Open Fabrics Verbs API (ibverbs)
mxm	for systems with Mellanox IB host channel adapters using the Mellanox Messaging Accelerator API
psm	for systems with the Intel Omni-Path Fabric
shmem	for native SHMEM implementations on Cray X1 and SGI Altix

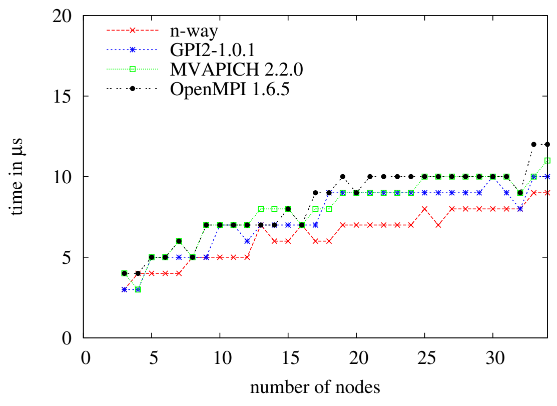
These conduits can be considered as wrappers around low-level communication APIs, which are necessary for a highly performant communication library. A long-term goal for GASPI needs to be similar portability.

B. Appendix to the Adapted n-way Dissemination Algorithm

This appendix includes additional results to runtime tests of the adapted n-way dissemination algorithm as described in Sec. 4.3. Figure B.1a shows the runtime results of the native GPI2 allreduce compared to two available MPI allreduce implementations and the allreduce within GPI2 with the adapted n-way dissemination algorithm for one integer. Figure B.1b shows the equivalent results for barrier implementations. Further discussion on the results can be found in Sec. 4.3.



(a) Allreduce



(b) Barrier

Figure B.1.: Comparison of different allreduces on Aenigma with one integer and the sum as reduction operation, respectively different barriers.

C. Appendix to GASPI_COLL

C.1. Algorithmic Structs of GASPI_COLL

The following structs are necessary for the implementation of the collective routines with different algorithms in GASPI_COLL.

```
typedef struct{

  gaspi_rank_t *peers;
  int n;
  int last_sender;
  int round_sender;
  int boundary_sender;
  int num_rnds;
  int start_new_data_offset[2];

}nway_struct;
```

```
typedef struct{

  gaspi_rank_t **peers;
  int *digits;
  int *numberReceivedElementsInRound;
  int num_msg;
  int num_rnds;
  int start_new_data_offset[2];

}brucks_struct;
```

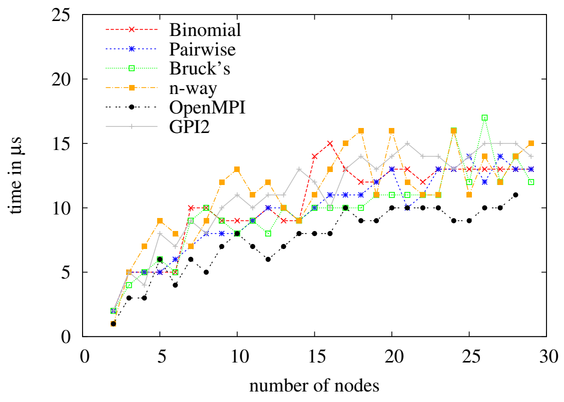
```
typedef struct{

  gaspi_rank_t *peers;
  int pow2;
  int diff;
  int num_rnds;
  int start_new_data_offset[2];

}pairwise_struct;
```

```
typedef struct{

  gaspi_rank_t parent;
  gaspi_rank_t *children;
  int num_children;
  int max_num-children;
  int myOffset;
  int start_new_data_offset[2];

}tree_struct;
```

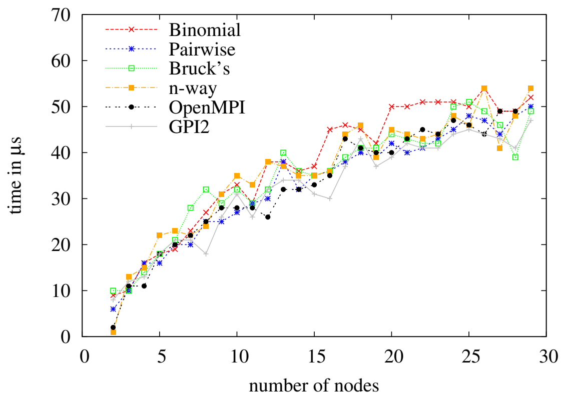
Each of these structs holds all necessary information needed for the different algorithms, the adapted n-way dissemination algorithm, Bruck's algorithm, the PE algorithm and tree-based algorithms like the BST.

C.2. Additional Results of Allreduce Experiments

In this section, additional results from the GASPI_COLL allreduce experiments are presented. The first set of results in Fig. C.1 are from the experiments conducted on Aenigma with the minimum operation used as the reduction operation. The experiments showed almost identical results as the experiments conducted with the maximum operation and discussed in Sec. 5.4.



(a) 1 integer, min



(b) 255 doubles, min

Figure C.1.: Comparison of allreduce implementations with the minimum as reduction operation on Aenigma.

On p. 90, the results of the allreduce experiments on MareNostrum III for the sum as reduction operation are presented in Fig. 5.7. Figure 5.7b showed only partial results for the experiments conducted with one integer and one process per node, to better compare the results with those in Fig. 5.7a. Figure C.2 now shows all results, hindering a good distinction between the single implementations. Instead, the severity of the single higher runtimes of the adapted n-way dissemination algorithm and almost all GASPI_COLL implementations with 72 group members is clearly shown.

On MareNostrum III, the experiments were not only conducted with one GASPI process started per node, but also with one GASPI process started per socket. Figure C.3 shows the additional results of the GASPI_COLL experiments on MareNostrum III with one process per socket. Afterwards, Fig. C.4 shows the results with one GASPI process started per node. All results are discussed in Sec. 5.4, but the figures are presented here to reduce the number of figures in the body of the thesis and to increase the readability. Special note has to be taken of the different range of the y-scale in Fig. C.3d. Instead of ending at 150 μs like the other plots for large messages in this section, the scale goes up to 300 μs to include the single high runtime of the MPI allreduce implementation with 48 processes.

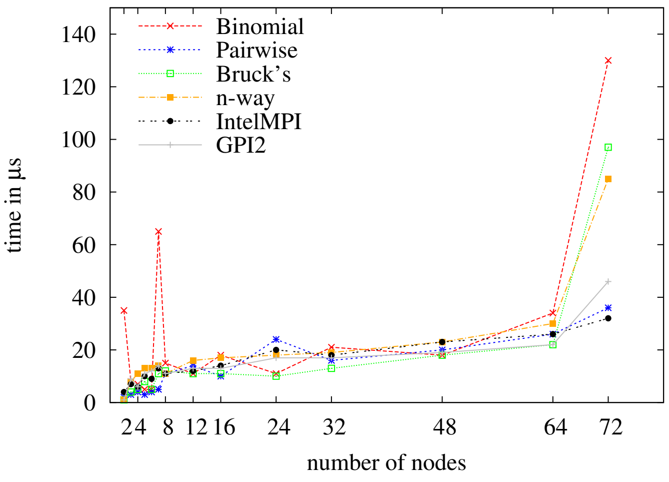
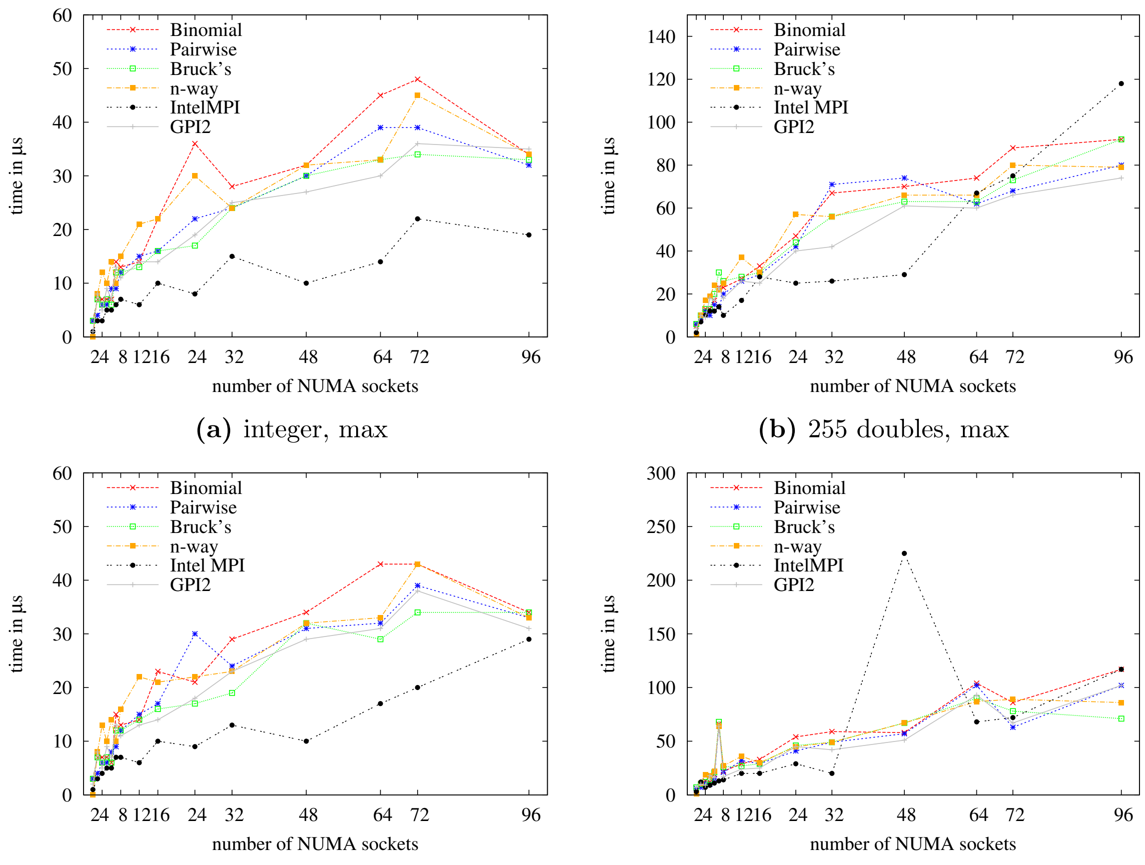


Figure C.2.: Unenlarged comparison of allreduce implementations with sum as reduction operation for one integer on MareNostrum III.



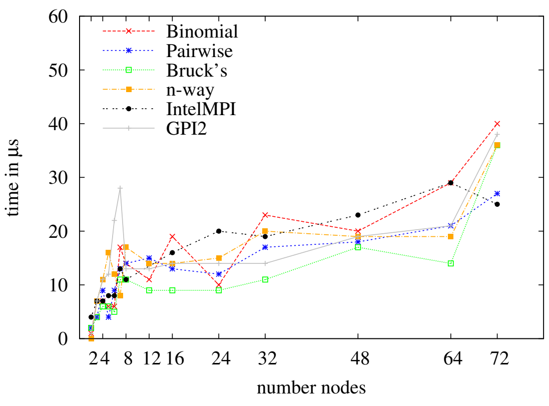
(a) integer, max

(b) 255 doubles, max

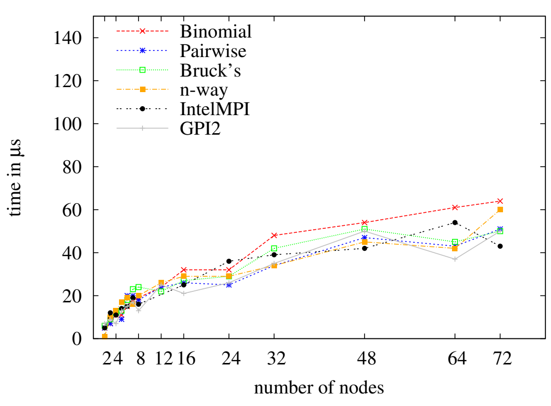
(c) integer, min

(d) 255 doubles, min

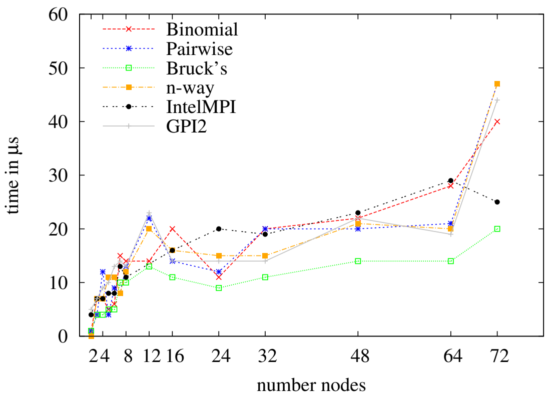
Figure C.3.: Comparison of allreduce implementations with the maximum ((a) and (b)) and the minimum ((c) and (d)) as reduction operations on MareNostrum III. One GASPI process was started per NUMA socket.



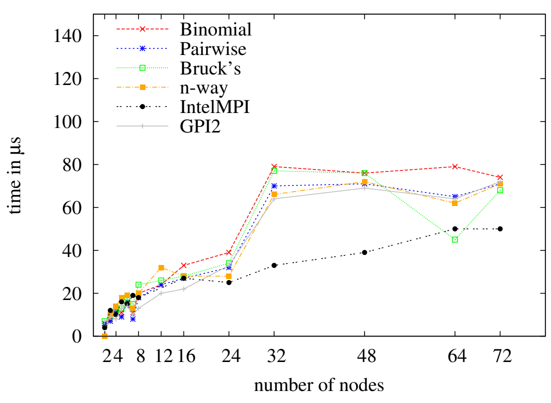
(a) integer, max



(b) 255 double, max



(c) integer, min



(d) 255 doubles, min

Figure C.4.: Comparison of allreduce implementations with the maximum ((a) and (b)) and the minimum ((c) and (d)) as reduction operations on MareNostrum III. One GASPI process was started per node.

D. Appendix to Notified Read

D.1. Notified Read Code

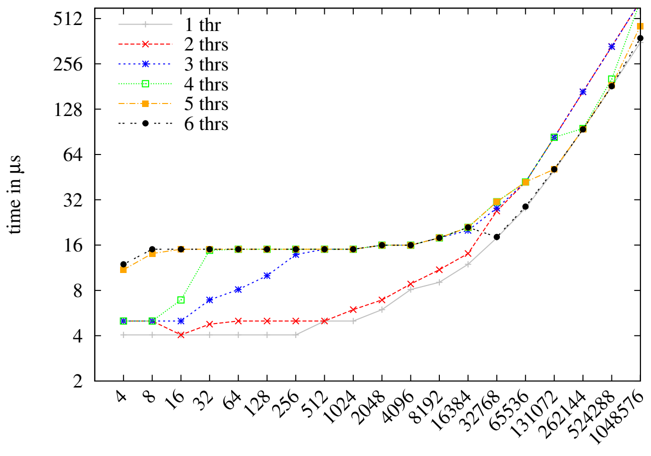
```c
gaspi_return_t
pgaspi_read_notify (const gaspi_segment_id_t segment_id_local,
        const gaspi_offset_t offset_local,
        const gaspi_rank_t rank,
        const gaspi_segment_id_t segment_id_remote,
        const gaspi_offset_t offset_remote,
        const gaspi_size_t size,
        const gaspi_notification_id_t notification_id,
        //const gaspi_notification_t notification_value,
        const gaspi_queue_id_t queue,
        const gaspi_timeout_t timeout_ms)
{

#ifdef DEBUG
  if (!glb_gaspi_init)
    {
      gaspi_print_error("Invalid function before gaspi_proc_init");
      return GASPI_ERROR;
    }

  if(_check_func_params("gaspi_read_notify", segment_id_local, offset_local,
      rank,
      segment_id_remote, offset_remote, size,
      queue, timeout_ms) < 0)
    return GASPI_ERROR;

#endif

  struct ibv_send_wr *bad_wr;
  struct ibv_sge slist, slistN;
  struct ibv_send_wr swr, swrN;

  if(lock_gaspi_tout (&glb_gaspi_ctx.lockC[queue], timeout_ms))
    return GASPI_TIMEOUT;

#ifdef GPI2_CUDA
```

```
  if(glb_gaspi_ctx_ib.rrmd[segment_id_local][glb_gaspi_ctx.rank].cudaDevId >=
   0)
    slist.addr =
      (uintptr_t) (glb_gaspi_ctx_ib.
       rrmd[segment_id_local][glb_gaspi_ctx.rank].addr +
       offset_local);
  else
#endif
    slist.addr =
      (uintptr_t) (glb_gaspi_ctx_ib.
       rrmd[segment_id_local][glb_gaspi_ctx.rank].addr +
       NOTIFY_OFFSET + offset_local);

  slist.length = size;
  slist.lkey =
    glb_gaspi_ctx_ib.rrmd[segment_id_local][glb_gaspi_ctx.rank].mr->lkey;

#ifdef GPI2_CUDA
  if(glb_gaspi_ctx_ib.rrmd[segment_id_remote][rank].cudaDevId >= 0)
    swr.wr.rdma.remote_addr =(glb_gaspi_ctx_ib.rrmd[segment_id_remote][rank].
   addr +
            offset_remote);
  else
#endif
    swr.wr.rdma.remote_addr =
      (glb_gaspi_ctx_ib.rrmd[segment_id_remote][rank].addr + NOTIFY_OFFSET +
       offset_remote);

  swr.wr.rdma.rkey = glb_gaspi_ctx_ib.rrmd[segment_id_remote][rank].rkey;
  swr.sg_list = &slist;
  swr.num_sge = 1;
  swr.wr_id = rank;
  swr.opcode = IBV_WR_RDMA_READ;
  swr.send_flags = IBV_SEND_SIGNALED;
  swr.next = &swrN;

  slistN.addr = (uintptr_t) (glb_gaspi_ctx_ib.rrmd[segment_id_local][
   glb_gaspi_ctx.rank].addr + notification_id * 4);

  //  *((unsigned int *) slistN.addr) = notification_value;

  slistN.length = 4;
  slistN.lkey =
    glb_gaspi_ctx_ib.rrmd[segment_id_local][glb_gaspi_ctx.rank].mr->lkey;

#ifdef GPI2_CUDA
  if((glb_gaspi_ctx_ib.rrmd[segment_id_remote][rank].cudaDevId >= 0))
  {
    swrN.wr.rdma.remote_addr = (glb_gaspi_ctx_ib.rrmd[segment_id_remote][rank
   ].host_addr+notification_id*4);
    swrN.wr.rdma.rkey = glb_gaspi_ctx_ib.rrmd[segment_id_remote][rank].
   host_rkey;
  }
  else
#endif
    {
    swrN.wr.rdma.remote_addr =
      (glb_gaspi_ctx_ib.rrmd[segment_id_remote][rank].addr +
       65536 * 4);
    swrN.wr.rdma.rkey = glb_gaspi_ctx_ib.rrmd[segment_id_remote][rank].rkey;
  }

  swrN.sg_list = &slistN;
  swrN.num_sge = 1;
  swrN.wr_id = rank;
  swrN.opcode = IBV_WR_RDMA_READ;
  swrN.send_flags = IBV_SEND_SIGNALED;
  swrN.next = NULL;

  if (ibv_post_send (glb_gaspi_ctx_ib.qpC[queue][rank], &swr, &bad_wr))
  {
    glb_gaspi_ctx.qp_state_vec[queue][rank] = 1;
    unlock_gaspi (&glb_gaspi_ctx.lockC[queue]);

#ifdef DEBUG
    _print_func_params("gaspi_read_notify", segment_id_local, offset_local,
   rank,
         segment_id_remote, offset_remote, size,
         queue, timeout_ms);
#endif

    return GASPI_ERROR;
  }

  glb_gaspi_ctx_ib.ne_count_c[queue] += 2;

  unlock_gaspi (&glb_gaspi_ctx.lockC[queue]);
  return GASPI_SUCCESS;
}
```

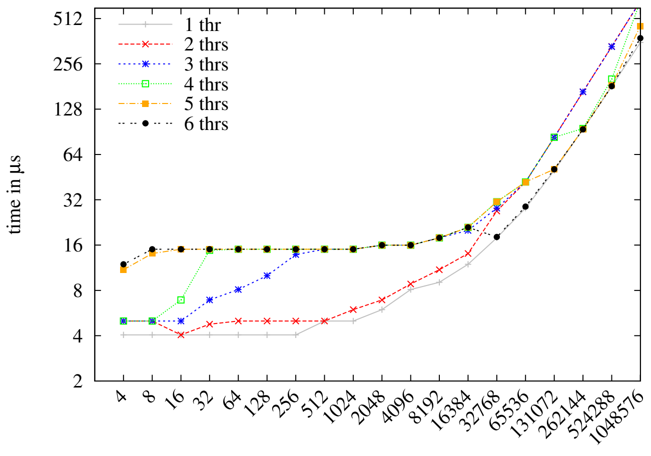
Listing D.1: Implementation of `gaspi_read_notify` as part of the GPI2-1.1.1 implementation.

D.2. Additional Ping Pong Results

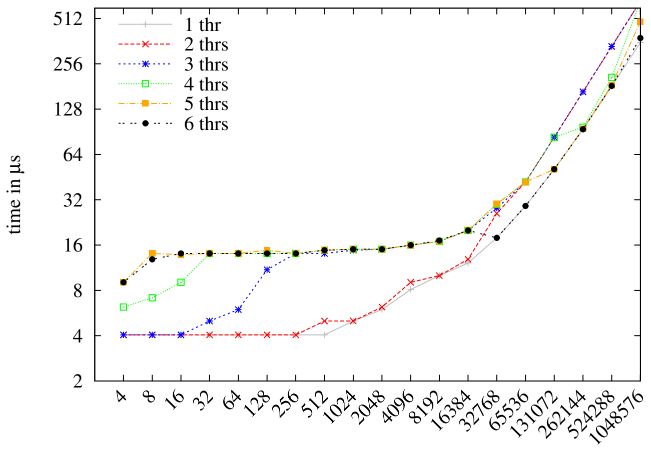
This section shows supplementary figures to the experiments described and results discussed in Sec. 6.3. While the figures in the main part of the thesis only showed results for message sizes of up to 2^{12} B, the following figures show all results of up to 12^{19} B. This choice was made to increase the readability of the figures in the main body and because the most interesting results for the following sections are those of message sizes far below 8 kB. Please note the logarithmic time scales of the plots presented here.
First, the complete results of the ping pong benchmark on Aenigma are shown. Figure D.1 shows the results of the different implementations NWN, RWN and `gaspi_read_notify` without any pinning model applied. Figures D.2 and D.3 show the results for the benchmark run with all threads pinned to core 0 or with the threads each pinned to one of the first <number of threads> cores. As already discussed in Sec. 6.3, the results of the different implementations is almost identical in the different pinning models, i.e., there is not much difference between the implementations at all. The main difference is always, at which message size the runtimes of smaller thread numbers advance those of larger thread numbers.
Exceptions to this are seen in Figs. D.2a and D.3a. In these cases (all threads pinned to core 0 or pinned to the first <number of threads> cores), also larger thread numbers show fast runtimes for very small messages (4 and 8 B). The same behavior is not seen from the other two implementations in the same pinning model. Considering the very small parameter space, in which this implementation shows significantly different results than the other implementations, this is neglectable in the overall discussion.
On Aenigma another pinning version was tested: all threads were pinned to the first <number of threads> evenly numbered cores. In a processor with multiple cores, each core is given a fixed number, which can be looked up in the configuration. With these numbers, threads can be pinned to designated cores. The results of this experiment are shown in Fig. D.5. Since the results of higher thread numbers are very erratic and in addition the runtimes are in general much higher than in the other pinning models, this pinning model is not of interest for HPC applications and was thus not further investigated.
Starting on p. xliv, the results for the experiments on MareNostrum III are shown. The discussion was done in Sec. 6.3 and here additional results are shown. As for the results on Aenigma, Fig. D.6 shows the results for unpinned threads, Fig. D.7 shows the results when all threads are pinned to core 0 and Fig. D.8 shows the results of the experiments with threads pinned to the first <number of threads> cores.



(a) NWN

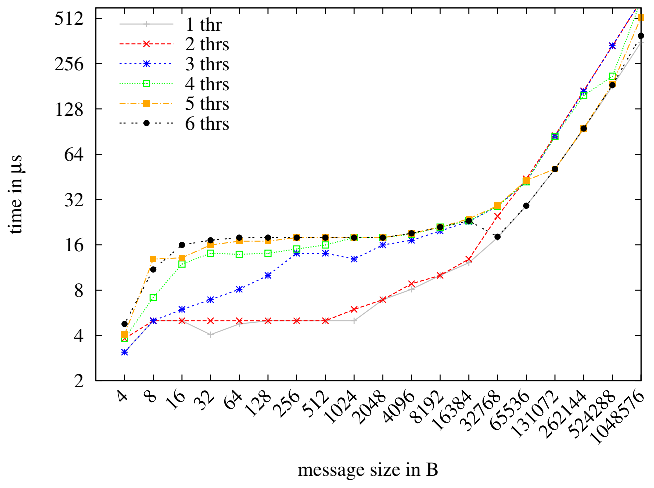


(b) RWN

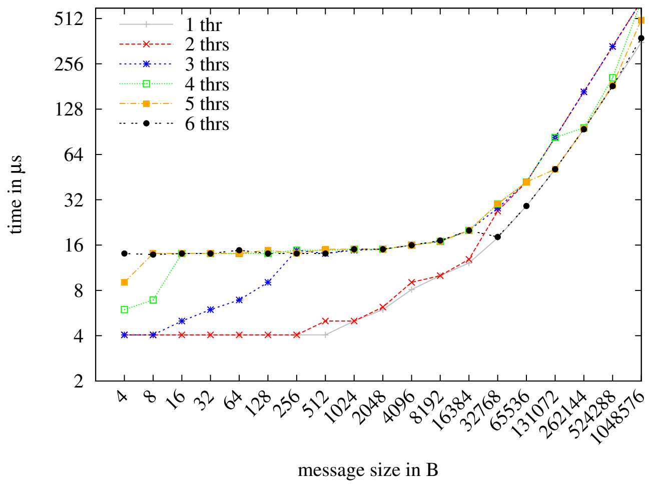


(c) `gaspi_read_notify`

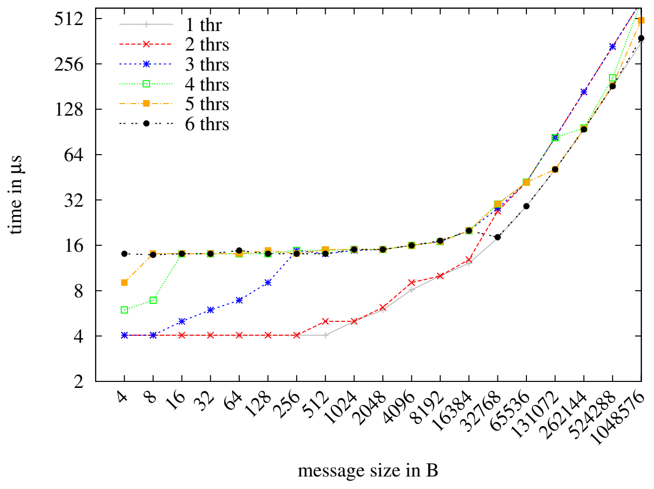
Figure D.1.: Median runtimes for different notified read implementations on Aenigma with unpinned threads.



(a) NWN

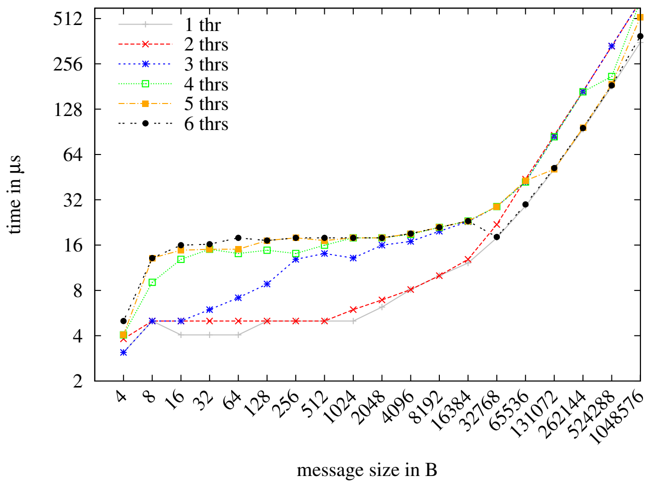


(b) RWN

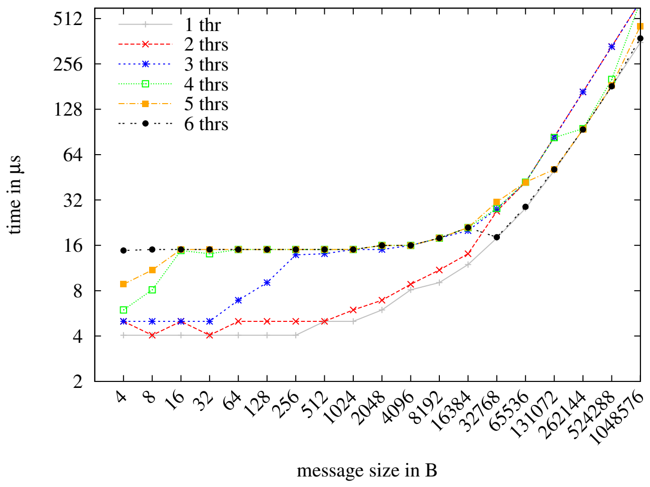


(c) `gaspi_read_notify`

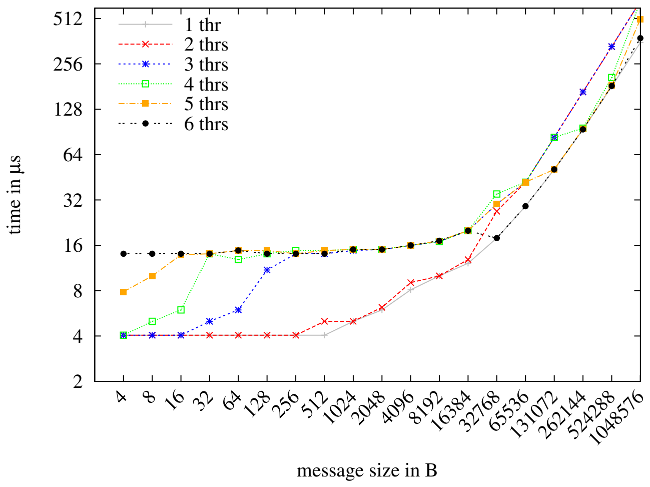
Figure D.2.: Median runtimes for different notified read implementations on Aenigma with threads pinned to the core 0.



(a) NWN

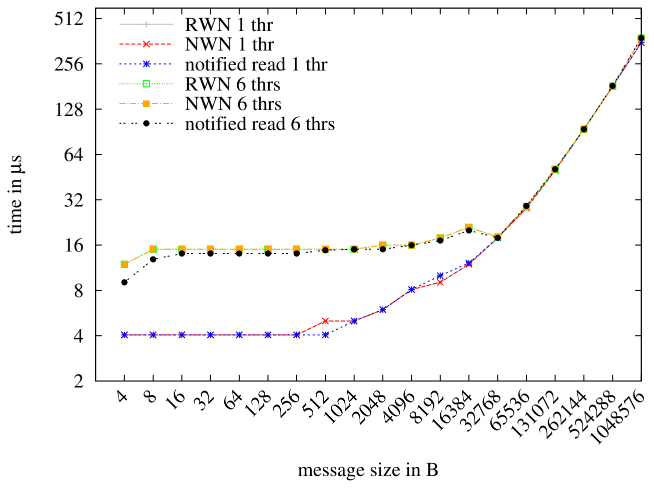


(b) RWN

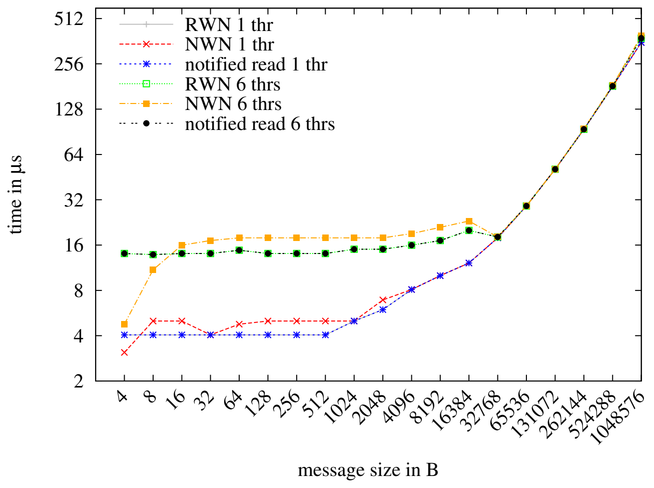


(c) `gaspi_read_notify`

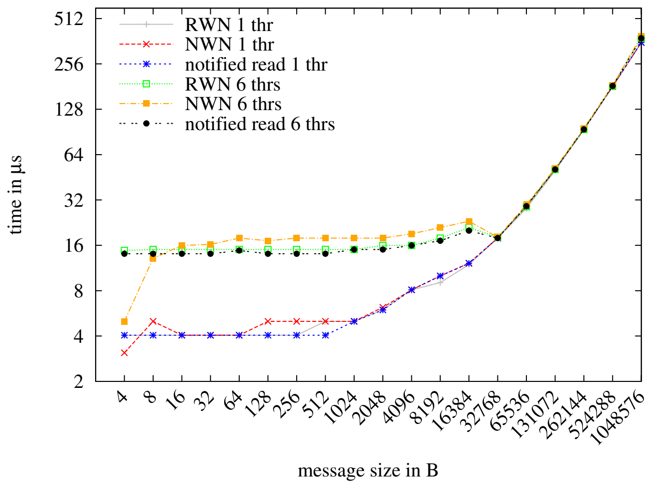
Figure D.3.: Median runtimes for different notified read implementations on Aenigma with threads pinned to the first <number of threads> cores.



(a) unpinned

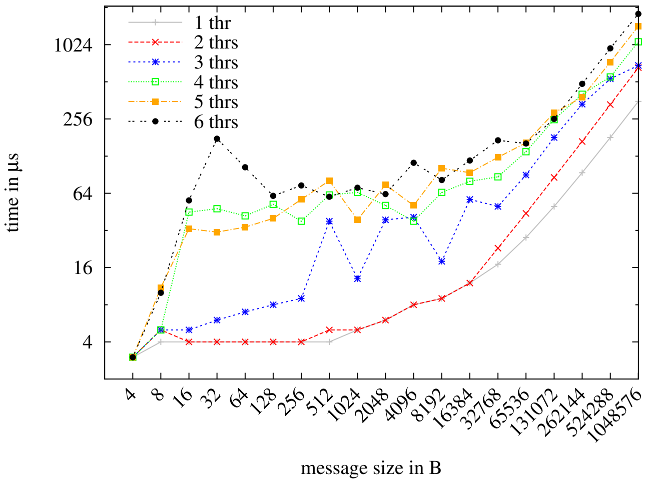


(b) pinned 0

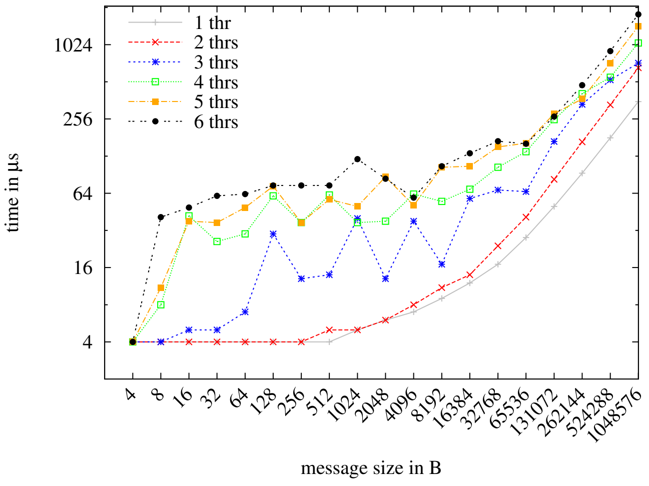


(c) pinned 0-5

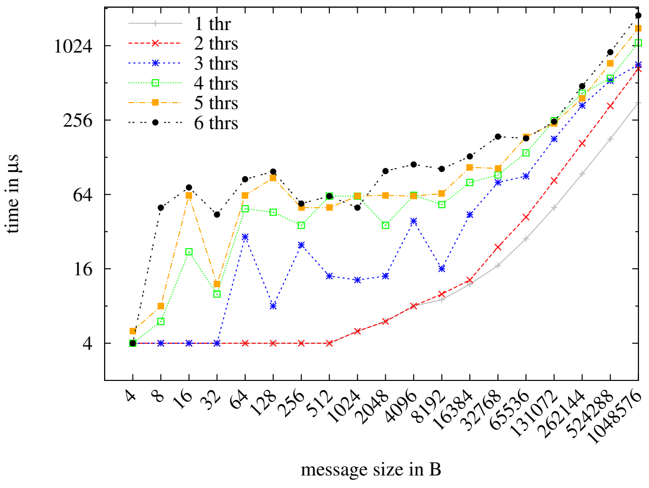
Figure D.4.: Direct comparison of median runtimes for different notified read implementations in the different pinning models on Aenigma.



(a) NWN

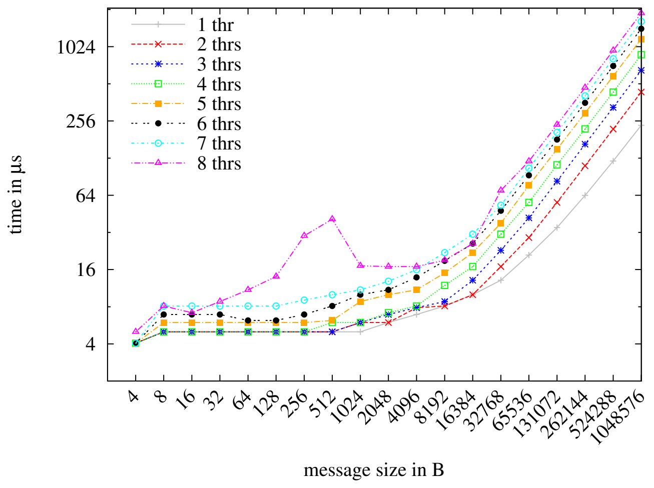


(b) RWN

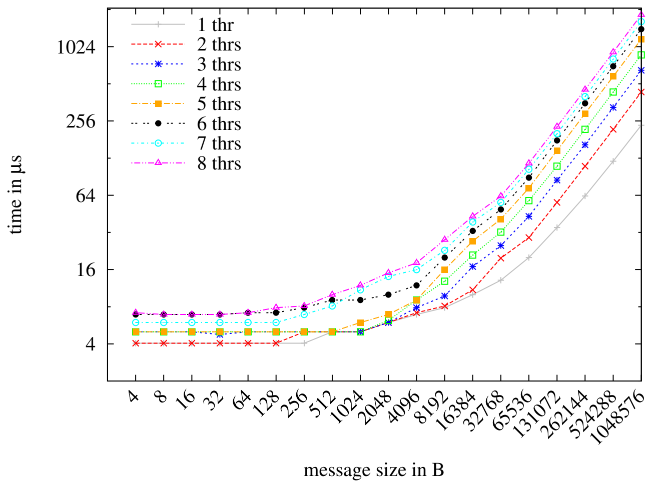


(c) `gaspi_read_notify`

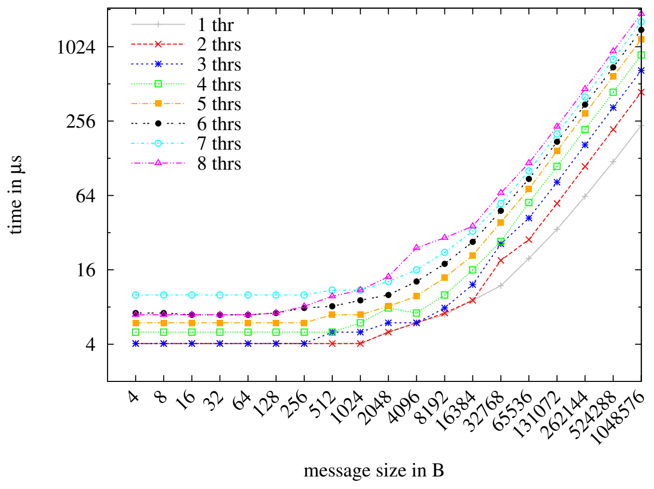
Figure D.5.: Median runtimes for different notified read implementations on Aenigma with the threads pinned to the first <number of threads> evenly numbered cores.



(a) NWN

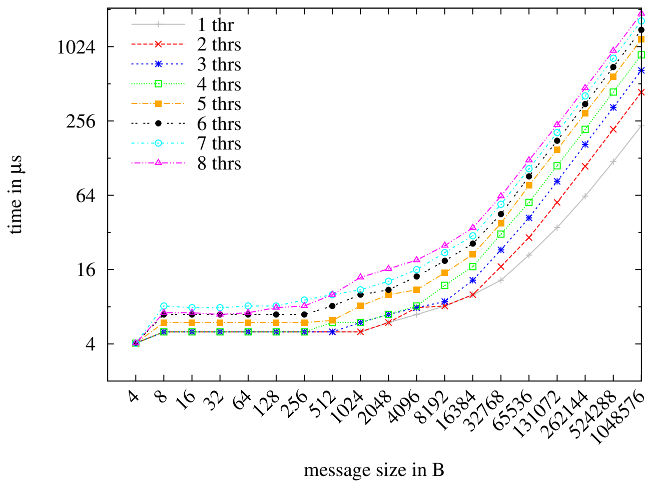


(b) RWN

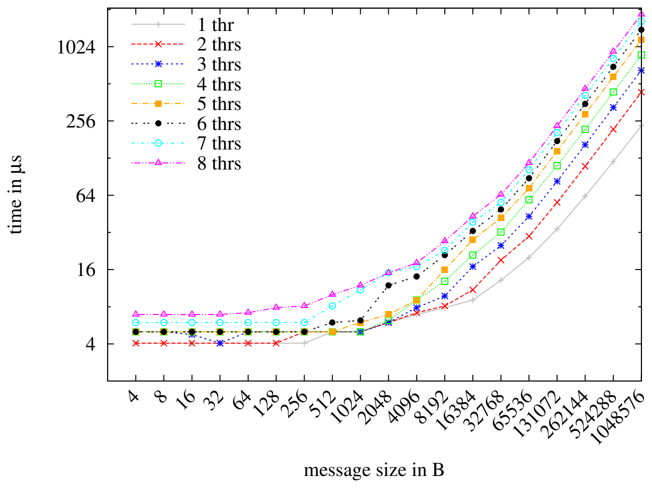


(c) `gaspi_read_notify`

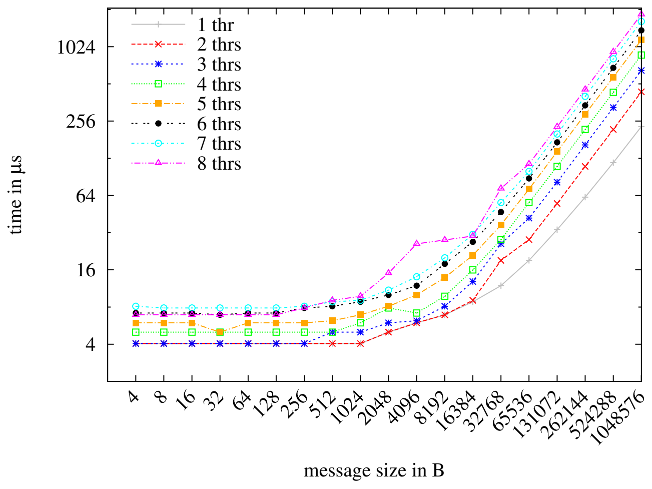
Figure D.6.: Median runtimes for different notified read implementations on MareNostrum III with unpinned threads.



(a) NWN

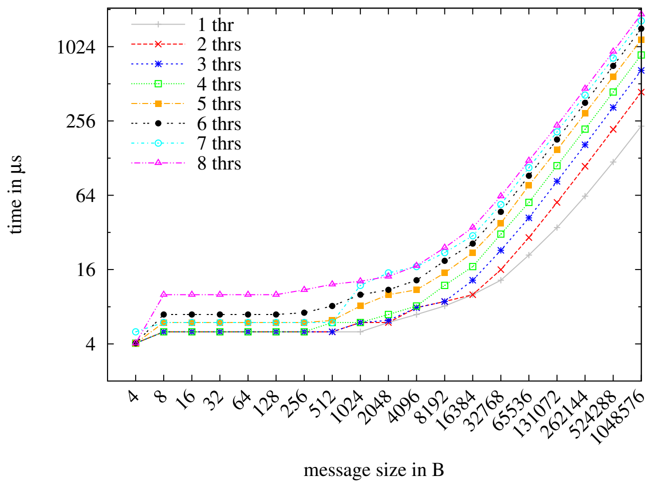


(b) RWN

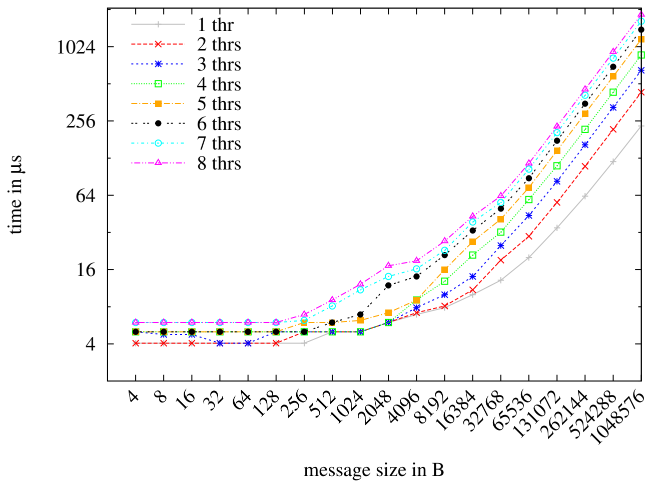


(c) `gaspi_read_notify`

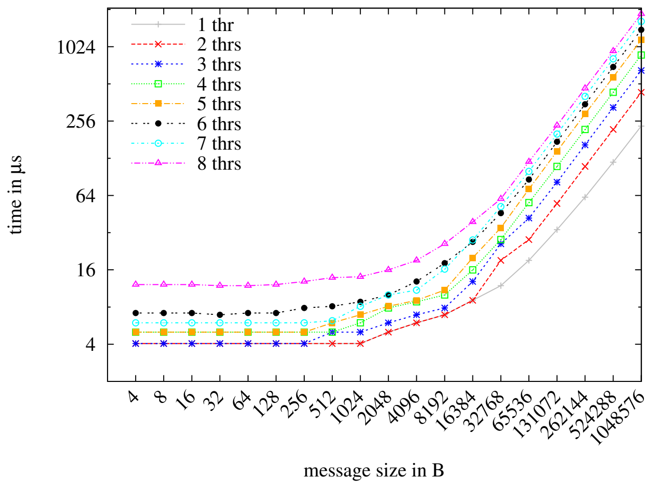
Figure D.7.: Median runtimes for different notified read implementations on MareNostrum III with all threads pinned to core 0.



(a) NWN

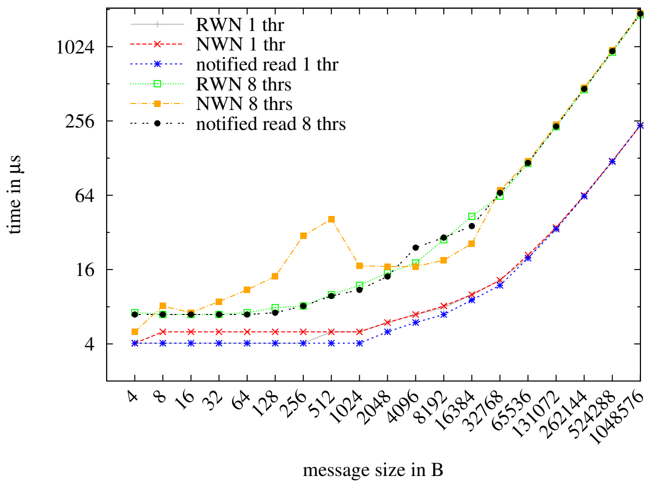


(b) RWN

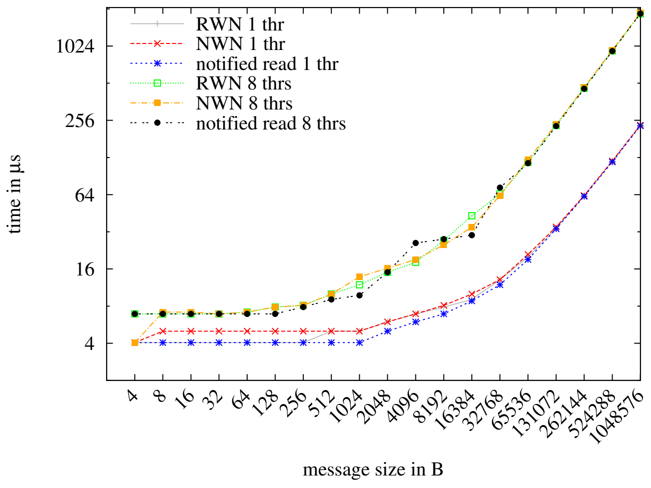


(c) `gaspi_read_notify`

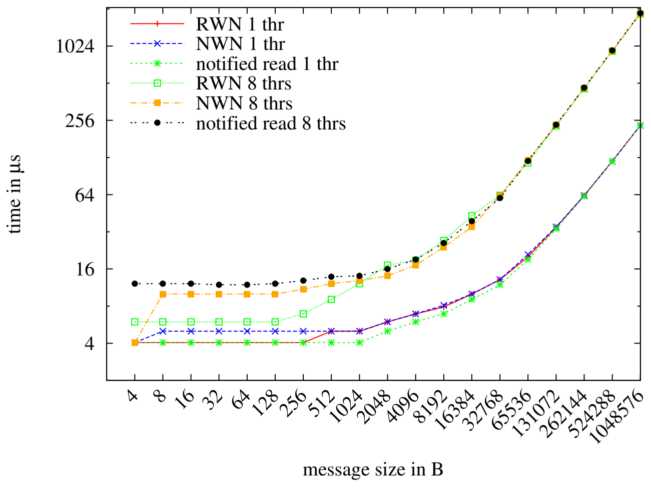
Figure D.8.: Median runtimes for different notified read implementations on MareNostrum III with all threads pinned to the first <number of threads> cores.



(a) unpinned



(b) pinned 0



(c) pinned 0-7

Figure D.9.: Direct comparison of median runtimes for different notified read implementations in the different pinning models on MareNostrum III.

D.3. Pipelined Transpose Code

```
#pragma omp parallel default (none) shared(...)
{
  if(this_is_the_first_thread()){
    for(m = 0; m < num_blocks[local]; m++){
      data_compute(&m);
    }
  }
  else{
    for (k = 0; k < num_initial_reads ; ++k){
      int temp;
      temp = __sync_fetch_and_add(&read_ctr , 1);

      if(temp < nProc -1){
  notified_read(data_available);
      }
      else{
  local_read_switch = 0;
  break;
      }
    }
  }
  while((temp_ctr = get_notification_ctr()) < nProc - 1){
    notify_waitsome(work_id);

    if(ret == GASPI_SUCCESS){
      notify_reset(&work_id , &val);

      if(val == 1){
  increment_notification_ctr();

  if(local_read_switch == 1){
    int temp = __sync_fetch_and_add(&read_ctr , 1);
    if (temp < nProc - 1){

      notified_read(data_available);
    }//end new read
    else{
      local_read_switch = 0;
    }
  }//end read_switch == 1

  for(m = 0; m < num_blocks[first_id]; m++){
    int idx = block_pid_idx[first_id][m];
    data_compute(...);
  }//end non-local block work
```

```
      }//end got a notification (else)
    }//end ret == GASPI_SUCCESS

  }//end while
}//end parallel region
```

Listing D.2: Schematic implementation of the pipelined matrix transpose with notified read.

D.4. Additional Pipelined Transpose Results

In addition to the impact of the matrix size on read-based transpose implementations discussed in Sec. 6.5, also the influence on the write-based implementation was tested. As shown in Fig. D.10, the matrix size has only marginal influence on the transpose rate of the pipelined transpose.

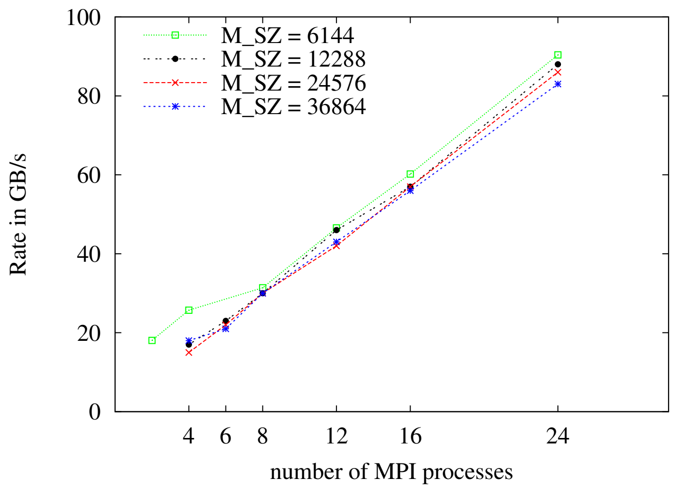


Figure D.10.: Matrix size impact on transpose rates for the `gaspi_write_notify`-based implementation on Aenigma. Started with one process per node and 6 threads per process.