Karlsruher Institut für Technologie

Bachelor thesis

Exploring Aggregation Techniques for Irregular All-To-All Communication

Gregor Peters

Date: 15. Mai 2024

Supervisor: Prof. Dr. Peter Sanders

Advisors: M.Sc. Matthias Schimek
M.Sc. Tim Niklas Uhl

Institute of Theoretical Informatics, Algorithm Engineering
Department of Informatics
Karlsruhe Institute of Technology

Abstract

The structure in many data sets often yields a graph, making graph algorithms an essential foundation for HPC applications. Highly irregular graphs in particular are also becoming increasingly important in this context. This results in irregular communication patterns where a few processes communicate with many others, but typically exchange small amounts of data.

Using traditional communication approaches, the processes communicating with disproportionate many partners are delayed by the numerous send startups required. Even the exchange of meta information, such as the number messages to send, triggers this startup latency overhead. This problem particularly affects small message lengths.

Known solutions reduce startups by introducing indirect message delivery at the cost of increased communication volume. To mitigate this, for specific applications as a distributed array, messages can be further (semantically) combined.

Messages from different origins, but with close targets, are bundled during indirection. Therefore it occurs that messages with the same semantics and target, such as a read request for an element, intersect in a intermediate communication step. These duplicate requests can be aggregated, which reduces the number of total messages sent. It is important to note that corresponding responses must be disaggregated to ensure correct functionality.

This work examines in particular how indirection and aggregation interact and can be combined efficiently. For this purpose, grid-based indirection schemes of different dimensions and characteristics are investigated. Furthermore, different techniques to remove messages with duplicate semantics are explored.

The ideas are combined in an implementation under the abstraction of a distributed array. Evaluations show an obtainable speedup in communication of up to a factor of 20, compared to non aggregating and non indirecting methods, upon a scale of 4096 processes.

Zusammenfassung

Viele Datensätze sind in Form eines Graphen strukturiert, wodurch Graphalgorithmen eine wichtige Grundlage für viele HPC Anwendungen bilden. Dabei gewinnen insbesondere auch hochgradig irreguläre Graphen immer mehr an Bedeutung. Daraus folgen ebenso irreguläre Kommunikationsmuster, bei denen einige wenige Prozesse mit sehr vielen anderen kommunizieren, dabei typischerweise jedoch nur geringe Datenmengen austauschen.

Bei herkömmlichen Kommunikationsansätzen werden die Prozesse, die mit unverhältnismäßig vielen Partnern kommunizieren durch die zahlreichen dafür erforderlichen Sendeaufrufe aufgehalten. Selbst der Austausch von Metainformationen, wie beispielsweise die Anzahl der zu sendenden Nachrichten, löst dieses Problem aus.

Bekannte Ansätze reduzieren Sendeaufrufe, indem sie Nachrichten indirekt zustellen. Dies erfolgt auf Kosten eines höheren Kommunikationsvolumens. Um dem entgegenzuwirken, können Nachrichten für bestimmte Anwendungsfälle, wie ein verteiltes Array, zusätzlich (auf semantischer Ebene) aggregiert werden.

Während der Indirektionen werden Nachrichten aus unterschiedlichen Quellen, aber mit nahegelegenen Zielen gebündelt. Daher kann es vorkommen, dass Nachrichten gleicher Semantik und gleichem Ziel aufeinander treffen. Dies könnten beispielsweise zwei Leseanfragen auf das selbe Element sein. Diese doppelten Anfragen können dann zu einer aggregiert werden, was die Anzahl der insgesamt gesendeten Nachrichten wieder reduziert. Dabei ist zu beachten, dass die entsprechenden Antworten disaggregiert werden müssen, um die korrekte Funktionalität zu gewährleisten.

In dieser Arbeit wird insbesondere untersucht, wie Indirektion und Aggregation zusammenspielen und effizient kombiniert werden können. Zu diesem Zweck werden Gitterbasierte Indirektionsschemata unterschiedlicher Dimensionen und Eigenschaften untersucht. Darüber hinaus werden verschiedene Techniken zum entfernen von Nachrichten mit doppelter Semantik exploriert.

Die Ideen werden unter der Abstraktion eines verteilten Arrays zusammengeführt. Auswertungen zeigen bei 4096 Prozessen eine erreichbare Beschleunigung der Kommunikation von bis zu Faktor 20 im Vergleich zu nicht aggregierenden und direkt zustellenden Methoden.

Acknowledgments

I would like to express my gratitude to my advisors, Tim Niklas Uhl and Matthias Schimek, for their assistance and guidance throughout this thesis. Their prompt responses to my queries were invaluable.

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 15. Mai 2024

Contents

1 Introduction

1.1 Motivation

In High Performance Computing (HPC), applications are executed on distributed systems with thousands or millions of cores. These supercomputers communicate via high-speed networks such as e.g. InfiniBand.

Many applications spend a large part of their execution time on communication in which data is distributed in the network. Regular communication patterns such as all-to-all and stencil exchanges are efficiently supported by current MPI implementations and scale up to a high number of cores. Irregular communication patterns, in which a few cores have a large number of communication partners, which may hinder performance of existing implementations. This is due to the fact, that in many traditional approaches linear startup latencies in the number of processes arise.

This is especially a problem for massively parallel distributed graph algorithms, since it is often necessary to exchange information between nodes of the graph. The structure of large real-world graphs, such as social networks and web graphs, usually leads to irregular sparse communication patterns.

As already shown in [11], [19], it therefore makes sense to group up data with the same destination rather than sending it straight away. Indirect forwarding combines many small startups into larger, but fewer startups with less latencies. This thesis particularly focuses on how messages can be aggregated on a semantic level.

A distributed array, where each process holds a continuous section of data, is a good abstraction of communication for many applications. Read and write requests can be submitted to the array, which often only contain a small amount of data of individual processes, but require communication with many different participants. The indexing of the array allows identical requests with the same target to be identified and efficiently aggregated. Semantic aggregation in distributed arrays means that, in the case of write requests to the same index, only one of the two requests will prevail and others will be overwritten. In the case of read requests, an index does not have to be requested multiple times. It is sufficient for each index to be requested only once and the response to be disaggregated again to the corresponding recipients.

1.2 Contribution

This thesis investigates the effects of combining the startups to save latency, together with the resulting possibility of semantic aggregation, i.e. filtering read requests targeting the same element. Potential responses to the messages must be disaggregated accordingly, to reach their requester.

In particular, different indirect routing schemes are considered and several approaches to aggregate duplicate messages are explored. This shows that semantic aggregation can be combined with message routing and therefore does not cause any major overhead.

The approach is implemented under the abstraction of a distributed array and evaluated in various experiments. The combination of these techniques results in a communication speedup of almost up to 20 on realistic use cases, compared to approaches without indirection and aggregation.

Moreover, a general overview of all-to-all exchanges in context of irregular communication is given.

1.3 Structure of Thesis

First, some basic definitions are made and a brief overview of technologies for parallel data exchange is given.

Then in section 3, we take a closer look to all-to-all communication algorithms and irregular variants. We dive into indirection techniques to combine messages and the theoretical approach of Ranade et al., which provides flow control, aggregation and theoretical runtime guarantees. However, it is unclear how this approach can be efficiently implemented in practice.

Subsequently, in section 4 a number of simpler and more practical approaches to aggregation and disaggregation are explored, which are then evaluated in section 5.

Finally, the results are summarized and an outlook is given.

2 Fundamentals

2.1 Alpha-Beta Model

The alpha-beta model is a basic model for calculating delays in networks. Alpha defines the latency, i.e. the time it takes to establish a connection. Beta specifies the transmission time of a data word. The cost of transmitting a message of size n is $\alpha + \beta n$. Alpha is typically greater than beta and dominates the time required when n is small. In order to reduce latencies, messages can be bundled in intermediate hops. Although this increases the total amount of data sent, experiments have shown a speedup for small message sizes [11] [19].

In the context of supercomputers, this model is only suitable for certain scopes, as the infrastructure levels have different bandwidths and latencies. It must also be noted that not just one process sends its data, but many at the same time. Since not every process usually has a point-to-point connection to every other process, congestion can arise in the network, which is also not captured in the model.

2.2 Existing Technologies

2.2.1 RDMA

Remote Direct Memory Access offers the possibility of one-sided access to memory without involving the remote process, but also offers two-sided communication. As the kernel is bypassed in one-sided communication, lower latency and higher throughput can be provided. This is the reason why RDMA often forms the basis for other technologies, such as one-sided communication in common MPI implementations.

RDMA communication is based on queues that are processed or filled asynchronously. There are different queues:

Work queues, such as the send and receive queue, contain work requests. A program submits work requests, which are jobs that the hardware should fulfill, for example writing to a memory location or waiting for an incoming message.

Then there is the completion queue, which holds information about completed requests. When a work request is completed, it can create a work completion object, which notifies the remote process about the action, in case of one-sided communication.

2.2.2 MPI

The Message Passing Interface[2] is widely used to enable the communication of programs with many parallel processes. Communication in MPI takes place via communicators that define the context of the communication. New communicators can be split off from the global `MPI_COMM_WORLD` or created from a group. A group is a set of processes to which no context is assigned. The creation of communicators is associated with an overhead, as the processes in the communicator must agree on a common context. This overhead tends to be amortized as communication on large communicators is more expensive than on small ones. By splitting communicators, optimizations can be made as the exchange of meta information between processes that do not communicate can be saved.

MPI includes both point-to-point communication and collective operations. We are particularly interested in the collective operation `MPI_Alltoall`, respectively the irrgeular variant `MPI_Alltoallv`. Here, each process sends data to every other process. In the irregular case, the amount of data can vary, or only certain processes may be targeted. Communication patterns can arise in which individual processes receive a large number of identical messages. This is where this work comes in to solve this problem by reducing duplicate messages.

One-sided communication and file system access are also supported. With one-sided communication, a process can, for example, open a memory window that can be accessed by other processes without the host process being involved. In common MPI implementations, this is realized internally by RDMA.

MPI also offers the option of structuring communication virtual topologies on which neighborhood collectives can be executed. Blocking and non-blocking barriers are also provided to synchronize processes.

2.2.3 PGAS

Partitioned global address space is an alternative paradigm for parallel programming. Each process has its own address space, which is part of the global partitioned address space. Memory accesses to the global address space are more expensive than to the local address space. One popular application of the paradigm is Unified Parallel C++ (UPC++). UPC++ provides remote memory access (RMA), which is realized as far as possible by RDMA, as well as remote procedure calls (RPC). RMAs and RPCs return so-called futures. One can wait for the completion of a future and read out the value, or combine them into new futures. RMAs run completely asynchronously and there are no guarantees as to the order in which requests arrive and are processed. This results in a very simple and powerful programming model in which synchronization is left to the user. Due to the fact that the RMAs run asynchronously, mechanisms such as indirection and aggregation are challanging to realize, as there are no logical global communication steps. Therefore, it is unclear in which cycle indirection and aggregation should take place. This problem is solved by Steil et al. [18]

using mailboxes that are emptied after a certain threshold. Due to the asynchronicity, it is not necessary to wait for slow processes reaching the communication context later. The local calculation can be continued as soon as the process has sent and received all messages.

Another example of a PGAS implementation is global arrays[16], or the RDMA based DArray[4]. The idea of globally distributed arrays is also taken up in this work, as it provides a decent interface.

3 Related Work

In this chapter an overview of different algorithms for all-to-all exchanges and irregular variants is given. To achieve this, we will examine which algorithms are used in the common MPI implementations MPICH and OpenMPI. We will then explore the problems in the interface of global all-to-all operations that have led to the development of virtual topologies in MPI.

Following this, we consider why indirect communication can be useful and various indirect communication patterns are shown.

Finally, the theoretical model of Ranade et al. for emulating a PRAM is outlined, which has favorable properties such as flow control and theoretical runtime guarantees, but cannot be implemented efficiently in practice without further investigation.

3.1 All-to-all Algorithms

All-to-all exchanges are a frequently occurring communication pattern in practice. There are different variants in which data is potentially exchanged between all processes.

In this section, we look at common algorithms for all-to-all exchanges.

For further understanding, Netterville et al. provide a visual overview of the all-to-all algorithms used in common MPI implementations such as MPICH and OpenMPI [14]. Depending on the number of processes and the size of the messages, different algorithms are used to realize the data exchange.

First algorithm is Bruck's algorithm. Bruck's algorithm is typically used for the efficient implementation of all-to-all exchanges with small message lengths. The key idea behind it algorithm is the reduction of startup latencies by combining messages and forwarding them indirectly. However, this comes to the cost of an increased overall transmission load.

The other two algorithms are the spread out algorithm and the pairwise algorithm. They send the data directly to the target process and are therefore better suited to large numbers of transmissions where latency is less of an issue. The difference between the two algorithms is that the pairwise algorithm uses blocking send calls, while the spread out algorithm uses non-blocking calls. For messages with a size of 32KB or more, the non-blocking pairwise algorithm is used, otherwise the non-blocking spread-out algorithm is used. If there are more than 8 processes and the messages are smaller than 256 bytes, MPICH uses the brucks algorithm; with fewer processes, the overhead due to the startups is too low.

3.1.1 Bruck's Algorithm

Bruck's algorithm is a store and forward algorithm in which data with the same destination is forwarded in increasingly larger blocks with the same final destination. The indirect hops allow each process to only need $\mathcal{O}(\log p)$ sends to another process so that all messages reach their destination. The indirect forwarding increases the total transmission load by $\mathcal{O}(\log p)$, but also reduces the latency through only $\mathcal{O}(\log p)$ startups. The overall cost, according to the alpha-beta model is $\mathcal{O}(\log p(\alpha + \beta n))$ The algorithm is divided into following three phases:

- The data is rotated by the rank of the process. At position i of the process with the rank $r \in [0, R-1]$ the message for the process with the rank $i+r \mod R$ is located.
- In this phase, the data is exchanged in $\log(R)$ steps. In step s, each process sends the data to the process $r + 2^i \mod R$ whose index has a binary 1 in bit s, i.e. $I = \{x \in [0, R-1] \mid x \& 2^s \neq 0\}$. The received data is written to the indices that have just been sent. This creates coherent chains of messages with the same destination, which double in length at each step.
- When the previous phase is complete, each process already holds the messages intended for it, but in an incorrect order. To restore the original sequence, another shift must be performed: The message from processor i is located at position $r - i(+R) \mod R$.

An illustration of the algorithm can be seen in 3.1. If the algorithm is slightly adapted, the last sorting phase can be avoided.

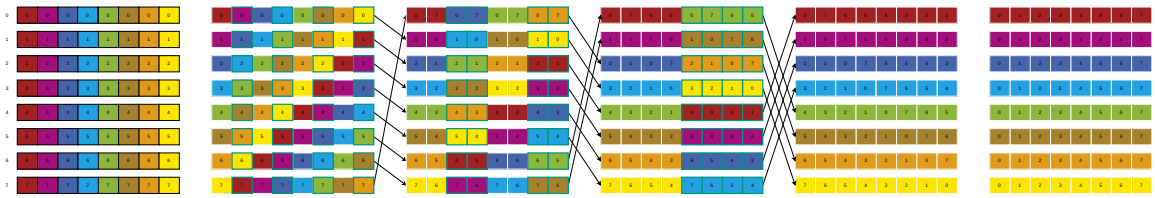


Figure 3.1: Standard Brucks algorithm on 8 cores. The figure is based on [14]

Non Uniform Bruck

Ke Fan et al. present two apporaches to extend Bruck's algorithm for irregular all-to-allv exchanges with different transmission sizes [5].

In the first approach, all messages are padded to the size of the largest message. This approach is very inefficient for large differences between the messages.

The other variant is the so-called Two-phase Bruck. Before each step, the sizes of the messages are transferred. Then, according data follows. Experiments have shown that the Two-phase Bruck performs better than the standard MPI implementation, for small average

message sizes. With a large number of processors, this effect diminishes again. With 32k processors, the messages must be smaller than 128 bytes, in order to achieve a speedup.

3.1.2 Spread out

The spread out algorithm consists of $p-1$ rounds in which each process sends and receives a message from two different processes. It is important that not all processes send the messages in the order of the ranks of the recipients, otherwise congestion would occur. Due to the non-blocking calls, the rounds can overlap, but the messages are still distributed, such that a bottleneck will not form immediately. At the end, the system waits for all messages to have reached their destination. This leads to a performance of $\mathcal{O}(p\alpha + \beta n)$.

Algorithm 1: Spread out

```
1 r ← process rank id
2 for i ∈ [0, P − 1] do
3 |   src ← (r + i)%P
4 |   r receives data-block (src, r) from src
5 for i ∈ [0, P − 1] do
6 |   rcv ← (r − i + P)%P
7 |   r sends data-block (r, rcv) to rcv
8 Wait for all requests to be completed
```

There are approaches to limiting the number of requests in circulation. This also limits the maximum bottleneck. Pseudocode for the spread out algorithm can be seen in algorithm 1.

3.1.3 Pairwise

The pairwise algorithm works in the same way as the spread-out algorithm, only with blocking send calls. If the number of processes is a power of 2, the process with the rank $r \oplus i$ can be used as both the sending and receiving partner. The reason for switching from non-blocking to blocking communication is not described in literature. One possible explanation could be that large messages make full use of the bandwidth, which is why the advantage of non-blocking communication cannot be utilised.

3.2 Indirection Patterns

In the last section it was demonstrated that indirect transmission of messages entails fewer send startups per process, which is utilised by Bruck's algorithm. Bruck's algorithm is very similar to a hypercube routing, but there are many other indirection patterns to consider.

The choice of the routing scheme has a significant impact on the performance of the communication, as it involves a trade-off between the number of indirect steps and the number of communication partners per step, which is the number of required start-ups.

As already described, indirect hops, in which messages are bundled, help to minimize latencies, but come to the cost of increased communication volume. This offers potential to further aggregate the messages, particularly on a semantic level as is being utilized by Ranade et al.[17].

They fokus primarily on the butterfly network, on which we will have a closer look later in section 3.4. On $2^n(n+1)$ nodes, $3n$ forwardings are required until the message reaches its destination.

Laxmikant V. Kalé et al. present in [11] a way to structure arbitrary numbers of processes in 2 and 3D grids, as well as in a hypercube. In a grid topology of dimension k there is a side length d_k for each dimension. This results in dimension-wise coordinates for each process. Communication traverses the dimensions in phases so that messages end up in the correct target dimension at each step. Grids with a side length of 2 are called hypercubes.

Hypercubes are also similar to butterfly networks in structure. If the layers of the butterfly are combined in one node, this leads to a hypercube. As a result, there is only one communication partner per transmission step, and a better ratio of indirection steps to the number of nodes. In each dimensional communication step of the hypercube, the destionation region halves in size and the number of messages that have reached their final destination doubles on average. This property will later be exploited to combine identical messages.

The paper experimentally confirms that for small message sizes, reducing the number of startups through indirection accelerates performance.

In addition, a possibility is presented to deal with the case that the number of processes cannot be mapped by the topology and holes arise. In the case of the 2-D grid, each edge that would lead to a hole is distributed evenly among the corresponding remaining nodes in the target dimension. A visualization of an incomplete 2-D grid with holes is shown in figure 3.2.

In the 3-D case, the holes are mapped to the corresponding nodes in the lowest level. As holes are always located in the top level and the exchange between the levels only takes place last, this ensures that every message reaches its destination. However, the load is not optimally distributed, as the messages could also be distributed between the completely filled levels.

If the number of processes in the hypercube does not correspond to a power of two, the next smaller power of two is selected and a hypercube is formed. Fewer nodes remain than there exist in the hypercube. Therefore, each remaining node can be assigned a node in the hypercube by subtracting the size of the hypercube from its rank. Messages for the processes outside of the hypercube are automatically forwarded to the corresponding representative by the standard hypercube routing. At the start of the communication step, each node outside the hypercube sends its messages to its partner, and once the exchange

in the hypercube is complete, the partner sends the messages back to it. As a result, there are only 2 send calls overhead to the ordinary hypercube. An an incomplete hypercube can also be seen in figure 3.2.

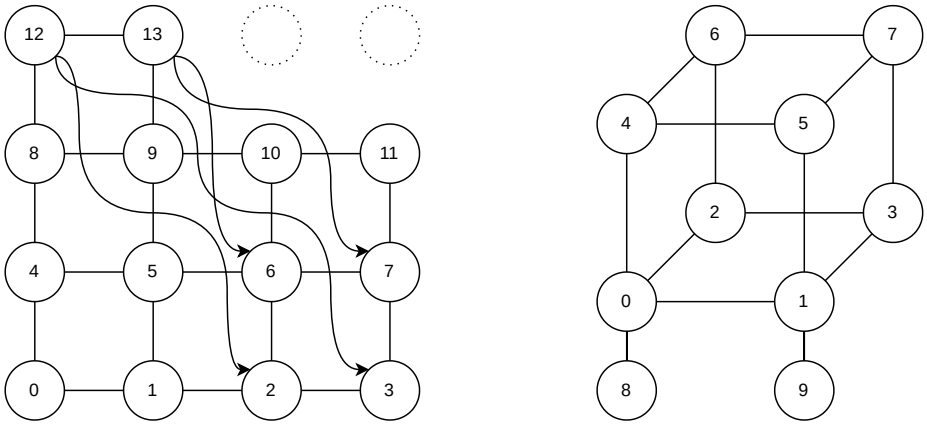


Figure 3.2: 2D grid and 3D hypercube with holes according to [11]

3.2.1 Locality-aware Indirection

Supercomputers are complex systems that group and connect underlying subsystems on several levels. This results in different regions such as processors, sockets or islands. Communication between different regions causes higher latencies than communication between processes closer to each other within the same region. Since the latency increases when communicating between regions, it may be useful to bundle the messages in the same region and then distribute them there.

Collom et al. suggest messages with the same target region to first be gathered within one region and then sent as a single large message to the destination region [3]. The approach from the paper consists of four communication phases:

- First, the data that remains within a region is exchanged. This exchange is called intra-region communication.
- Next, the data that leaves the region is redistributed within the region so that all messages with the same target region are on the same process.
- Each process exchanges its data with the corresponding process from the regions assigned to it. This is called inter-region communication.
- In the last step, the inter-region messages are distributed throughout their destination region.

  Note: This step could be combined with the first step, as in Node-Local Node-Remote routing proposed by Steil et al. in YGM [18], to further reduce latencies.

Experiments show that setting up locality aware persistent neighborhood communication generates an overhead that is reduced after about 20 iterations. The paper describes an extension to remove duplicates in inter-region communication, but does not discuss the technique used for this. Removal of duplicates turned out to be advantageous, but no further research was carried out in this regard.

Other approaches skip the second step and only send the messages bundled within a process to another region [7, 18].

This communication pattern does not allow the communicators to be split up and a global all-to-allv call also entails problems, which we will look at in more detail in 3.3.

3.2.2 Multidimensional grids vs. locality aware

The inter-region communication phase in the locality aware approach is determined by two parameters: the number of regions R and the number of processes per region $\frac{P}{R}$, which is usually specified by the hardware. The ratio, $\lceil \frac{R^2}{P} \rceil$, determines the number of other regions a process communicates with.

In a k-dimensional grid, inter-region communication will continue until the step size has decreased until being smaller than the size of a region. For the sake of simplicity, we assume that the dimension sizes of the grid are aligned with the size of a region. For a k-dimensional grid with dimension size d, the last $\log_d \frac{P}{R}$ steps are omitted on intra-region communication. Therefore, $k - \log_d \frac{P}{R}$ steps are necessary at inter-node level. For the hypercube, these are $\log_2(R)$ steps. Each of these steps contains $d - 1$ startups. The grid therefore benefits from small region sizes and a higher amount of regions.

3.3 Sparse Communication

Sparse all-to-all exchanges are an irregular variant in which processes only communicate with a subset of the others. A distinction is made between static (SSDE) and dynamic sparse data exchanges (DSDE). With SSDE, each process knows the consistent set of its communication partners, whereas for DSDE it is unknown and might change at runtime.

A general problem with sparse irregular all-to-all exchanges is that common interfaces, such as `MPI_Alltoallv`, require metadata, such as the number of messages sent and received, from all processes in the communicator. This metadata has also to be exchanged between processes without any communication taking place.

Hoefler and Träff [10] have presented a neighborhood interface as a solution, in which a fixed set of communication partners can be defined.

This led to the development of virtual topologies in MPI-3.0.

To solve the DSDE problem, Hoefler et al. introduce the $\mathcal{NBX}$[9] algorithm, shown in algorithm 2. By combining non-blocking barriers and send with receive operations, data

exchange is achieved.

Algorithm 2: $\mathcal{NBX}$ Algorithm for dynamic sparse data exchange from [9]

```
Input: List I of destinations and data
Output: List O of received data and sources
1  done ← false
2  barr_act ← false
3  foreach i ∈ I do
4      start nonblocking synchronous send to process dest(i)
5  while not done do
6      msg ← nonblocking probe for incoming message
7      if msg found then
8          allocate buffer, receive message, add buffer to O
9      if barr_act then
10         comp ← test barrier for completion
11         if comp then done ← true
12     else if all sends are finished then
13         start nonblocking barrier
14         barr_act←true
```

3.4 Emulating Shared Memory

In 1991, Ranade et al. [17] had the idea of emulating a CRCW PRAM for p processes on a butterfly network, only using constant sized queues and avoiding contention with a high probability. In contrast to other approaches, it guarantees a theoretical PRAM step runtime in $\mathcal{O}(\log p)$, but each message is sent separately, which in practice leads to high latency costs in practice. Further research is needed to find out how messages can be efficiently combined in this approach. This work therefore focuses on approaches that are more suitable in practice.

Existing practical approaches, such as the DArray [4], also examine distributed cache behavior.

Routing on butterfly networks is not trivial. Ranade has successfully combined flow control with the aggregation of messages using index-priority-based scheduling.

First, we take a closer look at the butterfly network and then go into the details of this approach.

3.4.1 The Butterfly Network

The processes are arranged in the virtual topology of a butterfly network. A butterfly network is divided into layers. Each of the $n + 1$ layers comprises 2^n processes. A process is represented by its coordinates (l, p) consisting of the layer l and the position in the layer p. Each process, or node in the network graph (l, p) is connected to $(l+1, p)$ and $(l+1, p \oplus 2^l)$ from the next layer, for $l \in [0, n]$. Except for the first and last layer, each node has four communication partners. An example for a butterfly network is shown in 3.3.

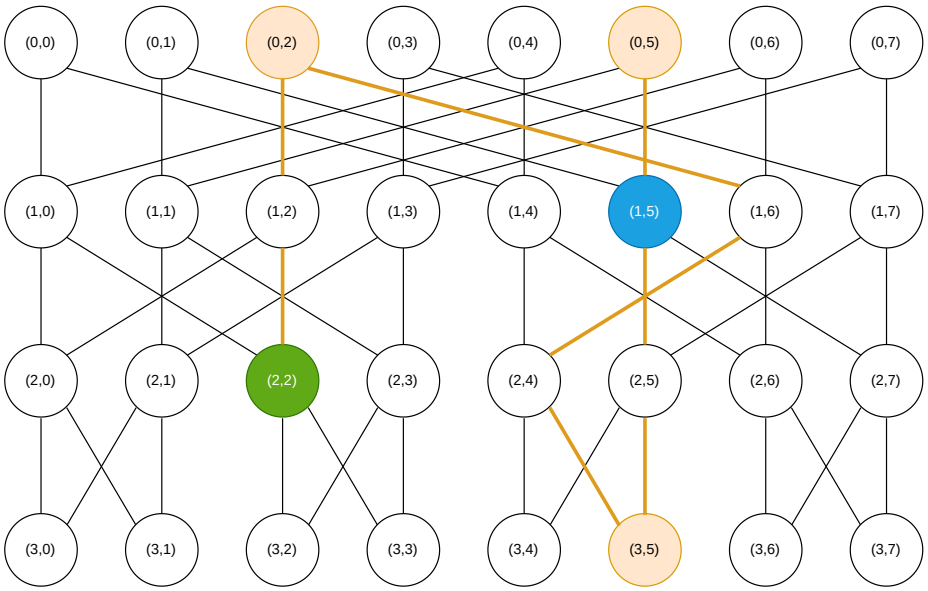


Figure 3.3: 32 node butterfly network with message path from [17]. (2,2) sends a message to (1,5). The orange highlighted nodes are the end points of the three phases

3.4.2 Message Structure

Ranade categorizes messages into three different types: request, ghost or EOS. EOS messages are sent at the end of a communication step to determine that all messages have reached their recipients. Additionally, a message holds a tag, which consists of the destination node concatenated with the memory address. This tag is later used to prioritize the messages and determine the path through the network, with EOS messages having the lowest priority. The mapping from the memory address x to the destination node is defined by the function $g(x)$. Finally, a message contains the data.

3.4.3 Message Path

To ensure that each message arrives at its destination, Ranade describes a path through the butterfly network. Consider a message issued by processor (l, p) for PRAM location x stored in memory module $h(x) = (l', p')$. The messages are forwarded in 6 steps:

- In the first step, the message is forwarded backwards to the start of the network, to node $(0, p)$.
- Then the message is forwarded to the correct layer position (n, p'). To find out which of the two forward edges must be taken, look at the tag bit at the $n - i$ last position in layer i.
- It traverses the network backwards until the node (l', p') where the data is located. The response is generated here and continues its way backwards to $(0, p')$.
  Note: if the response would immediately return to its sender, this would destroy their order.
- In the last 3 phases, the messages are routed in reverse direction to the original sender via reply routing.

Figure 3.3 illustrates the message path using an example and may therefore contribute to understanding.

The phase only applies to each individual message and is not synchronized globally. As described in the following section, messages with a small tag are preferred. This can lead to sequences of long "polarized" messages. However, this is as shown by Ranade not very likely.

3.4.4 Scheduling

Each node holds a queue of a fixed size for each incoming edges. There is another queue containing the messages dispatched by the node, sorted by tag. At the end of this initial queue there is always an EOS message.

If none of the edge queues is empty, the message with the lowest tag of the two queues is selected for dispatch. Before sending, it must be ensured that the recipient has sufficient capacity in their queue.

If one of the edge queues is empty, the system waits until it contains at least one element again. Since the nodes from the last layer have no incoming connections in the first step, they fill the queues of the nodes above them and the messages propagate through the network.

When two heads of the queues have the same tag, the messages reach the same destination and can be aggregated. Whenever a message is sent, an according ghost message with the same tag but without data is sent to the other node.

Ghost messages implicitly maintain the order, as the tag of a received ghost message is a lower bound for each subsequent message tag. This is a method of avoiding congestion

in particular, as received ghost messages are deleted at the end of each step unless they are sent further. This ensures that there is a free path through the network for potential messages with a small tag so that no congestion occurs.

3.4.5 Reply routing

Each node in the network has a FIFO queue for each edge queue pair. In phases 1-3, it is written to this queue whether the sent message originates from the upper or lower edge queue or has been aggregated from both.

In phases 4-6, the messages must return to the sender. As the sorting of the reply messages is retained, the head element of the queue can be used to decide to which node the message must be forwarded. The fact that all messages with the destination (l', p') intersect their path at the latest at node (n, p'), ensures that no message with duplicate targets arrive at the recipient (unless the recipient is (n, p')).

4 Engineering Distributed Array

Considering the implementation of a PRAM on top of distributed memory, a distributed array provides a useful abstraction. In a distributed array, each process holds a subset of the data. A distributed array of size l, which is distributed over p processes, is divided into consecutive sections of length $\frac{l}{p}$, which are held locally by the processes. The process with the rank r holds the elements with the indices $\left[r\frac{l}{p}, (r+1)\frac{l}{p}\right)$.

If a process wants to access the data, it has to submit read or write requests.

To be able to access the data on other processes, these requests are then passed on to the corresponding process on which the data is located, for example by message passing. As explained in previous chapters, indirect forwarding can save latency costs, as the data is forwarded collectively and start-ups are minimized. However, since the data does not reach its destination directly, the proportion of bandwidth-dependent costs increases. To counteract this effect and reduce the overall transfered data, the following observation can be exploited:

During the indirections, different messages with the same group of destination ranks accumulate. If there are duplicate requests with the same destination index in a PRAM step, they can encounter before they reach their destination. The two requests can then be aggregated to one, which saves bandwidth costs. This process is called *aggregation*.

In the following, we will explore different aggregation techniques and whether the additional local work would be worthwhile.

In order to achieve this, we will first get a more detailed overview of the procedures in the distributed array. Next, in section 4.2 is explained, how the data is indirectly routed through the network. Then we will look at how the routing information for each message can be stored so that each message reaches its destination. Section 4.4 explores different ways to aggregate duplicate messages. Once the requests have retrieved their data, their responses must be returned to their origin. In section 4.5 we examine how to reverse the aggregation so that each response reaches its source. The process of rolling aggregation back is called *disaggregation*.

4.1 Overview

The first question when implementing a distributed array is the distribution of the entries to the processes. To keep the mapping as simple as possible, the indices are distributed to

the processes in consecutive blocks, as already described. The question arises as to how the distributed array can be accessed. For this purpose, the following interface is available: `distributed_array.read(index, variable)` submits a read request to the array, where the value at the index is written to the variable.
`distributed_array.write(index, value)` offers write the counterpart. The requests are not executed directly after being created, but are triggered by
`distributed_array.globally_synchronized_lock_step()`. The lock steps define the time frame in which messages are combined and aggregated.

To prevent conflicts, the write requests are processed strictly before the read requests to ensure that read requests always deliver the latest value.

4.1.1 Processing of Write Requests

Initially, the process holds an array with all the write requests it has made. Indices may already be written twice at local level.

In order to aggregate these duplicate requests, it must be decided which of the two values will be overwritten. For this purpose, an overwrite strategy can be defined when the array is created.

The local removal of requests with the same index is called *local aggregation*, as only local requests are involved.

After local aggregation, the messages are forwarded according to the chosen routing scheme until they arrive at their final destination. On the intermediate hops, duplicate indices may occur again. However, this can only occur between requests that were received by different processes in this sending step, as they have already been aggregated within the sending process, in the previous step. If these requests are combined, this is referred to as *global aggregation*, as requests from all processes are aggregated.

After aggregation phase, outgoing messages need to be forwarded to their next hop, which is another component of local work. The indices must be placed in the correct position in the send buffer. Logically, there is a separate send buffer for each communication partner, but MPI requires them consecutively in a large send buffer.

For this purpose, let $\mathrm{destination}\colon \mathrm{Messages} \to \mathrm{Processes}$ the function mapping from a message m to its destination process, a routing function
$\mathrm{route}\colon \mathrm{Processes} \times \mathrm{IndirectionStages} \to \mathrm{Processes}$, which determines the next hop in the current indirection stage dim for a destination. The message is to be written to the corresponding send buffer $\mathrm{send\_buffers}[\mathrm{route}(\mathrm{destination}(m), \mathrm{dim})]$.

In the context of MPI this means ordering messages in one global contiguous send buffer, which is split into the single send buffers $S[i]$ by the metadata arrays `send_counts` und `send_displacements`. The naive approach, is to use bucket sort, in which the buckets are the next recipients. Some aggregation techniques allow more efficient send buffer construction, by interleaving duplicate detection and bucket construction. The consecutive

mapping of the indices to the processes in relation with the grid routing scheme also allows sorting by index in order to obtain a correct send buffer, as explained in more detail in section 4.2.

Once the write requests have reached their final destination, their position in the memory is calculated from the global index and the corresponding value is written.

Once all write requests have been processed, the read phase follows.

4.1.2 Processing of Read Requests

The processing of read requests consists of two parts. First, the requests need to be sent to the processes that hold the requested data. These then create responses and send them back to the requesting process.

As with write requests, read requests are first aggregated locally. A simple elimination of duplicate indices is sufficient for this, as read requests targeting the same index are identical. It should be noted, that the requests which are combined have to be persisted, so that aggregation can be reversed upon receipt of the responses.

The variable to which the response is to be written is only required by the process making the request and therefore does not need to be sent. The payload of a read request packet is therefore only the array index.

After local aggregation, the requests are forwarded again according to the routing scheme and aggregated globally until they have reached their destination. The aggregation techniques used for this are explained in more detail in section 4.4.

Once they reach their destination, the indices are resolved to local memory addresses, the corresponding data is read and stored in response messages. In addition to the retrieved data, these also consist of the indices to enable identification of the content.

Unlike the write requests, we are not yet finished here, as the responses have to reach their requesters again. There are different approaches to store the routing information, as explained in section 4.3. Either the information is stored explicitly in the request, or the routing of the responses is combined with the disaggregation.

To ensure that the processes whose requests have been aggregated also receive their data, these must be disaggregated. Various disaggregation techniques are described in section 4.5. Important here is that the responses follow the path described by the routing scheme in reverse order. Only this ensures all aggregations are rolled back, as only the processes contain the necessary information about where the aggregation took place. The responses must, again, be written to the correct position in the send buffer during the return process so that they reach the correct subsequent process. Disaggregation during indirection is referred to as *global disaggregation*. The routing scheme therefore has a significant influence on the performance of the distributed array. We will therefore take a closer look at this in the following chapter.

Once the responses have reached the requesting process, it still has to assign the data to its requests by writing the data to the corresponding memory locations. This process is called *local disaggregation*.

4.2 Routing Scheme

The choice of the routing scheme affects the amount of indirection steps the messages need to reach the destination and which messages with the same destination are aggregated. It can be observed that an increase in the number of intermediaries leads to a reduction in the number of communication partners and startups. Aggregation benefits from larger exchanges at an early stage, as early aggregated messages do not have to be forwarded anymore and overall messages sent are reduced. Conversely, using grid routing schemes, latencies decrease in late indirect stages, as communicating processes become physically closer.

This thesis mainly focuses on k-dimensional grids, which include the hypercube, but also 2-D grids. A grid routing scheme is determined by the number of intermediaries, which is also referred to as the number of dimensions and the shape of the respective exchanges. A $2 \times 2 \times 4$ grid is shown in fig. 4.1.

Processes are identified in a k-dimensional grid by their coordinates $(d_0, \ldots, d_{k-1})$. The communication exchange takes place in ascending order through the dimensions. The mapping from rank to coordinates and the splitting of the communicators is described in Alg. 3. One could consider using sparse communication[10] as MPI topologies instead of splitting the communicators. However, we will not go into this in detail in this thesis, as the focus is set on aggregation.

The destination space $T_{r,s}$ of a process with rank r in step s contains all indices that can still be reached from there in the further course of the routing. It should be noted here that the distance between the ranks becomes smaller in later dimensions until they are numbered continuously in the highest dimension.

This has two positive effects: Firstly, in each step sorting the indices simultaneously results in sorting by send buffers, where it is only necessary to determine the boundaries. That is because in each step s the target space is divided into $\max(d_s)$ equally sized, consecutive sections that are distributed across the processes in the same order as the ranks. The target space can therefore be described by the following interval:

$$\begin{aligned}
\text{target_size}_{\text{s}} &= \prod_{i=s}^{k-1} \max(d_i) \\
\text{target_displacement}_{\text{r,s}} &= \left\lfloor \frac{r}{\text{target_size}_{\text{s}}} \right\rfloor \text{target_size}_{\text{s}} \\
T_{r,s} &= [\text{target_displacement}_{\text{r,s}}, \text{target_displacement}_{\text{r,s}} + \text{target_size}_{\text{s}}]
\end{aligned}$$

This is exploited by an aggregation technique introduced later. On the other hand, nearby ranks are usually also physically located close to each other in the supercomputer, for example on the same node.

This results in and lower latencies in the late routing phases [12]. In the case of irregular access patterns, the peak loads often occur in these late phases.

Another important observation can also be made: In other routing schemata, a process may receive an request for an index for which it has already forwarded another request at an earlier point in time. However, this does not occur in grid routing, as there is a unique path for each indirect stage to all indices in the target area of a process. Consequently, it is sufficient to consider only the elements currently present in the process when aggregating.

Other routing approaches, such as the node-remote-node-local routing from [18, YGM] are specifically designed to exploit the hardware topology. This routing technique could be readily incorporated into our approach, as it exhibits similar properties.

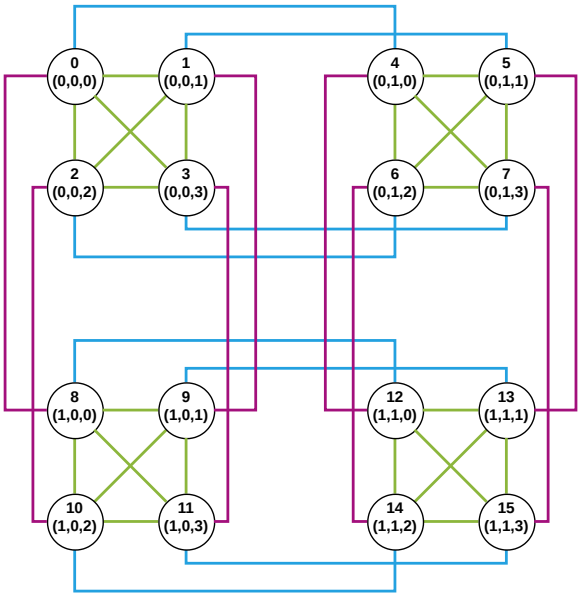


Figure 4.1: 2×2×4 grid with 4 processes per node. In the final dimension a node intern alltoallv is performed. The dimensions are ordered in the sequence red, blue, green.

4.3 How to get Routing Information

On the way of a message to its recipient, intermediate nodes can derive the destination address from the index. In the case of read requests, the requested data must find its way back to the requester after it has arrived at its recipient. This can be achieved either by tracing

Algorithm 3: Communicator splitting of a multidimensional grid

```
Input: dim_sizes
dim_fill[|dim_sizes| − d − 1] ← ∏_{i=0}^{d} dim[i], d ∈ [0, |dim_sizes| − 1]
r ← process rank
for d ∈ [0, |dim_sizes| − 1] do
    coords[d] ← r / dim_fill[d]
    r ← r mod dim_fill[d]
for d ∈ [0, |dim_sizes| − 1] do
    pivot_coords ← coords
    pivot_coords[d] ← 0
    dim_comm[d] ← comm.split(to_rank(pivot_coords))
```

the passing messages in forward flow and saving their origin, which can be combined with the aggregation tracing technique, or by writing the requester of the message in its header. If the requester of the message is written in the header, it only remains to store whether the message has been aggregated. Adding routing information to every read package will increase its size by additional 4 bytes. The following two types of packets occur:

Index packets are read requests to a process. Their size doubles due to the additional routing information.

Index-data packets contain the responses to read requests with the data. If the data is relatively small, the proportion of routing information increases.

In figure 4.2 a sketch of the packages can be seen.

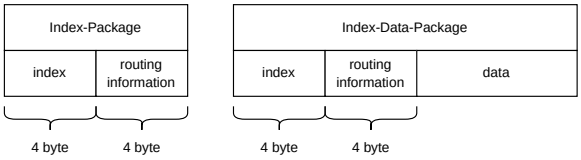


Figure 4.2: Packages with routing information

4.4 Aggregation

The purpose of aggregation is to ensure that each index is only transmitted once to the next hop in order to decrease overall communication load.

A visualization of aggregation is given in figure 4.3.

There are two different types of aggregation, local and global.

Local aggregation is when the data is initially aggregated within the requesting process. The local duplicates are spread across a large buffer and must be filtered out. The buffer must then be set up such that the messages are forwarded according to the routing pattern. While some of the aggregation techniques described below result in a correct send buffer, others require this to be generated subsequently.

In the case of global aggregation, the duplicates are aggregated with the data from other processes in an indirection stage. It can be assumed that indices, in a former step received from the same node no longer contain any duplicate indices, as they would have been aggregated earlier. Therefore, only the different partitions need to be scanned for duplicate indices.

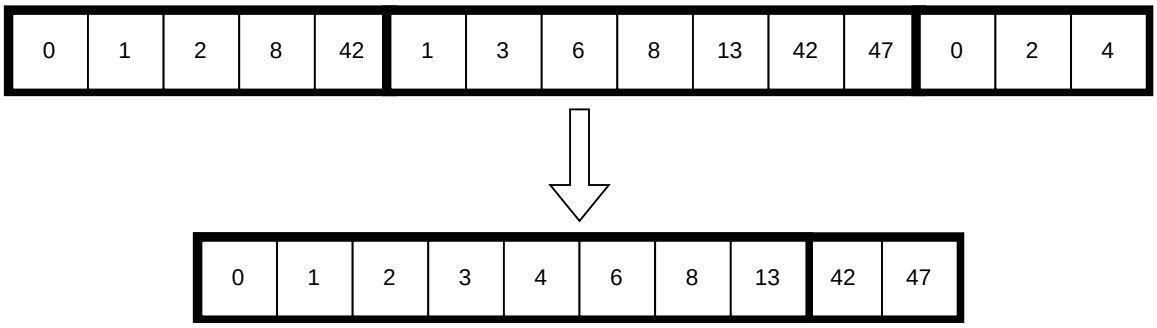


Figure 4.3: Aggregation with sorted messages. Upper line is the receive buffer. The thick lines group requests by their sending process in the last step. The bottom line shows the aggregated requests. Here the thick lines divide the buffer into different destinations for the next hop.

4.4.1 Aggregation via Hash Set

The naive approach when thinking about removing duplicates from a collection is a hash set. Using an hash set, entries can be searched for and new ones inserted in an expected constant amount of time.

However, if all entries from the receive buffer are sequentially added to a hash set, this leads to many cache misses for large amounts of data. Another point is that in this approach in every indirection stage the next hop for each message has to be re-evaluated and the send buffer has to be filled accordingly. This is circumvented in the following approach.

4.4.2 Aggregation via Search Tree

As already described in section 4.2, sorting the requests by index results in a correct send buffer. The boundaries of the different receivers in the buffer can be efficiently determined by an upper bound search.

This insight leads to the next approach of using a b-tree to detect and eliminate duplicate indices. After inserting all received indices into a search tree, they end up being sorted. Only the send counts need to be obtained by an upper bound search for each following destination.

However, this variant is likely to be slower than the previous one, since each insertion into a search tree requires $\mathcal{O}(\log m)$.

4.4.3 Aggregation via Sorting

Sorting m requests by their index naively yields in a time complexity of $\mathcal{O}(m \log m)$, which is slower than the $\mathcal{O}(m)$ using a hash set. Duplicates can be erased after sorting by checking consecutive elements. However, there are some advantages for this method:

- By sorting, the elements end up correctly in the send buffer and only the send counts need to be determined. No copies to other data structures are necessary.
- For local aggregation, we sort the entire read requests with the addresses. This can be reused later to write the data back efficiently.
- In the case of global aggregation, the elements have already been sorted by each sender, which means that the receive buffer contains sorted segments. Therefore, the entire buffer does not have to be re-sorted, but can be combined from the segments using mergesort or a priority queue. This eliminates the effort of sorting m messages to $\mathcal{O}(m \log \sqrt[k]{p}) = \mathcal{O}(\frac{m}{k} \log p)$. For all k dimensions this yields in $\mathcal{O}(m \log p)$ (For comparison: using a hashset leads to $\mathcal{O}(mk)$ and with hypercube routing $k = \log p$).

Algorithm 4: global aggregation via sorted buffers

```
1  foreach dimension do
       // calculate send counts
2      start, end ← 0
3      for i ← 0 to |send_counts| − 1 do
4          end ← min {e ∈ [start,|send_buf| − 1] | i > next_hop_local_rank(e)}
5          send_counts[i] ← end - start
6          start ← end
7      send data with send_buffer and send_counts
8      receive data in send_buffer
9      merge_presorted(send_buffer, recv_counts)
10     send_buffer ← {e_0, e_1, ... e_k ∈ send_buffer | ∀i ∈ [0, k] : e_i ≠ e_{i+1}}
```

4.5 Disaggregation

After the read requests have reached the process that contains the requested data, their responses must be forwarded back to all requesting nodes. This involves splitting up responses to aggregated requests again, which is called disaggregation.

Three buffers play an important role in this process: The index receive buffer holds the incoming requests, grouped by sender. This provides information on which responses need to be forwarded to which processes. By writing this information in the messages, as described in section 4.3, only the routing information for the aggregated requests needs to be stored. Then there is the index-data receive buffer, which contains the response data for each requested index. The third buffer is the send buffer, to which the responses must be written in the correct position to reach their requester.

A visualization is provided in figure 4.4.

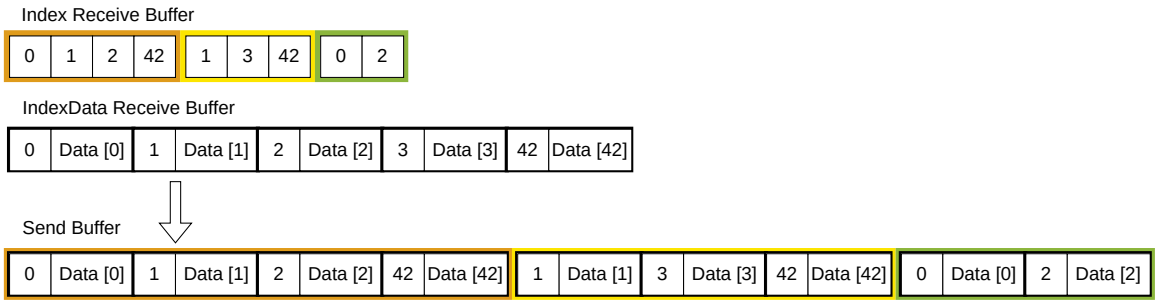


Figure 4.4: Disaggregation of sorted messages and creation of new send buffer.
The colored areas in the index receive buffer group the requested indices according to the sender of the previous stage. Data belonging to the indices from the index receive buffer must be forwarded to the same processes that made the request for this index. To resolve the indices to their data, the index-data receive buffer is provided.

4.5.1 Disaggregation via Hash Map

A hashmap can be used to link the requested indices with their data. For this purpose, the sender of the previous stage of the request is stored in the hashmap upon receipt of a request. If the routing information is stored in the messages, only the indices that have actually been aggregated need to be stored in the hashmap. During disaggregation, the received index data buffer is iterated through and the next process to which it must be forwarded is looked up in the hashmap for each index. If an index has been aggregated, there are several entries for the index in the hashmap.

As an improvement, the position in the index receive buffer can be stored instead of the sender, as this also corresponds to the position in the new send buffer. This means that the

responses can be written directly to the contiguous send buffer and no further copying is required.

This disaggregation technique is particularly compatible with aggregation using hashsets, since the key set of the hash map that we store are exactly the aggregated indices.

Disaggregation using hashmaps is not very cache-efficient with a lot of data, as their lookups result in a random access patterns.

4.5.2 Disaggregation via Search Tree

If a search tree is used for aggregation, the position in the index receive buffer, which is identical to the position of the response in the send buffer, can also be saved in its entries. Analogous to disaggregation via hash map, the corresponding position in the send buffer can be determined for each entry in the index-data receive buffer.

The variant of disaggregation via search trees is also asymptotically slower than disaggregation via hash sets, but provides a higher data locality and cache efficiency due to the sorting of the data.

4.5.3 Disaggregation via Sort and Search

This disaggregation technique requires that the indices have been requested in sorted order.

The answers will be sent in the same order in which the requests were received. Thus, the position of an index in the index receive buffer is identical to the position at which the response is written in the send buffer. This ensures that the received index-data buffer is also sorted in the order of the indices.

The requested indices are sorted as described for each sender. The indices queried by each sender form contiguous segments of the received index buffer.

To bring the data from the sorted index data buffer into the order specified by the received index buffer, the index data buffer is traversed once for each segment. Once the data for an index has been found, it is appended to the send buffer.

For a better understanding of this, consider an example. In figure 4.4, the segments of the received index buffer are highlighted in color. For each segment, the index data receive buffer is scanned through once. The elements found are appended to the send buffer.

As the requested indices are sorted for each sender, each element found reduces the search space in the index-data buffer for the next elements, as subsequent indices are equal to or larger than the previous index.

There are various methods for scanning the index-data buffer here:

- binary search: A fast way to find individual elements in a sorted array is the binary search. In each step, the size of the search space is halved, resulting in a logarithmic

runtime. However, if several closely consecutive elements are searched for, the binary search becomes sub-optimal. If many of the elements contained in a buffer are searched for, target elements are usually not far apart. This benefits approaches that traverse the buffer in a consecutive manner.

- linear search: In a linear search, each element is compared piece by piece with the searched element. This seems sub-optimal at first, but if the effort to find the next element in other approaches exceeds the distance of the elements, linear search becomes more efficient.
- exponential search: A middle ground between the two is the exponential search. Here, the search space is traversed in exponentially increasing steps until the target has been exceeded. Then the steps decrease until the element being searched for is found. This approach is slightly worse than the binary search for individual elements, but copes better with dense search spaces, as the runtime is only logarithmically related to the distance to the next match instead of to the entire search space. This is why this variant was chosen in this approach.

In the case of local disaggregation, the read requests are already sorted and each message received must be destined for this node. Therefore, it is possible to assign the sorted answers to the equally sorted queries in a linear iteration.

Since this method uses the entire index buffer, meaning that the routing information is available for all responses, there is no benefit in writing the requester in the messages. In addition, the information about the requester would only provide information about the correct section of the send buffer. For this disaggregation technique to work, however, the exact sorted order of the messages is required.

Algorithm 5: global disaggregation via received sorted buffers

```
1  foreach dimension do
2      index_recv_buffer, index_recv_counts ← request_stack.pop()
3      pos ← 0
4      foreach c ∈ index_send_counts do
5          next_entry ← 0
6          for i ← 0 to c − 1 do
               // lower bound search for next index
7              next_entry ← min {i∈ [next_entry, |index_data_recv_buffer| − 1)|
               index_data_recv_buffer[i].index ≥ index_recv_buffer[pos]}
8              pos ← pos +1
9              send_buffer.push(index_data_recv_buffer[next_entry])
10     send data with send_buffer and index_recv_counts
```

Algorithm 6: exponential lower bound

```
Input: start, end, target
if start = end or start→index = target then
    return start
interval_size ← 1
while (start + interval_size) < end and (start + invertval_size)→index < index do
    start ← start + interval_size
    interval_size ← 2 interval_size
while interval_size ≠ 0 and start→index ≠ target do
    if (start + interval_size) < end and (start + invertval_size)→index < index then
        start ← start + interval_size
    interval_size ← interval_size / 2
if start→index ≠ target then
    start ← start +1
return start
```

5 Experimental Evaluation

Since the runtime of distributed algorithms depends on many factors, a purely theoretical analysis of the approaches is not sufficient. Even if an approach scales better asymptotically, it can still be associated with high constant factors or be slower due to effects that are not covered by the theoretical model. For this reason, a series of experiments will be performed in this section to determine which combination of techniques perform the best. More precisely, communication patterns that occur in known graph algorithms are considered and the behaviour with maximum global aggregation is investigated.

5.1 Setup

All benchmarks were performed on the SuperMUC-NG Phase 1 at the Leibniz Supercomputing Center [1]. The system consists of eight islands, each comprising 792 nodes. One node is composed of two Intel Skylake Xeon Platinum 8174, each with 24 cores, resulting in 48 cores per node. However, the experiments have only been executed up to a scale of 4096 cores, which means they are within an island.

A 100 GB/s OmniPath network arranged in a fat tree is used for communication between the nodes. The MPI wrapper KaMPIng [8] is used on top of the MPI implementation intel-mpi 2021.9.0. The code is compiled, using gcc 12.2.0 with optimization level -O3.

5.2 Tuning Parameters

The behavior and performance of the distributed array is influenced by various decisions, such as the choice of routing scheme, the storage of routing information or the algorithm for dis- & aggregation. The following gives an overview of the investigated approaches and combinations of them.

5.2.1 Routing Scheme

A number of grid routing schemes with varying numbers of indirection hops and shapes of the indirecting groups were investigated. The number of intermediaries describes the amount of startups required for a single message to reach its destination. The shape defines

the size of the groups for the respective exchange in an indirection step and therefore the amount of startups in a indirection step.

As previously described in section 4.2, larger dimensions result in more startups and associated latency costs. Consequently, a larger number of messages can be aggregated since received messages originate from a larger set of sources. The following routing schemata were investigated: `equal`, `hypercube`, `increase`, `decrease`, `hourglass`, `inverse hourglass` and `locality`.

As the name implies, the dimensions in the `equal` pattern are of equal size. The `hypercube` represents a special case of the `equal` pattern, where the dimensions are always a size of two. The name comes from the fact that a multidimensional grid with side length 2 forms a multidimensional cube.

The `increase` pattern begins with small dimensions and gradually increases in size. For 4096 processors, this pattern has the shape $2 \times 2 \times 2 \times 4 \times 8 \times 16$. Other distributions are also conceivable. However, to ensure comparability with the other routing schemes, the number of dimensions set to six. Once a certain dimension has been exceeded (in this case for the last dimension), communication only occurs on an intra node level. Resulting low latencies are being utilized by the larger exchanges in the end phases.

In contrast, the `decrease` pattern involves a reduction in the dimensions sizes. The pattern has exactly the opposite shape of its predecessor. Large beginning dimensions allow early aggregation of more messages, reducing the number of messages that must be delivered in subsequent phases.

The `hourglass` pattern ($8 \times 4 \times 2 \times 2 \times 4 \times 8$) attempts to combine the two preceded approaches. Large dimensions at the beginning and end of the communication process utilize the described effects. Lean centre dimensions reduce startups. Conversely, the `inverse hourglass` pattern, which employs large dimensions in the middle of the communication process, is investigated as reference.

A hybrid approach combining node remote routing from [18] and the hypercube forms the `locality` aware routing. Here, a hypercube is used at the inter-node level, up to a dimension size of 16, which is the largest common divisor of the 48 cores per node and larger powers of two, which is required for some instances.

For better comparability, the number of dimensions of each pattern is given.

The baseline builds `plain all to all`, where each process delivers its data directly via a global all-to-allv call. There is also an RDMA-based variant without indirection mentioned as `Plain RDMA`, but due to the time constraints of this thesis, this variant is not included in all benchmarks.

5.2.2 Routing Information

As already described in section 4.3, the routing information can either be stored in the messages or locally. If the requester is sent with the message, an analogy can be made

to a envelope with the sender's name on it, which is why these approaches are labelled envelope. As it would go beyond the scope of this thesis to list all possible combinations, only the following approaches are considered here:

`envelope hash` denotes aggregation by a hashset and disaggregation by a hashmap.

`envelope sort` denotes aggregation by sorting and disaggregation by a hashmap, as disaggregation by sorting cannot be combined with `envelope`.

5.2.3 Algorithm Choice

Various aggregation and disaggregation techniques were presented in sections 4.4 and 4.5.

It is convenient to combine similar techniques in aggregation and disaggregation. If the experiments show that differing approaches perform particularly well in aggregation or disaggregation, it should be investigated whether they can be combined.

The following self-talking approaches result: `hash`, `search tree` and `sort`.

In the benchmarks where aggregation has been switched off, the routing technique of the respective approach is used as far as possible. Thus, in the case of `sort`, mergesort is used to place the messages correctly in the send buffer and in the case of `hash`, the map is used to deliver all messages correctly.

In the case of `plain all to all`, the messages are sorted by recipient and index and then aggregated.

5.3 Instances

Distributed graph algorithms offer a wide range of options for different access patterns that occur in practice. Different types of graphs have varying characteristics, such as locality, diameter or degree distribution. Graphs with a high degree of locality form densely connected communities. Highly inhomogeneous degree distributions can lead to overloading of individual processes.

To generate the instances, KaGen [6] is used, which assigns a consecutive interval of nodes to each process, enabling a convenient integration with the distributed array. Typically, there is one entry in the array for each node of the graph. As a result, the majority of the entries for the nodes of a process are local to this process.

5.3.1 R-MAT

R-MAT graphs are characterized by low amount of locality, small diameter and an exponential degree distribution. These properties are also found in real-world graphs and are similar to those of social networks.

When generating R-MATs, the adjacency matrix is divided recursively into four parts. According to a probability distribution, one of the parts is selected until a single element is reached, which is then taken as an edge. The specifications from [13, Graph500] were used as the parameters.

This type of graph is very suitable for aggregation, as the high degree vertices are frequently queried in many applications.

5.3.2 RGG

In a random geometric graph, the n nodes are distributed in a metric space. The m edges are chosen such they conntect all vertices within a given radius. This typically creates a community structure in which parts of the graph are densely interconnected. However, the chosen parameters do not allow for any major developments of these communities.

RGG instances are therefore characterized by a medium degree of locality with a large diameter.

Since the edges of the RGG graph connect close vertices, it can be assumed that there is a lot of potential for local aggregation on the same process, for which no indirection is required.

5.3.3 GNM

A GNM graph is uniformly chosen from the set of graphs with n nodes and m edges. Due to the random distribution, there is almost no locality. The degree distribution is homogeneous as it is unlikely that individual nodes have many or no connections.

5.4 Evaluation

There are several use cases where multiple processes repeatedly access same data.

In combination with the structure of the underlying graph, this results in various access patterns that are evaluated in the following benchmarks.

The aim is to analyse the effects of semantic aggregation combined with indirect routing. For this purpose, the synergy of different grid routing patterns with the aggregation techniques described in section 4 will be compared.

The values measured from the benchmarks were averaged over five iterations. No significant deviations between iterations were observed. The individual measurements were synchronised and the process that took the longest to complete was taken as the measurement.

5.4.1 Boruvka

Boruvka's algorithm [15] is designed to determine minimum spanning trees on undirected graphs. For this purpose, each node identifies the edge with the lowest weight in each step and includes it in the spanning tree. The resulting components are merged using pointer doubling. This process is repeated recursively. To minimize the impact of design decisions in the implementation, only the first round of the algorithm is performed in the benchmark. In pointer doubling, there are many accesses to the same elements.

The graph instances have a size of 2^{13} nodes and 2^{17} edges per process. For a weak scaling benchmark the number of vertices per rank is fixed, while the number of ranks is varied. Only the time spent on accessing the distributed array is measured. Local work within the algorithm is ignored.

Different Number of Processes

The behavior of the different approaches, for an increasing number of processes, is shown in figure 5.1. Sorting was chosen as aggregation algorithm, which as will be shown later, proves to be the best variant. Since not all numbers of processors allow every topology, the best result of all topologies is also given. The behaviour with and without semantic aggregation is examined. Indirect approaches are compared with direct approaches using MPI and RDMA. Direct approaches can only be aggregated locally.

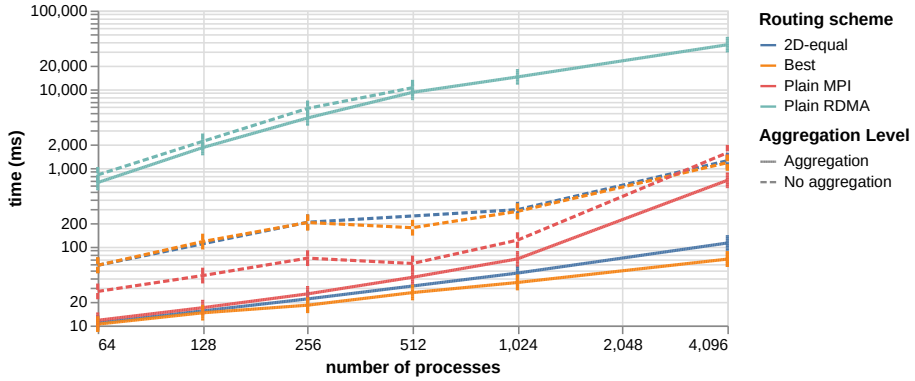
For 4096 processes, a speedup of up to 20 is possible on the RMAT graph compared to a MPI all-to-allv without aggregation and indirection. Semantic aggregation in particular pays off here and contributes to the majority of the speedup.

On the other two graph types, the indirect approaches also scale better, but the speedup due to aggregation is much smaller at around 15%.

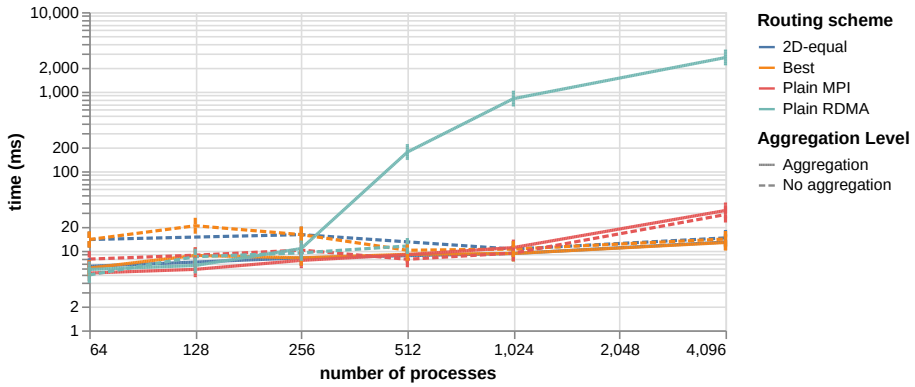
Algorithm and Topology Choice

For a more detailed comparison of the correlation between the algorithm and the routing scheme, the experiment on R-MAT graphs with 4096 processes is provided in figure 5.2. It also shows detailed running times for the different algorithm phases of the approaches. The detailed plots of the other instances can be found in the section A.1 of the appendix.

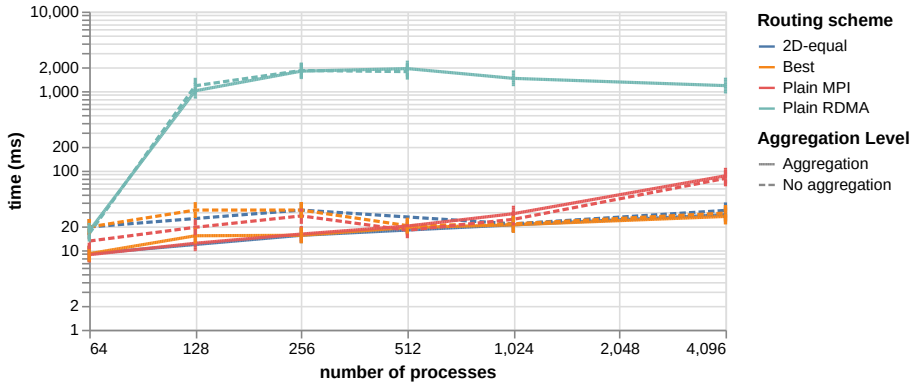
The most efficient technique is `sort`. This is likely due to its straightforward integration into the routing process, eliminating the need for additional data structures to be rebuilt. As aggregation and disaggregation are closely intertwined with routing in the `sort` approach, the use of `envelope` does not provide any advantages. A closer look at the individual phases reveals that the hash-based approaches take considerably longer to write compared to the other approaches. From a theoretical point of view, this is unusual, but an examination of the implementation did not reveal any anomalies that would explain this behavior.



(a) Boruvka benchmark on R-MAT



(b) Boruvka benchmark on RGG



(c) Boruvka benchmark on GNM

Figure 5.1: Scaling behaviour of the approaches for the boruvka benchmark. The orange chart combines the best topologies, as not all topologies can be constructed for each number of processes.

Note that some approaches, such as `sort`, benefit from more dimensions, while others, such as `hash`, perform better with smaller dimensions. This indicates that `hash` has a worse performance than the time saved by reducing startups. A review of the other instances reveals, that instances with less irregular access patterns exhibit a negative effect when there are too many indirection steps.

Surprisingly, no significant advantage can be found by using different shaped grid routing schemes with the amount of intermediaries. There may not have been sufficient differences in the communication groups for any effects to occur.

5.4.2 Neighborhood

In this benchmark, each node queries its neighboring nodes. The querying of neighbouring nodes can be found in algorithms such as a breadth-first search and therefore constitutes a communication pattern to be investigated. Different forms of the pattern are determined by the choice of the underlying graph instance.

The graph instances again have 2^{13} nodes, but with 2^{15} slightly fewer edges per process than the Boruvka benchmark, as this benchmark has a potentially larger volume of communication on more different indices.

Different Number of Processes

Figure 5.3 shows running times for weak scaling benchmarks for the three graph types. On the RGG graph, indirection alone is not sufficient to outperform the conventional variant. However, when semantic aggregation is included, it is outperformed. Remarkably, the execution time is reduced with growing number of processes for small instances despite the increasing problem size.

Algorithm and Topology Choice

A closer look at the individual algorithms, routing schemes and phases on a R-Mat graph in figure 5.4 shows that `sort` also performs the fastest for this benchmark. In-detail plots of the RGG instance can again be found in section A.2 of the appendix.

For the R-MAT instance, the majority of time is spent on local aggregation, which corresponds to the initial sorting of all requests. This phenomenon is likely attributable to the exponential degree distribution of R-MAT graphs. In instances with more homogeneous degree distributions such as GNM, this effect can not be observed.

The choice of routing scheme for the `sort` approach only shows advantages on the GNM instance, where a number of three indirection steps performs best, as can be observed in figure 5.5.

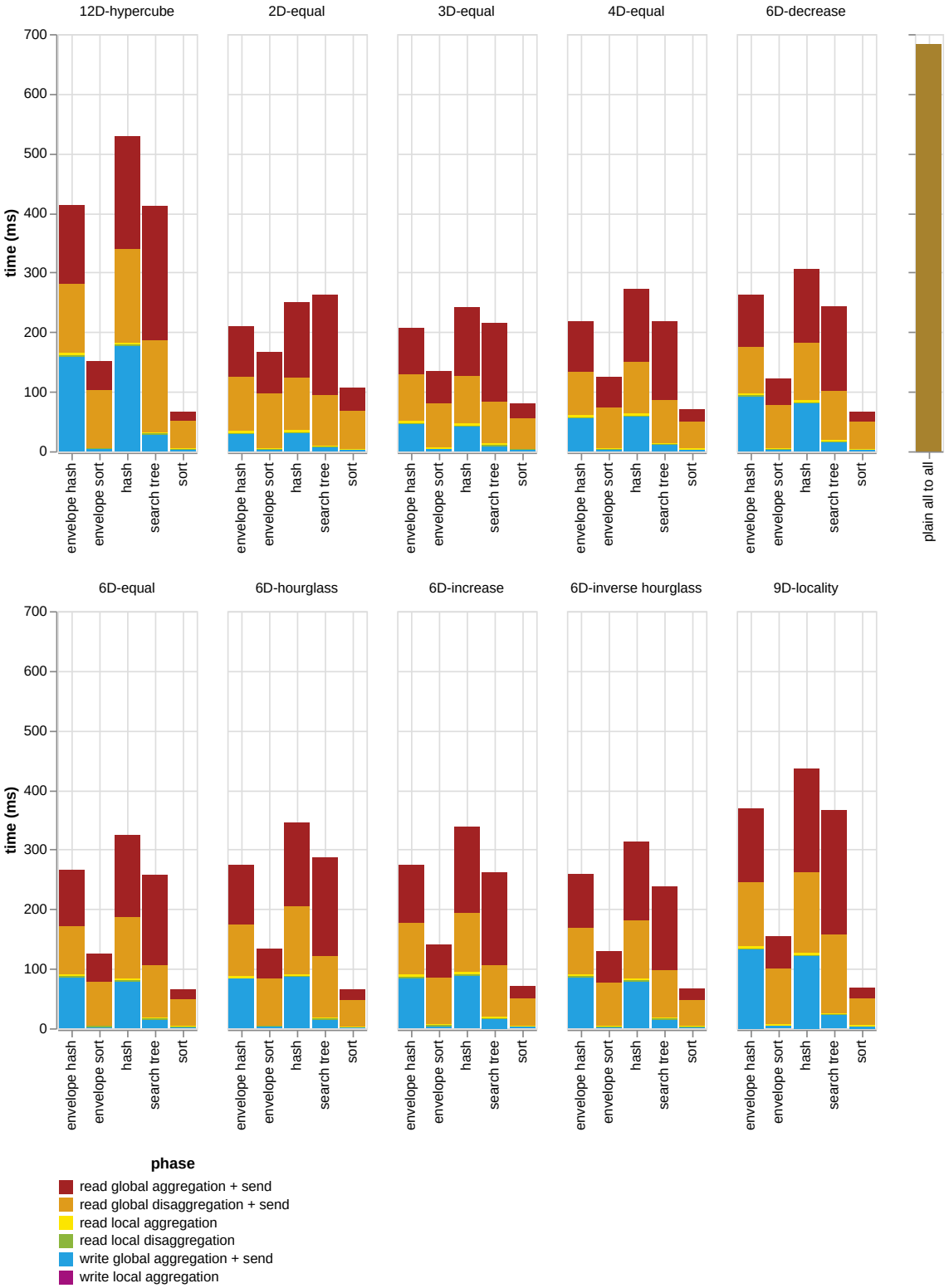
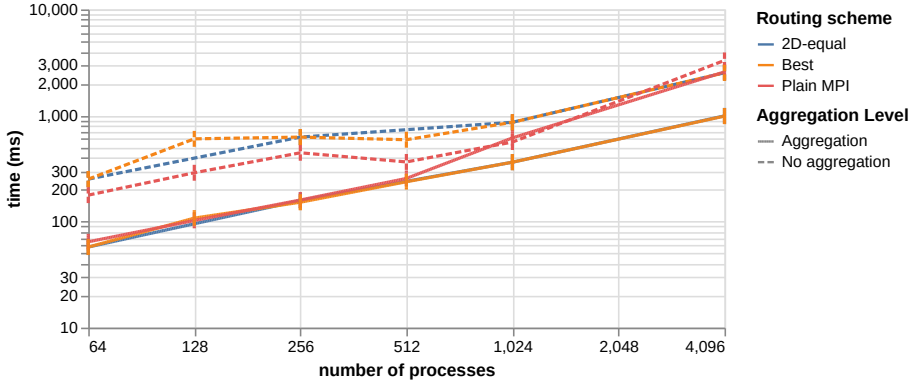
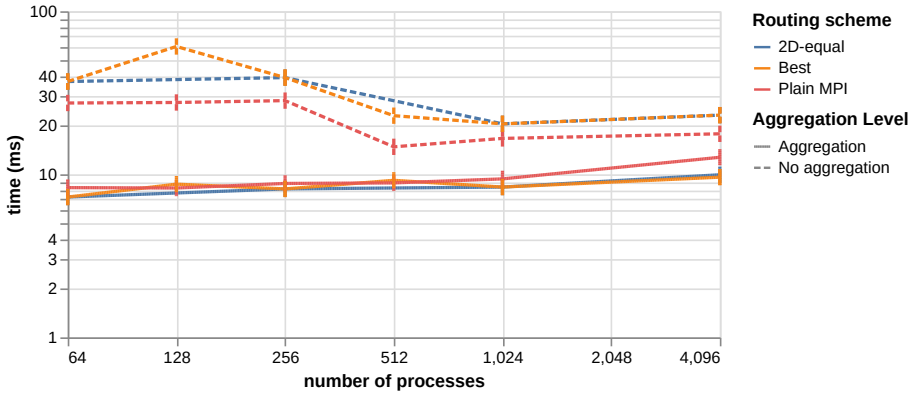


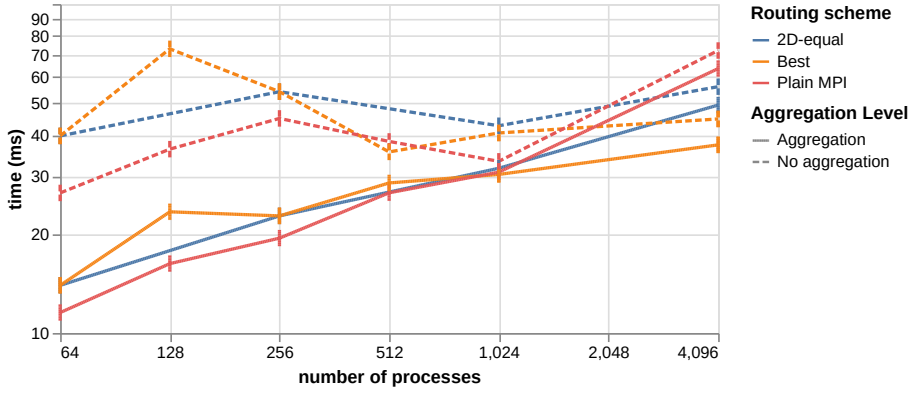
Figure 5.2: Boruvka benchmark on R-MAT graph, 4096 processes.



(a) Neighborhood benchmark on R-MAT



(b) Neighborhood benchmark on RGG



(c) Neighborhood benchmark on GNM

Figure 5.3: Scaling behaviour of the approaches for the neighborhood benchmark. The orange chart combines the best topologies, as not all topologies can be constructed for each number of processes.

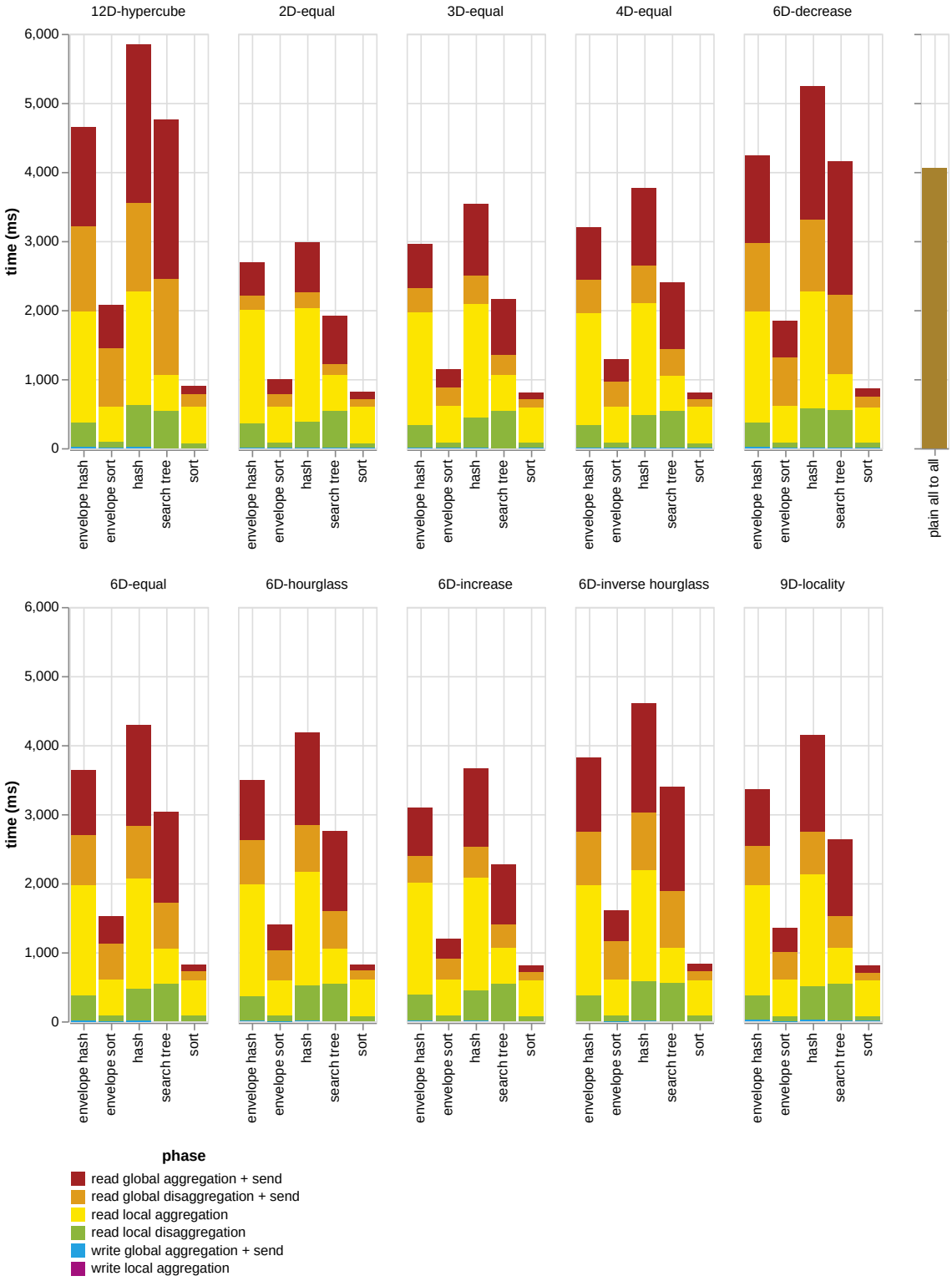


Figure 5.4: Neighborhood benchmark on R-MAT graph, 4096 processes.

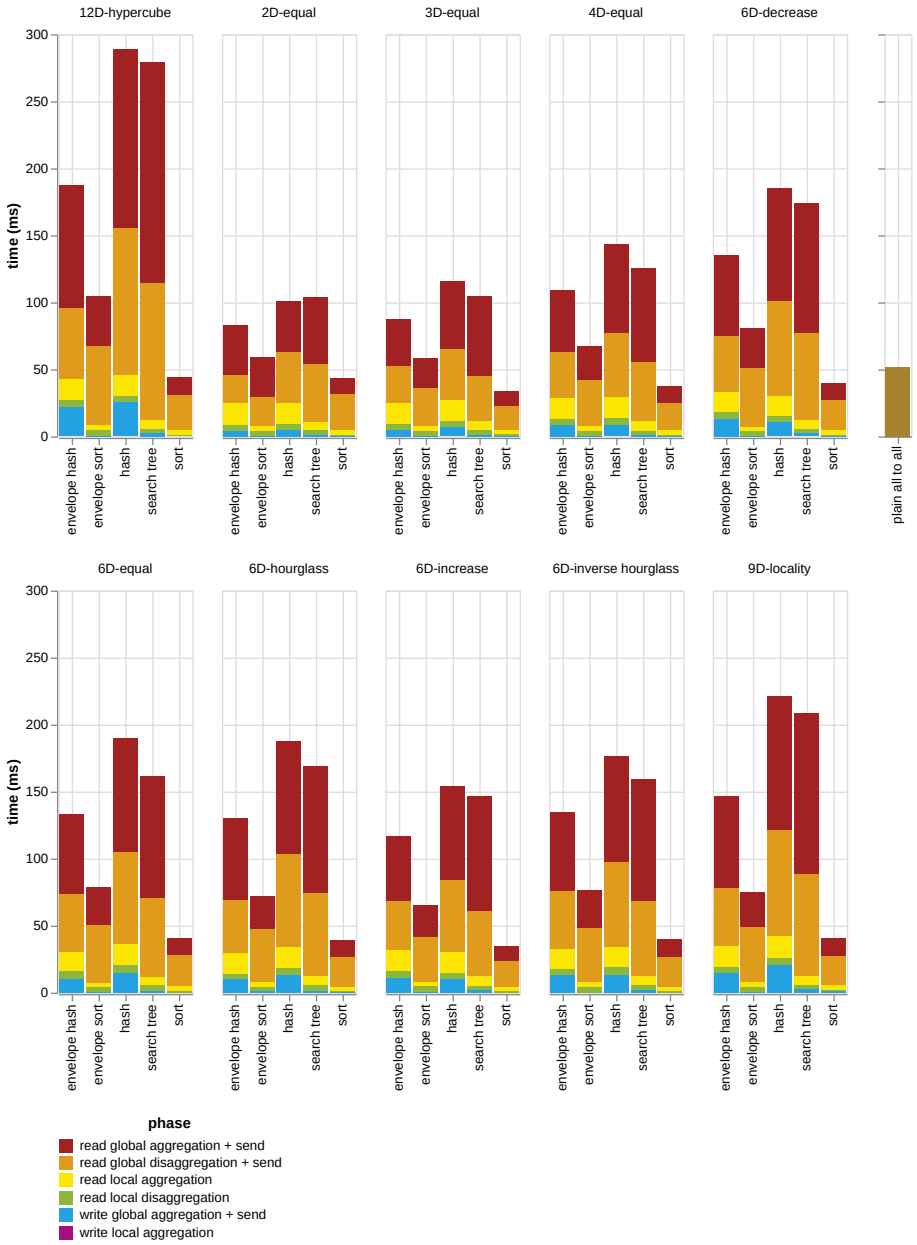


Figure 5.5: Detailed neighborhood benchmark on a GNM graph, 4096 Processes

5.4.3 Overload

In this benchmark, a process is overloaded with requests. To achieve this, all processes submit read and write requests for all indices held by a single process.

The benchmark was evaluated on two instances, with each process holding 2^8 and 2^{13} indices, respectively.

The structure of the benchmark offers a very large potential for global aggregation, local aggregation does not occur. This is also reflected in the results to be seen in figure 5.6, as semantic aggregation without indirection offers no benefit.

Given that the small-scale benchmark already demonstrates the effects of pure indirection, the larger-scale benchmarks were conducted only with aggregation in order to avoid unnecessarily large execution times. Approaches without indirection are therefore also left out.

An interesting observation is the major difference of a factor of 20 between the 2D grid and the best topology, which in this case is exclusively the hypercube.

In the hypercube, the transmission load is halved in each direction, resulting in a single request per index reaching the target process. In contrast, in the 2D grid, $\sqrt{p}$ requests with the same index arrive at the target process. Furthermore, the high startup load of the process holding the requested elements is significantly reduced by the hypercube routing.

As anticipated, this indicates that as the irregularity of communication increases, indirect routing steps may be employed to compensate for this irregularity to a certain extent.

5.5 Results

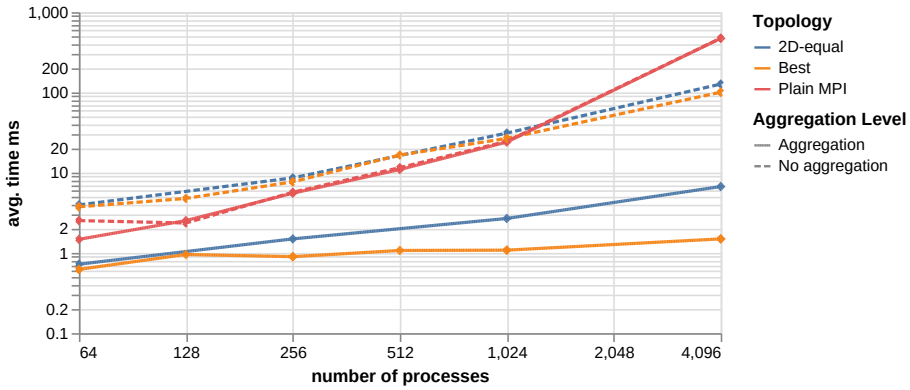
The presented approaches were examined with regard to scalability and interaction. It was found that the approach based on sorting is particularly well suited to combination with indirect routing.

Highly irregular communication patterns offer greater potential for improvement through indirect routing, as the many startups of a few set of processes are better compensated for. The greater the number of processes involved in a communication process, the greater the irregularities that may result. Different shapes in the size of intermediate exchanges within the grid routing scheme had less effect than expected. The number of indirections is a more important measure when choosing a routing scheme. The greater the degree of irregularity in the communication pattern, the more many indirect hops pay off.

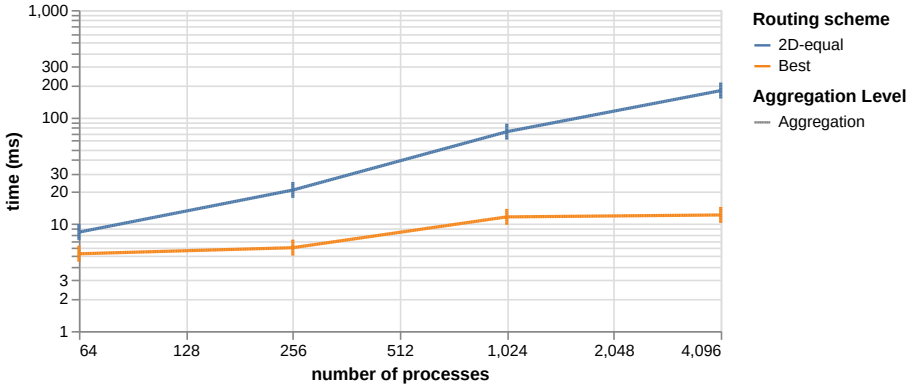
The semantic aggregation in conjunction with indirect routing can result in a speedup of up to 20 in certain instances (boruvka, R-MAT, 4096 processes) when compared with techniques that do not employ indirect routing and aggregation. Of this, a 16-fold speedup can be attributed to the semantic aggregation alone.

On the other hand, there are no disadvantages due to semantic aggregation to be found in other instances. Further research is needed to determine the extent to which these findings can be generalised.

Semantic aggregation is thus a valuable instrument for improving communication in particular in the context of distributed graph algorithms. It should therefore always be taken into account when implementing algorithms with semantically identical messages.



(a) Small scale overload benchmark, 256 elements per process



(b) Larger scale overload benchmark, 8192 elements per process

Figure 5.6: Scaling of the overload benchmark. The orange chart represents best observed routing scheme, which is always the hypercube for this case.

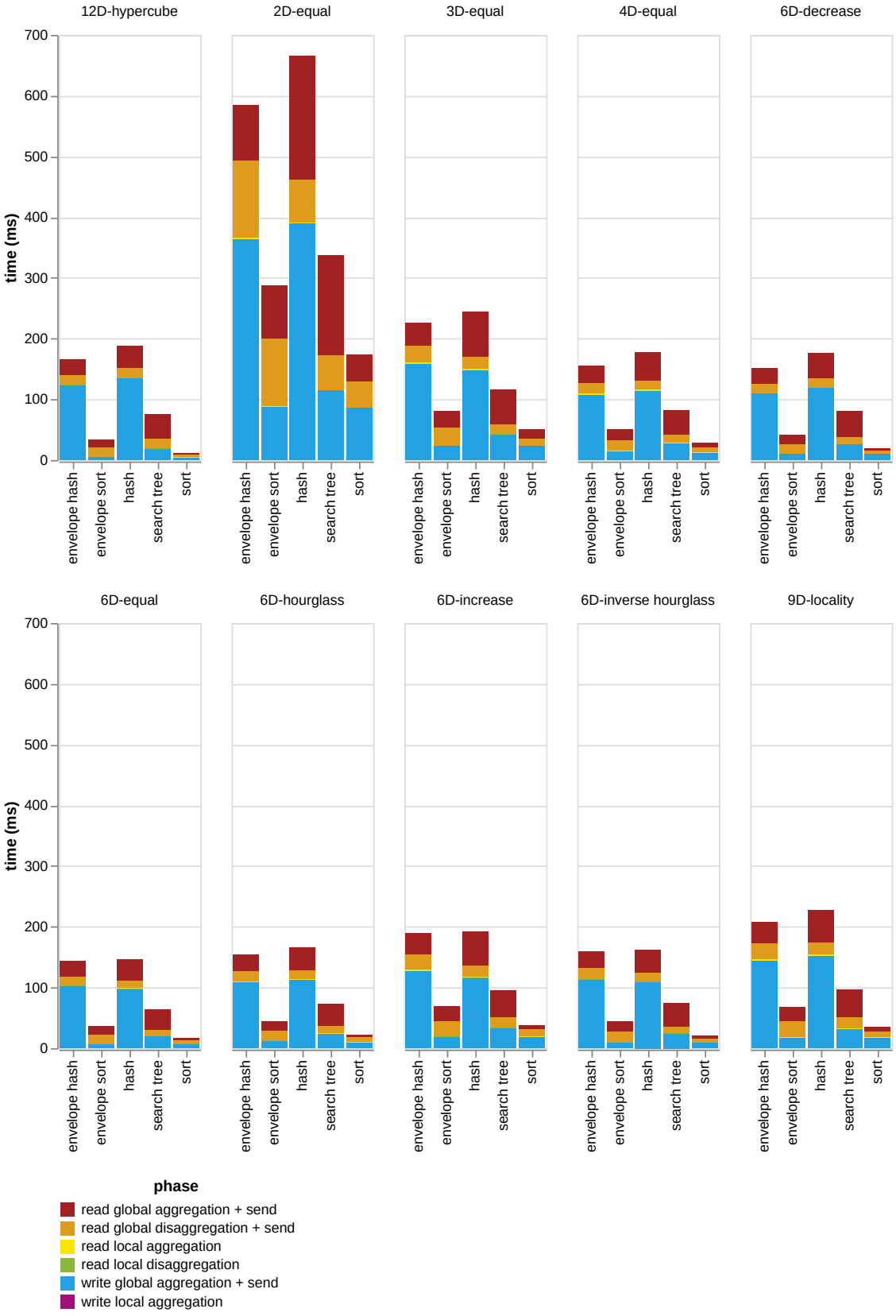


Figure 5.7: Detailed comparison of routing schemes, phases and aggregation algorithms for the overload benchmark on 8192 elements, 4096 processes.

6 Discussion

6.1 Conclusion

This thesis demonstrates different techniques on how messages can be indirected and aggregated in a distributed-memory setting, which is especially advantageous for irregular all-to-all communication. Based on the combination of messages through indirect routing in order to reduce startup latencies, duplicate messages can also be aggregated on a semantic level and thus reduce the communication workload.

A distributed array is chosen as the semantic context for this. Similar read requests and write requests can be eliminated or overwritten at the stages of indirection.

Various techniques for aggregation and variants of grid routing were investigated and evaluated experimentally on common communication patterns of distributed graph algorithms.

The results show a speedup of up to 20 times compared to approaches without indirection and aggregation, for common use cases as first iteration of Boruvka's algorithm on a R-MAT graph, up to 4096 processors. In this particular use case, semantic aggregation was identified as the main factor responsible for the speedup. Compared to indirect approaches without semantic aggregation, a speedup of 16 was observed on the same instance.

It was found that the most efficient way to identify duplicate messages is to sort them, as the resulting order can be utilized by multidimensional grid-based routing. In addition, the grid routing scheme facilitates the process of sorting into a k-way merge.

When choosing a routing scheme, only the number of intermediate hops to a destination had a significant impact on the performance of this approach. Different shapes in size of the intermediate exchanges affected performance less than expected.

In conclusion, semantic aggregation can be well integrated into indirect routing and offers great potential for some algorithms and instances to speed up communication. No significant overhead was found, but future research should focus more on this in order to make it suitable for general algorithms and problem instances.

6.2 Future Work

In the future, routing schemes that specialise even more in hardware topology, such as Node-Local Node-Remote routing proposed by Steil et al.[18], could be combined with semantic aggregation, as this has proven to be beneficial.

Furthermore, orthogonal topics such as load balancing and flow control need to be further investigated. To this end, the theoretical approach of Ranade [17] should be considered, since it provides guarantees of runtime.

Also, other underlying information exchange technologies such as RDMA could be explored.

Other open topics are worst-case scenarios for this approach and the detailed behaviour for different percentages of identical messages. Variations in message length should also be investigated, as aggregation of longer messages can save more transmission volume, but indirect routing becomes more expensive.

A Detailed Plots

A.1 Boruvka

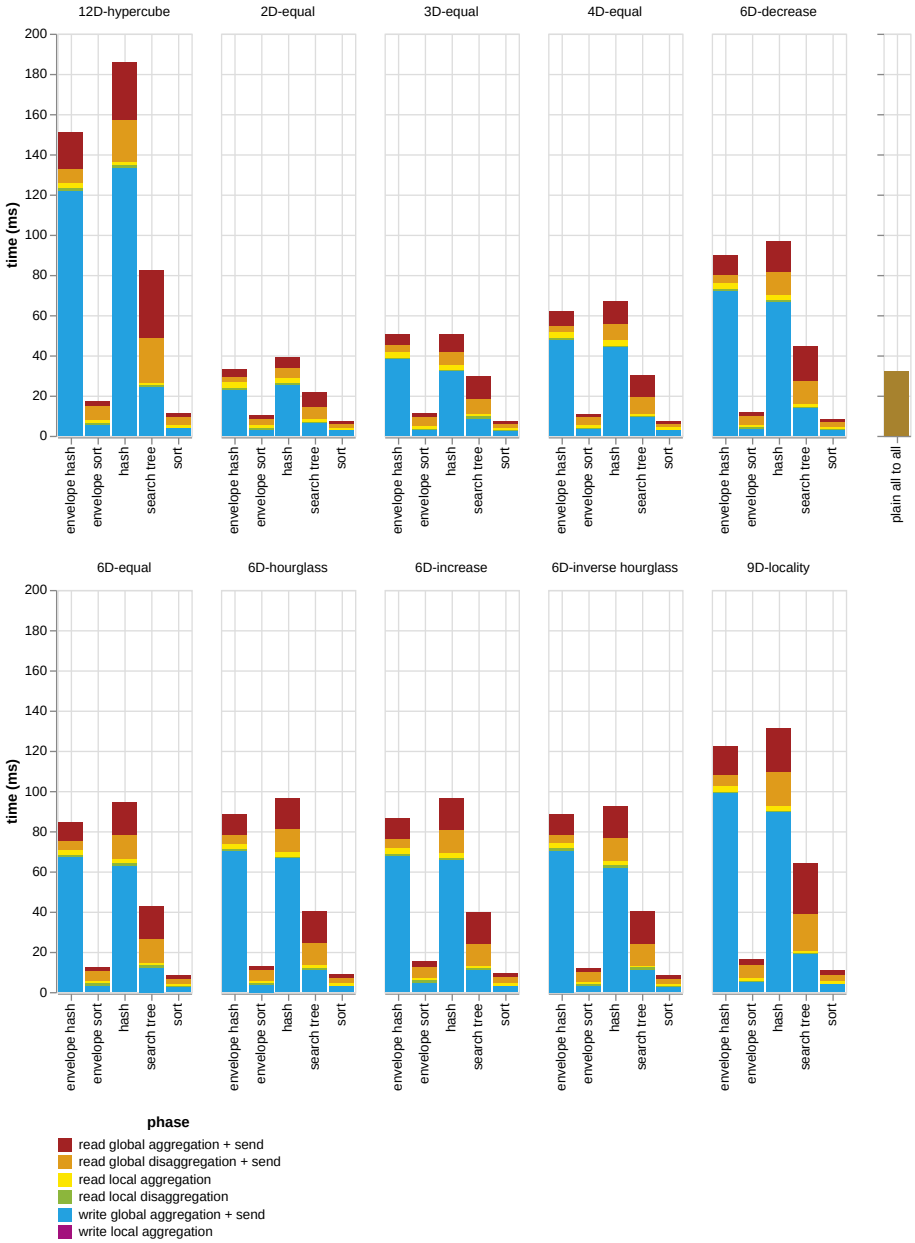


Figure A.1: Detailed boruvka benchmark on a RGG graph, 4096 Processes

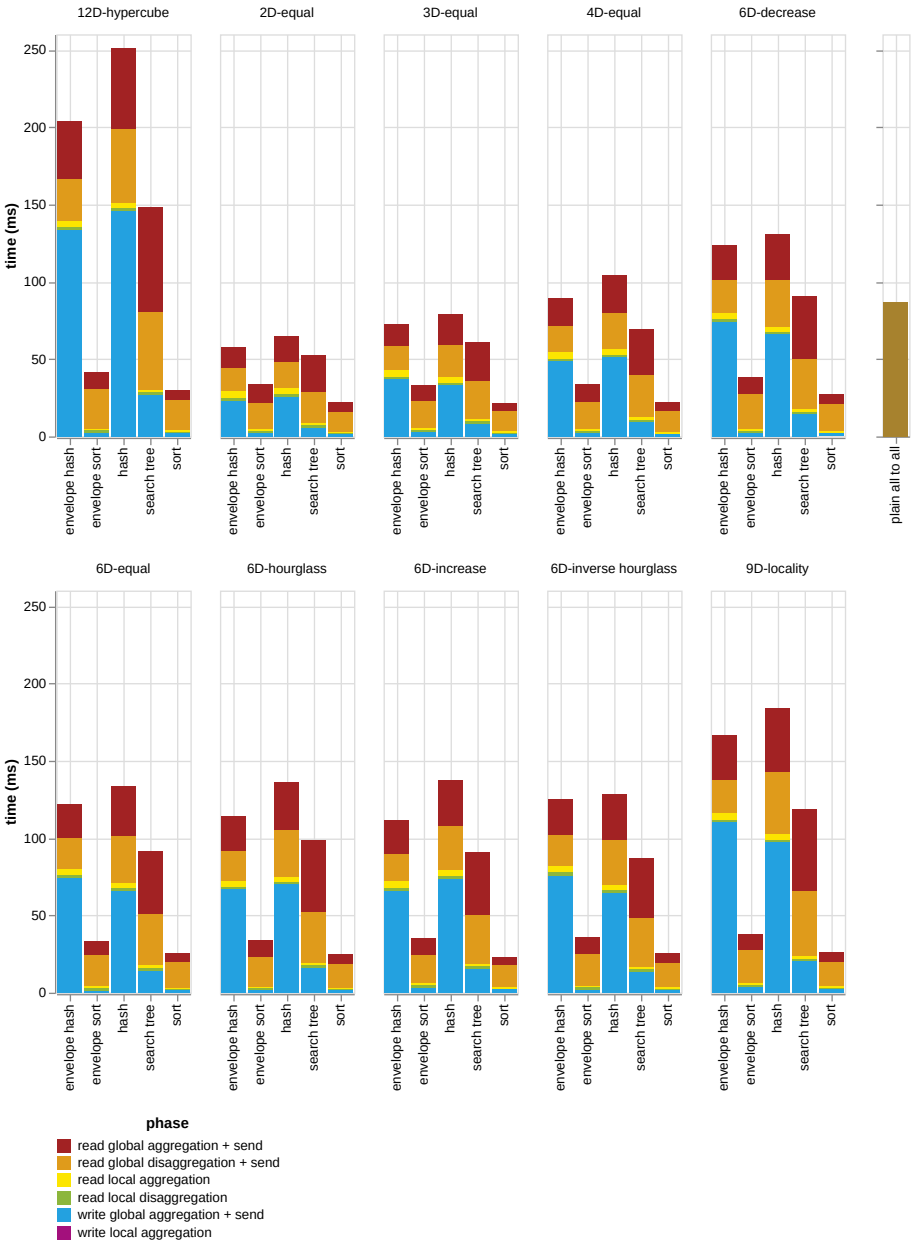


Figure A.2: Detailed boruvka benchmark on a GNM graph, 4096 Processes

A.2 Neighborhood

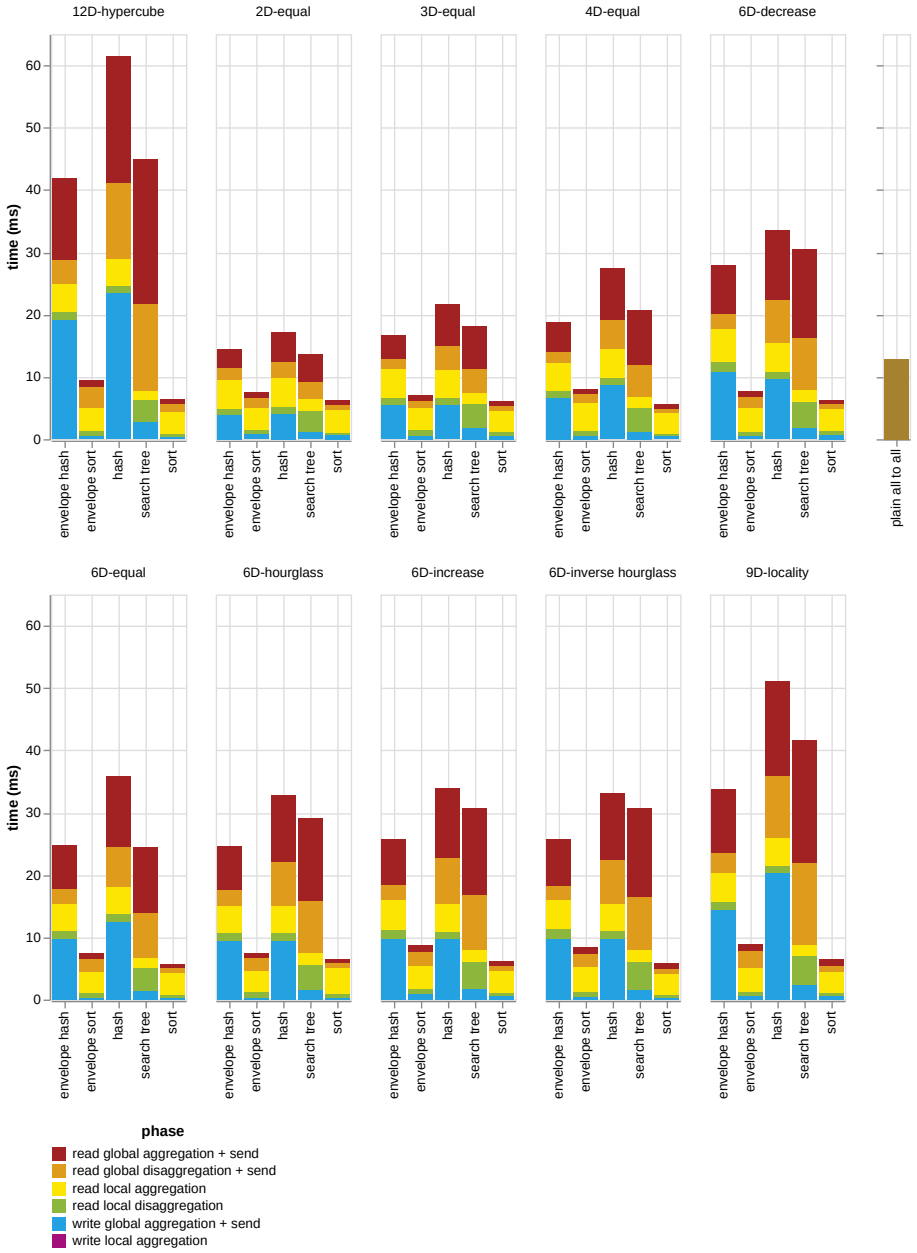


Figure A.3: Detailed neighborhood benchmark on a RGG graph, 4096 Processes

Bibliography

[1] Lrz hardware specification. Available at `https://doku.lrz.de/hardware-of-supermuc-ng-phase-1-11482553.html`.

[2] Mpi forum. `https://www.mpi-forum.org/`.

[3] Gerald Collom, Rui Peng Li, and Amanda Bienz. Optimizing irregular communication with neighborhood collectives and locality-aware parallelism. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 427–437, New York, NY, USA, 2023. Association for Computing Machinery.

[4] Baorong Ding, Mingcong Han, and Rong Chen. Darray: A high performance rdma-based distributed array. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 715–724, 2023.

[5] Ke Fan, Thomas Gilray, Valerio Pascucci, Xuan Huang, Kristopher Micinski, and Sidharth Kumar. Optimizing the bruck algorithm for non-uniform all-to-all communication. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '22, page 172–184, New York, NY, USA, 2022. Association for Computing Machinery.

[6] Daniel Funke, Sebastian Lamm, Ulrich Meyer, Peter Sanders, Manuel Penschuck, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation, 2019.

[7] Andrew Geyko, Gerald Collom, Derek Schafer, Patrick Bridges, and Amanda Bienz. A more scalable sparse dynamic data exchange, 2024.

[8] Demian Hespe, Lukas Hübner, Florian Kurpicz, Peter Sanders, Matthias Schimek, Daniel Seemaier, Christoph Stelz, and Tim Niklas Uhl. Kamping: Flexible and (near) zero-overhead c++ bindings for mpi. *arXiv preprint arXiv:2404.05610*, 2024.

[9] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, page 159–168, New York, NY, USA, 2010. Association for Computing Machinery.

[10] Torsten Hoefler and Jesper Larsson Traff. Sparse collective operations for mpi. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, 2009.

[11] L.V. Kale, S. Kumar, and K. Varadarajan. A framework for collective personalized

communication. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 9 pp.–, 2003.

[12] A.R. Mamidala, Lei Chai, Hyun-Wook Jin, and D.K. Panda. Efficient smp-aware mpi-level broadcast over infiniband's hardware multicast. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 8 pp.–, 2006.

[13] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19(45-74):22, 2010.

[14] Naeris Netterville, Ke Fan, Sidharth Kumar, and Thomas Gilray. A visual guide to mpi all-to-all. In *2022 IEEE 29th International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*, pages 20–27, 2022.

[15] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1):3–36, 2001. Czech and Slovak 2.

[16] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:169–189, 1996.

[17] Abhiram G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42(3):307–326, 1991.

[18] Trevor Steil, Tahsin Reza, Benjamin Priest, and Roger Pearce. Embracing irregular parallelism in hpc with ygm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[19] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.