Department of Informatics
Technische Universität München

Master's Thesis in Informatics

Design and Implementation of a Lightweight Communication Backend for HPC/Distributed Applications

Entwurf und Implementierung eines schlanken Kommunikationsbackends für verteilte HPC-Anwendungen

Author:	Alexander Kurtz
Supervisor:	PD Dr. rer. nat. Josef Weidendorfer
Advisor:	Dai Yang, M.Sc.
Submission date:	May 15, 2018

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Alexander Kurtz, May 15, 2018

Abstract

LAIK is a new library which makes writing Single-Program-Multiple-Data (SPMD) programs easier by abstracting the necessary inter-process communication. To achieve this, it requires the programmer to explicitly define the data containers shared between processes and to provide a mapping between simple integer indices and the specific data elements. Since LAIK then both knows about and manages all the shared data, it can automatically distribute the data among the available workers and facilitate the necessary communication to synchronize the individual processes.

For actually sending and receiving data between the collaborating processes, LAIK relies on its backend which also provides the necessary management information used to determine the total amount of processes and the local ID among those. Originally, LAIK only had a single backend which used the MPI API to talk to various MPI implementations. Unfortunately, this meant that LAIK's backend interface was not well tested and that a working MPI implementation was a prerequisite to run LAIK applications.

In this work, we extend LAIK with a new backend using native TCP sockets provided by the operating system. We first introduce the design by presenting the main challenges to overcome and the solutions we found for them. Then we introduce the resulting implementation and evaluate it in comparison to the existing MPI backend. For this, we use a small cluster of single board computers (SBDs) with a custom OpenMPI installation and a fast network interconnect.

We demonstrate that our new TCP backend can provide comparable performance to the existing MPI backend in most cases, with a few test case even showing significantly lower total execution times with the new backend in use. Most importantly, we find that the new TCP backend works especially well if given many send/receive operations per invocation, with a medium or large amount of bytes to transmit per operation. Conversely, the performance worsens dramatically if the backend is given only a few (or just one) operation per invocation and/or very small messages to transmit.

Finally, this work also showcases a way of bringing fault tolerance transparently to SPMD applications using LAIK by generating unique identifiers for each exchanged message. By simply not removing outgoing messages from the output buffer after successfully delivering them, we allow failed instances to be restarted (possibly even on a different host) and to regain their lost state by requesting the necessary messages once more. We also show that this approach to fault tolerance is possible while only using a subset of the MPI API to communicate with the core of LAIK, suggesting that this approach may also be used in existing MPI implementations.

Contents

1 Introduction

The Lightweight Application-Integrated data distribution for parallel worKers (LAIK) library is designed both for making Single-Program-Multiple-Data (SPMD) programs easier to write and for extending these programs with new features, such as communication abstraction, automatic work distribution, and eventually some level of fault tolerance. With the basic idea first presented in [47] and later refined in [48], LAIK is nowadays developed as an open source project [14] on GitHub.[1] Until recently, LAIK could only use MPI [6] to facilitate communication between the different instances of the SPMD program, but in this work we are presenting a new communication backend for LAIK, based directly on native TCP sockets [15] provided by the operating system.

The basic idea behind LAIK is simple: While the programmer of an MPI based application has to distribute work and initiate the necessary communication himself, a programmer using LAIK as an abstraction layer does not need to care about these two. Instead the programmer creates a set of data containers within which the individual data units are addressable using simple integer indices. Given such a mapping from integer indices to the actual data, LAIK can now partition the data and distribute it to the available worker processes automatically. Furthermore, since LAIK actually knows and manages the data distribution, it can shift away work from a node expected to fail, or keep redundant copies of the data in order to deal with unexpected failures. [47,48]

In this work however, we mostly ignore the automatic work distribution and possible fault tolerance of LAIK and instead focus on the feature which makes these two possible in the first place, i.e. communication abstraction. From this point of view, LAIK can be thought of as primarily being an abstraction library for different data exchange mechanisms supported by its backends. As mentioned, LAIK only supported a single backend until recently, namely the MPI backend shown in figure 1.1. In fact, the functionality provided by LAIK is not unlike that provided by MPI itself, namely simplifying inter-process communication for the programmer. In contrast to MPI however, LAIK does this by providing a much higher abstraction level to the programmer.

Since LAIK provides such a high-level abstraction layer, adding new communication backends besides MPI seems like the logical next step: We can test whether implementing LAIK's backend API (see figure 1.1) is suitable, bring LAIK to systems where using MPI is not possible, and see how much the backend matters in terms of performance by comparing multiple backends. Therefore, as mentioned above, this work will introduce an implementation of LAIK's communication abstraction using plain TCP sockets, and then answer the *research question* of how the performance of our new TCP backend compares to that of the existing MPI backend in combination with a widely used MPI implementation.

[1]For the purpose of this work, whenever we are referring to LAIK, we implicitly mean the version of LAIK contained in commit d2ee62e1d84cfd2139a0ab9a68d07aec740f90b9 in LAIK's git repository.

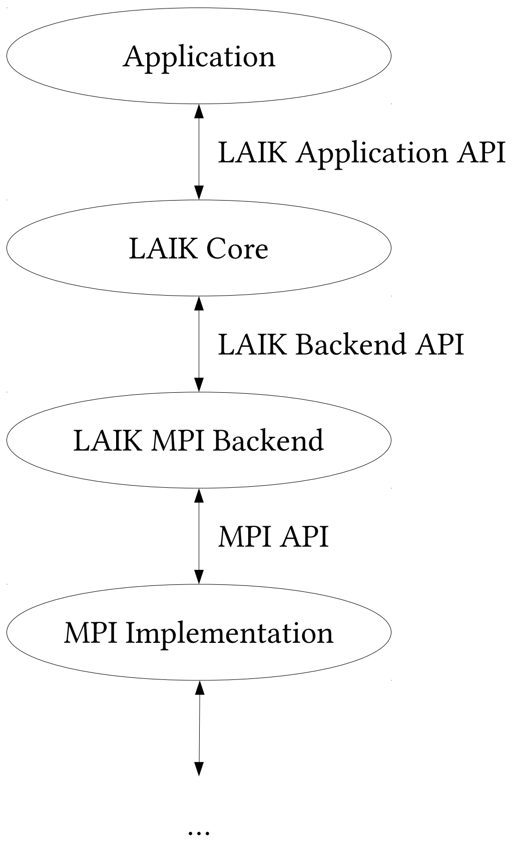


Figure 1.1: Stack overview of LAIK when the MPI backend is used

2 Motivation

Adding a new TCP backend to LAIK may seem unnecessary at first: Even the first version of the MPI standard [5] explicitly expected implementations on top of "standard Unix inter-processor communication protocols" which could work in "heterogeneous networks". Consequently many existing MPI implementations such as MPICH [30] and OpenMPI [44] support using standard IP-based networks as their transportation medium. They do not necessarily use TCP to transport the data (more on that in section 3.2), but nevertheless, they do work across the network. So then why bother with a new TCP backend for LAIK?

1. The new backend talks directly to the operating system and thus does not require an intermediary to facilitate inter-process communication. This means that in contrast to the existing MPI backend, we do not require a working MPI implementation on the target system, but just an operating system implementing the BSD Sockets API, i.e. anything that implements the POSIX standard [12].

2. MPI implementations which use the network for transportation may make certain assumptions regarding latency or throughput which some networks may not be able to satisfy. However, since the machines in these networks typically still need to use at least some network services and most network services nowadays use TCP sockets, it is very likely that even in those special-case networks, a working implementation of TCP sockets is available.

3. Since the new TCP backend can talk to the operating system directly, it is possible that leaving out the intermediary MPI implementation may actually improve performance.

4. MPI implementations typically react to a failed process by simply terminating the other processes and propagating the error to the user. While there are certain approaches to bring fault tolerance directly to MPI implementations (more on that in section 3.3), LAIK does not currently use or support them. Having full control over the entire stack might allow LAIK and/or the new TCP backend to implement its own error handling and possibly recover from failures gracefully.

All these arguments would also apply to a new backend based on UDP sockets [16] or even raw IP sockets [17]. However, implementing such a backend is significantly harder, as the messages exchanged between between the different instances of a LAIK application may very well exceed the size limits on UDP or IP packets. Therefore, such an implementation would at the very least have to roll its own message reassembly mechanism and most likely also some sort of flow control to avoid overloading the network or the receiver. In summary, we would effectively be re-implementing a bad copy of TCP, which is why we decided against these approaches.

3 Related Work

3.1 The Message Passing Interface (MPI)

The Message Passing Interface (MPI) is an API specification for libraries implementing message passing between a cooperating group of processes, possibly on multiple physical machines. Work on the standard began in 1992 as a shared effort by "parallel computing vendors, computer scientists, and application developers" [6] in order to take the best features from existing message-passing systems and combine them into a single, formalized description. This eventually resulted in the release of the MPI 1.0 standardization document in 1994. Since then, the standard has been continuously improved and extended, with the last release being MPI 3.1 from 2015. [5,6]

It is important to clarify that MPI does not provide an actual library to *implement* message passing, nor a runtime environment which distributes and manages the processes. Instead it gives application programmers an abstract API to exchange data between processes, not knowing how this data exchange is actually implemented. This allows operators of High Performance Computing (HPC) systems to provide MPI implementations specially tailored to their systems, possibly using dedicated communication hardware. At the same time, more generalized implementations using primitives provided by the operating system are possible, allowing MPI applications to also run on commodity hardware. [5]

While the current version of the MPI standard [6] defines literally hundreds of functions, the basic concepts are very simple. Assuming MPI is used for facilitating the communication of a SPMD application, the process typically consists of the following steps:

1. Distribute the application binary to one or more compute *nodes* and spawn one or more *processes* per node. This step is often done by a runtime wrapper program provided by the specific MPI implementation the application was compiled against. We will refer to the set of all such processes (on all nodes) as the *process group* from now on.

2. Call the MPI initialization function [31]:

```
int MPI_Init (int* argc, char*** argv)
```

   This function initializes the MPI implementation and returns a status code indicating success or failure (all MPI functions return such a status code). It will typically use information provided by the runtime wrapper program used in step 1, either passed in via environment variables or as command line arguments (in which case these will be filtered out from `argc` and `argv` after the call).

3. Communication between processes happens using opaque *communicator* objects. After the initialization in step 2, the first such communicator available as the predefined constant `MPI_COMM_WORLD` is ready to be used. The application may now inspect the communicator to determine the *size* of the process group using this function [32]:

```
int MPI_Comm_size (MPI_Comm comm, int* size)
```

   After calling this function, `*size` will contain the number of processes in the specified communicator. Furthermore MPI implementations have to provide applications with a total ordering of all processes whereby each process gets assigned a unique *rank* using this function [29]:

```
int MPI_Comm_rank (MPI_Comm comm, int* rank)
```

Assuming there are a total of n processes, each process will find its unique rank r in `*rank` after this function returns, with $r \in \{1..n\}$.

4. Using the information gathered in step 3, the application may start to exchange data using these two functions [33,34]:

```
int MPI_Send (const void* buf, int count, MPI_Datatype datatype, int dest, int
    tag, MPI_Comm comm)
int MPI_Recv (void* buf, int count, MPI_Datatype datatype, int source, int tag,
    MPI_Comm comm, MPI_Status* status)
```

`MPI_Send` sends the data provided in `*buf` (using the length and type information provided) as a *message* to the processes with rank `dest`. Conversely, `MPI_Recv` receives a message from the process with rank `source` into the provided buffer.

5. After using the functions from step 4 for a number of times, the application has to explicitly signal termination to the MPI implementation using this function [35]:

```
int MPI_Finalize (void)
```

Like `MPI_Init`, this function is entirely implementation dependent. It may be a simple no-op, but it may also be crucial to make sure that locally buffered messages are properly flushed. Therefore, the application may not terminate before calling this function.

There are a number of additional functions which most applications will typically use, such as the functions used to duplicate and/or modify the communicator objects [36,37], and the functions for shared reductions [38,39]. However, the functions here show the fundamental workflow of using MPI for data exchange and more importantly introduce the basic terminology used in MPI as well as throughout this work:

- *process*: A single instance of a MPI application, running on a specific node.
- *node*: A physical or virtual system running one or more processes.
- *process group*: The set of all processes on all nodes.
- *communicator*: An opaque object representing a subset of the the process group, with `MPI_COMM_World` representing the whole process group.
- *size*: The number of processes which are part of a specific communicator.
- *rank*: A number from $\{0..size - 1\}$; each process gets a different (and thus unique) rank.
- *message*: A data buffer sent from one process to another.

The first implementation of the MPI standard was MPICH [30] which was started during the initial standardization process as reference implementation to provide feedback on the emerging standard [40]. Another popular general-purpose implementation is OpenMPI [44] which we are going to use later on during the evaluation. However, as mentioned above, the MPI standard was explicitly designed to allow HPC operators to provide implementations which could fully utilize their specific hardware, so there are quite a few vendor-specific implementations out there.

3.2 MPI Network Performance

At the end of chapter 2 we shortly discussed the idea of using UDP or even raw IP sockets as communication primitives, but discarded the idea since we would have to re-implement many of the features TCP provides us for free. It turns out that [9] had a similar idea, but actually took it a step further: Since HPC clusters are often directly connected via Ethernet (without any routers in between), they extended the Linux kernel with a new socket family [18] called PF_ENET. This socket family provides both an unreliable datagram protocol similar to UDP as well as a data streaming protocol similar to TCP, but operates directly on layer 2, entirely bypassing the existing TCP/IP stack of the operating system.

The data streaming protocol of the new PF_ENET socket family, called Ethernet Streaming Protocol (ESP), is quite interesting: While they re-implemented most of the basic features of TCP such as sequences numbers, acknowledgments, and retransmissions, they only added 13 bytes to the existing Ethernet header for this, thus significantly reducing the network overhead. While this approach has severe limitations since the resulting network frames can not be routed, it provides clear advantages on networks where routing and other features offered by the conventional TCP/IP stack are not needed.

With their new Ethernet based protocol in place, they went ahead and extended OpenMPI [44] to use it and evaluated the performance of their approach in comparison to the existing TCP/IP solution provided by OpenMPI. They found that they could significantly reduce the latency and CPU overhead and in one case even halve the runtime of an example application which mostly exchanged very small messages between all processes. They do note however that a proper flow control algorithm still needs to be added, as larger messages caused a slight performance degradation.

We think that it is interesting to see the idea that we discarded early on actually implemented. However, their closing remarks confirm our hesitation, as they acknowledge that re-implementing TCP is non-trivial and and an incomplete re-implementation may actually perform worse in some circumstances. Finally, with their results in mind, we expect our TCP backend to be at a disadvantage when it comes to sending a lot of very small messages.

3.3 MPI Fault Tolerance

Without special measures, a single failed process usually brings down the entire process group when using MPI for inter-process communication, as shown in the following example which uses a LAIK application making use of LAIK's MPI backend:

```
$ LAIK_BACKEND='mpi' mpirun -n 2 ./jac3d 500
500 x 500 x 500 cells (mem 2000.0 MB), running 50 iterations with 2 tasks
Residuum after  1 iters: 377335777.222662
Residuum after 11 iters: 506561.854358
Residuum after 21 iters: 99032.165750
Residuum after 31 iters: 54811.260127
-------------------------------------------------------
Primary job  terminated normally, but 1 process returned
a non-zero exit code. Per user-direction, the job has been aborted.
-------------------------------------------------------
--------------------------------------------------------------------------
mpirun noticed that process rank 1 with PID 0 on node shepard exited on signal 9 (
    Killed).
--------------------------------------------------------------------------
$
```

Listing 3.1: When one process in the MPI process group is manually terminated, mpirun forcefully terminates the remaining processes

For understanding the concept of fault tolerance in the context of MPI, [8] provides an excellent overview. They explain that the MPI standard [6] mandates that the default error handler attached to the initial communicator MPI_COMM_WORLD is MPI_ERRORS_ARE_FATAL which causes all processes in the process group to abort if any one of them exits before calling `MPI_Finalize()`. However that does not mean that the MPI standard does not permit fault tolerance, it is just not the default. Consequently, [8] defines fault tolerance to be a property of the $(MPI\ implementation,\ MPI\ program)$ tuple which is fulfilled if the tuple can survive failures of some processes, for various definitions of "survive".

Leaving aside the approaches which require modifying or extending the MPI standard, [8] presents two ways in which an MPI implementation can gain fault tolerance: It can either periodically save its state to a reliable storage medium (checkpointing) and use these saved states to quickly recover when restarted after a failure. Alternatively it can be written in a way similar to classical client-server applications where a central manager process dispatches chunks of work to individual worker processes. Since the communication then only happens between the manager and the worker processes, a failed worker does not affect the other workers and the manager can simply hand the unfinished piece of work to some other worker.

However, both approaches require that the MPI implementation can correctly detect and report communication failures when being asked to do so (this is normally done by setting the `MPI_ERRORS_RETURN` error handler on the `MPI_COMM_WORLD` communicator immediately after startup). According to [8], not all MPI implementations can do this, at least not always. For example, the Intel MPI Library only supports returning error codes instead of aborting the whole process group if the special environment variable `I_MPI_FAULT_CONTINUE` is set to `1` [13].

Unfortunately, the approach of having a central manager process and several worker processes which only ever talk to the manager is unsuitable for many applications, so checkpointing is the only viable solution for them. [8] differentiates between two forms of checkpointing: User-directed checkpointing means that the programmer is solely responsible for serializing the program state and writing it to storage, while system-directed checkpointing hands this task over to the MPI implementation (at least partially). Even though user-directed checkpointing has some drawbacks[1], [8] recommends it over system-directed checkpointing, as the latter is much harder to implemented correctly because a program's state is typically scattered in many locations.

Nevertheless, there are some approaches to bringing system-directed checkpointing to mainstream MPI implementations, such as [10], [11], and [45]. However, all of them require at least some cooperation from the program using the MPI API which is not surprising given that the MPI implementation generally does not have enough information about the specific program to capture its state entirely or set a consistent program state on restart after failure. This is where LAIK's higher level of abstraction comes into play: Since LAIK applications actually declare all their state to LAIK, creating and restarting from checkpoints should be much easier.

In this work however, we are not going to use this feature of LAIK, but rather rely on the fact that LAIK proxies all communication requests. To be more precise, we are going to make use of the deterministic nature in which LAIK schedules communication through its backend: While a regular MPI implementation must be able to deal with applications which just request the next available message using the `MPI_Probe()` [41] function or the special `ANY_SOURCE` argument instead of a concrete rank in `MPI_Recv()` calls, the MPI calls emitted by LAIK will always have a concrete sender and receiver and will always be issued in a predicable order.

[1] [8] lists the facts that the programmer must take care of reliably storing the serialized data and must also make sure that no state is currently stored in in-flight messages. These constraints severly restrict the points and frequency at which checkpoints can be made.

4 Design

In this chapter we introduce the design for LAIK's new TCP backend. We first decide how to best interface with the existing code. Second, we look at the expected challenges the design has to handle. Then we formulate the basic design principles in the third part and give an overview of the network protocol in the fourth part. Finally, we explain how the presented design addresses the formulated challenges.

4.1 Interfacing with the Existing Code

Extending LAIK with a new communication method seemed straightforward at first, given that LAIK has a dedicated backend interface for that purpose: Get an understanding of how LAIK talks to its communication backend and what it expects it to do, have a look a what the existing MPI backend does to implement these requirements, and then re-implement them using standard TCP sockets. However, when we actually looked at code, we discovered some problems with this approach:

1. As the MPI backend was the only existing backend[1], it was not surprising that the backend interface is clearly influenced by the MPI API in several places. The most visible such influence is that LAIK does not simply hand a binary blob to the backend along with instructions on where to send it, but instead actually has a run-time type-system which the backend has to use in order to figure out what the data buffer attached to send and receive operations actually contains.
2. Since the MPI backend is fully synchronous, LAIK's backend interface expects to be able to re-use or release buffers passed to the backend after the backend returns. This implies that the new TCP backend either has to also work completely synchronously or that it has to create a memory copy of the data passed to it if it wants to decouple the data hand over to the backend from the actual sending and receiving.
3. The MPI backend does significantly more than exchanging data between processes: It also discovers the size of the process group, the rank of the local process and synchronizes changes in LAIK's view of the process group with the backend's internal data structures and those used by the MPI implementation. This implies that e.g. removing a process from the process group is something the backend has to be aware of and support.
4. LAIK supports a wider variety of reductions than plain MPI[2], but still wants to use the potentially faster, native MPI reduction functions if possible. Therefore reductions are expected to be handled entirely by the backend. Consequently, the MPI backend contains a implementation of a distributed reduction algorithm *in addition* to wrappers around the corresponding MPI functions and would inspect the reduction task closely to decide what alternative to use.

In addition to the problems described above, LAIK's backend interface was poorly documented, had no stability promises and no unit tests. Fortunately however, the existing MPI backend only

[1]There was also a dummy, single-process backend, but this backend essentially just consisted of stub methods.
[2]For example, LAIK allows reductions where only a subset of the process group takes part or where a custom reduction function is needed.

used a small subset of the MPI functions offered by the MPI standard. In essence, these were the used MPI functions:

1. `MPI_Init()` and `MPI_Finalize()` to respectively initialize and finalize the MPI subsystem and give it access to the programs command line arguments.
2. `MPI_Get_processor_name()` [42], `MPI_Comm_rank()`, and `MPI_Comm_size()` to determine where a process was running and which rank it occupied in the entire process group.
3. `MPI_Comm_split()` [37] to update the MPI subsystem of any changes LAIK wanted to make to the process group such as removing a process.
4. `MPI_Reduce()` [38] and `MPI_Allreduce()` [39] to run reduction operations natively inside the MPI implementation if possible.
5. `MPI_Send()` and `MPI_Receive()` to synchronously send and receive data buffers from other processes and to implement the software reduction algorithm mentioned above when the native MPI reduction functions were incapable of running the requested reduction.

Given that these functions were well documented by the MPI standard, had a strong stability promise and well defined semantics, and where significantly closer to the low-level BSD sockets API than the high-level LAIK backend API, we decided to re-implement these MPI functions instead of following our original approach of using LAIK's backend interface. This approach allowed us to initially re-use the existing code in the MPI backend, including the non-trivial software reduction algorithm, even tough we were in fact using TCP sockets for the actual data transmission. While we did eventually write a new backend for LAIK's backend API, this approach still proved invaluable as it allowed us to quickly get a working prototype so we could discover and eventually solve the problems outlined in this chapter.

4.2 Expected Challenges

In the section 4.1, we listed the subset of MPI functions used by the MPI backend and determined that we would re-implement them rather than use LAIK's native backend API to quickly get a working prototype. Since we wanted to use TCP for actually transmitting the data, it seemed clear that the most important functions to emulate would be the `MPI_Send()` and `MPI_Receive()` functions as the remaining functionality could then be expressed in terms of these functions. In theory, this seemed simple: Create a new TCP socket using `socket()` [19], and then implement the two MPI functions using the BSD Socket API by making the following system calls:

- `MPI_Send()`: `connect()` [20] and `send()` [21]
- `MPI_Receive()`: `bind()` [22], `listen()` [23], `accept()` [24], and `recv()` [25]

In practice however, just doing this simple replacement is not enough because of a few obvious challenges which we have to solve first.

4.2.1 Memory Exhaustion

To illustrate the first problem, let's consider a $N : 1$ reduction where N processes send their reduction input to a single, dedicated reduction process as shown in figure 4.1

We assume that the reduction function is not necessarily commutative (meaning that all but one reduction order would give an incorrect result) and that the node running the dedicated reduction process does not have enough memory to hold all reduction inputs in memory at the same time. Figure 4.2 shows problem: If the inputs arrive out of order, the reduction fails since

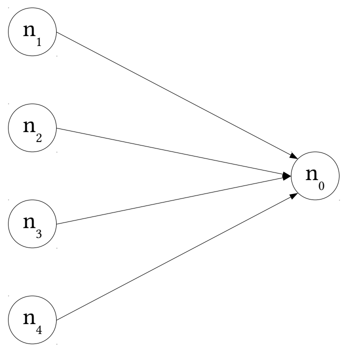


Figure 4.1: $N : 1$ reduction where nodes n_i ($i \in \{1, 2, 3, 4\}$) send data to node n_0

the dedicated reduction process can neither consume the next input (and thus free the memory occupied by it) nor receive another part of the input (because it has no memory to store it in).

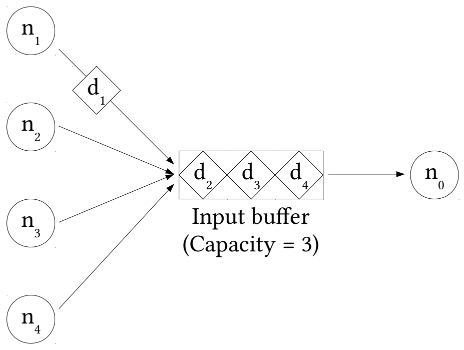


Figure 4.2: $N : 1$ reduction where nodes n_i send data d_i ($i \in \{1, 2, 3, 4\}$) to node n_0 and exhaust its memory

4.2.2 Connection Exhaustion

The problem described in section 4.2.1 boils down to the fact that on $N : 1$ communications, the messages might arrive out of order at the recipient and it might not have enough memory to store all messages so it can process them in order. A possible solution to this problem would be to not store the messages at the recipient, but to instead block the sender until the recipient is ready to receive their message as seen in figure 4.3

With TCP sockets, blocking the sender is easily achievable: The receiver can just accept the new connection from the sender, determine if it is currently interested in this connection's data, reading the data if so and storing the connection for later otherwise. However, this means that both the sender and the receiver would have to keep open the connection until the data can actually be transmitted. In the worst case (where the messages arrive exactly in the reverse of the required order), this means the receiver of a $N : 1$ communication has to keep open N TCP sockets concurrently.

Since the maximum number of concurrently open TCP sockets is limited both per application (as the operating system typically only allows a certain number of open file descriptors per

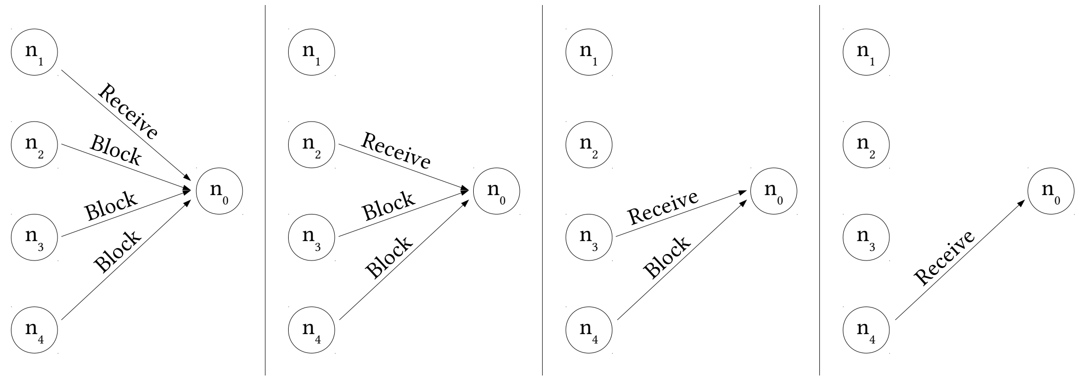


Figure 4.3: $N : 1$ reduction where nodes n_i send data to node n_0 and are blocked until needed

process[3]) and per host (since each new connection requires OS resource to handle), it is possible, given a sufficiently large N in the $N : 1$ communication that the receiver can not accept enough new connections to actually get to the connection from the first sender it wants to receive from.

It might seem possible to avoid this problem by using *reverse* connections where the receiver connects to the sender to fetch the requested data instead, thus only needing one concurrently open TCP socket at a time (the one to the sender holding the data the receiver is currently interested in). However, this approach breaks on $1 : N$ communications where a single sender sends to multiple receivers: The potential receivers would then all connect back to the single sender, and unless these connections arrive in exactly the right order, it now has the exact same problems as the receiver in the previous paragraph.

4.2.3 Unfair Connection Acceptance

In section 4.2.1 and 4.2.2 we have shown that the naive approach of simply sending messages when they are ready potentially overwhelms the receiver: Either the receiver can exhaust the available memory while trying to buffer the messages so it can read them in a specific order or it can reach the maximum number of open connections allowed by the operating system while trying to accept connection after connection until it finds the next required message. The logical consequence is that the receiver must always be able to decline incoming messages and signal the sender that it is currently not ready to receive that particular message.

A simple but feasible solution is therefore that the sender prepends the message with some metadata (e.g. a sender rank, message length, etc.) which the receiver can use to decide whether it currently wants to accept and store this message or not. However, this implies that the sender has to handle the case where the receiver signals that it did not accept the message. The obvious solution here is to have the sender retry the transmission after a short time, and as long as the sender opens a new connection for every message, this actually works: The FIFO queue provided by the operating system for listening sockets guarantees that every new connection (and thus every message) eventually gets inspected by the receiver. Therefore the first required message will eventually be received, then the second, and so on.

There is just one problem: Creating a new TCP connection for every message is prohibitively expensive, requiring several system calls on both sides and more importantly a significant number of network round trips. For this reason, many TCP based network protocols supporting

[3]For example, on Linux this number defaults to 1024.

delivery of individual message can re-use established connections, e.g. HTTP [4], SMTP [46], and SSH [1]. As we want to be able to send *many* messages and explicitly also want to support medium- and high-latency networks, supporting connection-reuse is mandatory.

However, with connection-reuse there is now more than one socket which can potentially hold a new message, one listening socket and an arbitrary number of established connection sockets. This means that the receiver has to actively decide which socket to inspect when it wants to receive the next message. This decision is non-trivial: If the receiver always prefers the listening socket (and there are enough new connections per second), it might never get to the established connection socket which holds the message required next. Conversely, if it always prefers processing new input on an established connection over accepting a new connection from the listening socket, it might never accept the new connection which brings that long-awaited next message, at least not if the senders on those connections are fast enough in re-trying (and failing) to deliver their unwanted messages.

4.2.4 Message Identification

From the challenges described in sections 4.2.1, 4.2.2, and 4.2.3, we can conclude that we will need the ability to initially decline messages and only receive them at a later time when the we actually want to process them. Furthermore, it seems clear that we want at least some level of message buffering on both the sender and receiver side as unbuffered sending and receiving would imply that the whole application will frequently stall while waiting for network I/O. From this, we can conclude that we will need some sort of identification mechanism for an individual message.

Unfortunately, the MPI API provides no way for the application to uniquely identify a single message. Instead, send and receive calls are implicitly matched by the order in which they are issued on the sender and receiver side. So, in order for message buffering and retransmission to work, we have to find a way to generate unique *message identifiers* internally while still exposing the MPI API externally.

4.2.5 Implicit Global Serialization

A fundamental problem experienced by every programmer attempting to use MPI or a similar synchronization protocol is making sure that every message send call eventually finds a matching receive call on the other side so that communication deadlocks can never happen. A naive solution for such an algorithm could look like this:

```
for (int sender = 0; sender < group_size; sender++) {
    if (sender == my_rank) {
        // It's our turn to send data
        for (int receiver = 0; receiver < group_size; receiver++) {
            if (receiver != my_rank) {
                send_message (receiver, data, ...);
            }
        }
    } else {
        // It's not our turn to send, instead we should receive data from the current
          sender
        receive_message (sender, data, ...);
    }
}
```

Listing 4.1: A naive send/receive ordering algorithm which prevents deadlocks but effectively serializes the message exchange among all processes

While this code avoids loops in the dependencies of the execution order of the various send and receive calls, we have just introduced a serious performance problem: For all but the process with rank 0, sending data can only begin once one or more messages from another process were sent, transmitted over the network, and received locally. The process with the rank $group\_size - 1$, will even only start sending its *first* message once *all* other messages *in the entire network* were sent and received.

This might not seem like a big problem as long as the latencies and the number of processes are sufficiently low. However, this is effectively a global serialization as process N will only start sending data after process $N - 1$ has completed doing so. Therefore the total time t_{total} required for a $N : N$ communcation scheme (in which every process sends a message to every other process) can be estimated using the following equation whereby t_{min} is the minimal delivery time for a single message:

$$t_{total} >= N \cdot t_{min} \tag{4.1}$$

This is obviously unacceptable if we want to deliver reasonable performance on high-latency networks or with a lot of process in the process group. Therefore, we have to find an improvement to the naive algorithm described above while still retaining its guarantees regarding deadlock-avoidance.

4.2.6 Peer Discovery and Rank Assignment

A significant service provided by MPI implementations to applications is answering the following questions, typically on start-up:

1. What is the size of the process group?
2. What is my rank in the process group?
3. How can I contact the process with rank r?

Our backend and thus the MPI compatibility layer we are building also needs to be able to answer these questions. However, there is a chicken-and-egg problem here: Unless we answer these questions first, we cannot communication with other processes in our process group, but we have to communicate with the other process in order to determine how many processes there are and what rank they have chosen.

The obvious solution here is to statically provide the answers to these question to the process on startup, for example via environment variables. Alternatively we could have a dedicated *management* process which all other process contact on startup to answer the questions formulated above. However, both approaches are less than ideal: The static nature of the first prevents later on adding or removing processes or modifying the process group in any other way. The second implies having a dedicated management process which could be a potential performance bottleneck at least on startup. Furthermore, it still requires a supplying static piece of information (how to contact the management process) to all processes on startup.

4.2.7 Fault Tolerance

Message buffering in combination with a message identification mechanism suggests an interesting improvement on top of the existing semantics MPI implementations provide: As shown in section 3.3, for these implementations the abnormal termination of a process in the process group (for example because of memory exhaustion or power failure), usually means that the remaining processes in the group can no longer complete the computation, and will therefore (potentially after some timeout) also terminate with an error. Now, it is of course possible to restart the single failed process (possibly on some other computation node), but this will not help much because of two fundamental problems with the MPI API and its implementations:

1. As mentioned in section 4.2.4, the MPI API provides no way for the application to uniquely identify a single message. Since no such identification is possible anyway, MPI implementations typically do not retain messages after they have been successfully delivered to the intended recipient.
2. In contrast to lower level network protocols, MPI implementations typically cannot deal with duplicated messages correctly, since they have no way to identify such duplicates. Instead, a sender which for some reason lost its state and therefore re-sends old messages will typically cause errors on the receiver's side as the received messages will not be what it expects according to its own state.

Now, if we have to implement message buffering and unique identification anyway, these problems suddenly seem to be low-hanging fruits, so it would be interesting to see if we can actually solve them while still only using the MPI API to communicate with LAIK and the actual application.

4.3 Basic Design Principles

With the problems described in the section 4.2 in mind, we came up with a design following five basic principles:

1. Peer discovery as well as rank determination should work via a simple configuration file: Each process would read this configuration file on startup (and refresh it periodically later on) and extract an ordered list of $(hostname, port)$ address tuples. It would then try to bind a socket to each of these addresses, stopping at the first successful one. This scheme allows the process to easily determine its own rank among the process group (by looking at the index of the address it could bind successfully) as well as the total size of the process group (this is identical to the total amount of address tuples found in the configuration file). Furthermore, the basic semantics provided by the operating system when binding a socket to a name ensure that no two processes can arrive at the same conclusion regarding their own rank. Finally, contacting the process with rank r is just a simple array lookup.
2. Since data transmission should work via TCP sockets and frequently establishing new connections is expensive (especially for very small messages), connections should be aggressively reused. This means that after a message has been successfully delivered, both the sender and the receiver must keep the connection open for some time and be able to reuse it for later messages.
3. Sending and receiving a message should be done fully buffered and mostly asynchronous to improve performance. This implies that each process needs a buffer for outgoing message (called *outbox* from now on) and a buffer for incoming messages (called *inbox* from now on). In order to avoid memory exhaustion, both these buffers must have a maximum size, upon which adding new messages should either block or fail. Furthermore, it must of course be possible to retrieve and/or remove a specific message (identified by its *message identifier*) from these buffers, making room for new message to be stored there. Finally, it should be possible to send and receive multiple messages in parallel, so all the data structures involved must be fully thread-safe.

4. When sending a message, the sender should be able to submit both messages which must be accepted as well as messages which may be dropped by the receiver if it currently has no room to buffer them for later use. However, since the sender has to know whether it can remove a message from its own buffer or not, the receiver has to signal the sender if it currently can not store a message. Furthermore, the receiver must be able to contact the sender later on to signal that it is now interested in the message it previously refused.
5. All TCP sockets should be created with the O_NONBLOCK [26] and TCP_NO_DELAY [15] options enabled. The first option avoids endlessly blocking in a `send()`/`recv()` call, instead we can use `poll()` [27] to wait for the socket to be ready or a timeout to occur which allows us to quickly detect network errors. We can then either retry the data transmission with a new connection or propagate the error up, eventually aborting the program. The second option disables buffering small data segments before transmitting them over the network in one big chunk. While this behavior is desirable when doing bulk data transmissions, we want our messages to arrive with as little latency as possible to avoid the stalling the receiver.

4.4 Network Protocol

We designed a protocol based upon a classical client-server scheme and three different request types explained below. The basic idea is that whenever a process wants to initiate communication with another process, it either creates a new connection or reuses an existing connection to that process, thus assuming the role of the client. The other process either accepts the new connection or notices the new input data on an existing connection, thus assuming the role of the server. Once the connection is ready, the client sends one of three little endian 64-bit unsigned integers, indicating the request type, as shown in figure 4.4.

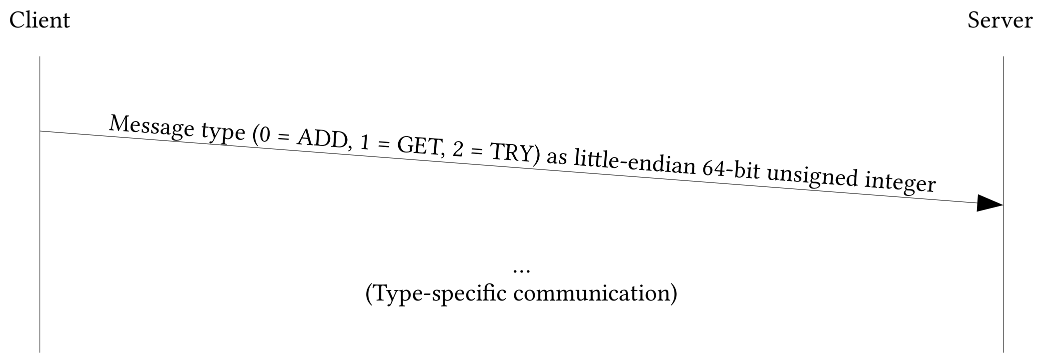

Client
Server
Message type (0 = ADD, 1 = GET, 2 = TRY) as little-endian 64-bit unsigned integer
...
(Type-specific communication)

Figure 4.4: Submission of the request type from the client to the server

First, the clients submits the type of the request to the server; either 0 (i.e. an *ADD* request), 1 (i.e. a *GET* request), or 2 (i.e. a *TRY* request). What happens next depends on the concrete type of the request, but once the request handling is completed, both client and server push the connection back to a connection pool so it can be reused for later communications, assuming no errors occurred while processing the request.

4.4.1 ADD Requests

ADD requests are a form of communication whereby a client can submit a message to the server and be sure that the server accepts and stores the message, no matter what. Having such a request type may seem surprising at first, since it clearly allows overloading the receiver by making it exhaust its available memory. However, some scenarios require that a message is actually delivered once the method accepting the outgoing the message on the sender side returns, instead of just being buffered in the list of outgoing messages. The most obvious example (and indeed the only place where this message type is used in our implementation) is synchronization before shutdown: Before terminating, the program has to make sure that no other process still needs a message stored in its outbox. ADD requests can be used to accomplish this using a simple "Shutdown OK?"/"Shutdown OK!" protocol between the processes.

Figure 4.5 shows the data transmitted over the network for an ADD request. Every arrow represents one `send()`/`recv()` call (or multiple if the operating system accepts/returns less than the requested amount of bytes). Furthermore, since we set the TCP_NODELAY option, it is very likely that each `send()` also directly translates into a dedicated TCP segment.

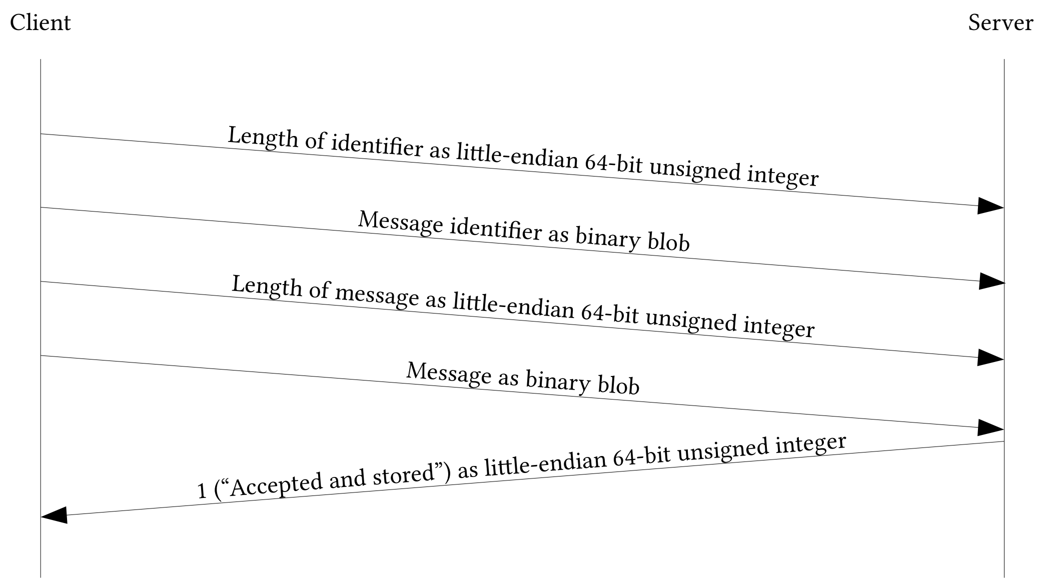

Client
Server
Length of identifier as little-endian 64-bit unsigned integer
Message identifier as binary blob
Length of message as little-endian 64-bit unsigned integer
Message as binary blob
1 ("Accepted and stored") as little-endian 64-bit unsigned integer

Figure 4.5: Communication flow for a ADD request

The client first sends the message identifier and then the actual message, each prefixed with their length as little endian 64-bit unsigned integers. Since the server has no choice here whether it accepts the message or not, it will always respond with a single little endian 64-bit integer with a value of 1, indicating that it successfully received and stored the message.

4.4.2 TRY Requests

TRY requests work almost like ADD requests, except that here the server can refuse a message and indicate to the client that it should keep the message in its outbox for later retrieval. An accepted TRY requests is shown in figure 4.6.

In contrast to ADD requests which should only be used to send very small control messages to minimize the risk of overloading the receiver, TRY requests are allowed to carry large message,

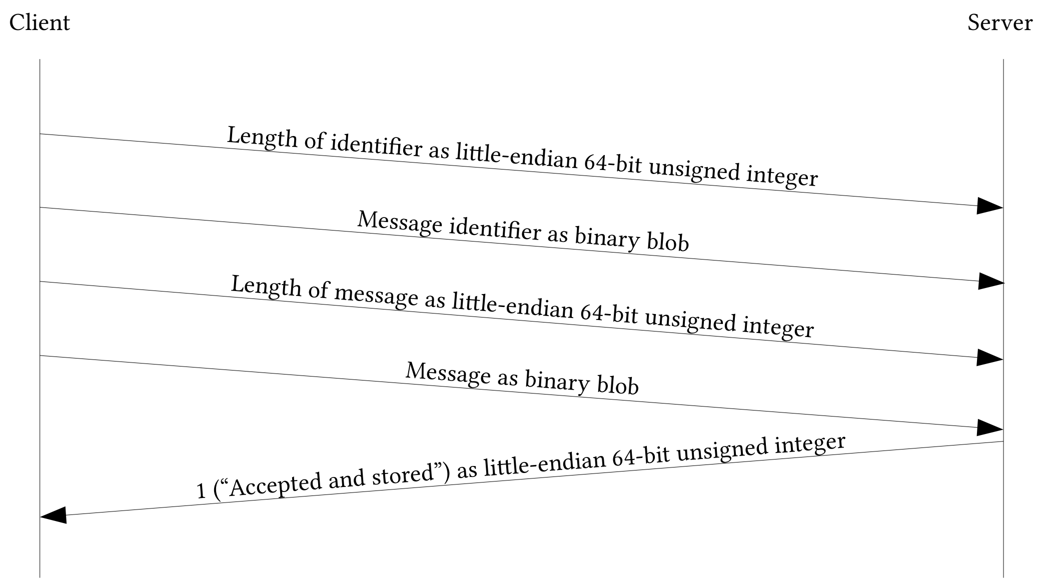

Client
Server
Length of identifier as little-endian 64-bit unsigned integer
Message identifier as binary blob
Length of message as little-endian 64-bit unsigned integer
Message as binary blob
1 (“Accepted and stored”) as little-endian 64-bit unsigned integer

Figure 4.6: Communication flow for an accepted TRY request

potentially too large to fit in the receivers inbox. In such a case, the server will refuse the request, as shown in figure 4.7.

It is worth noting that the server bases the decision on whether to accept or refuse a message solely on the size of the message in relation to the remaining space in the inbox. This directly implies that messages which are too large to fit in the inbox, can not be delivered via TRY requests, as they will always be refused. Furthermore, TRY requests are never retried: If a TRY requests is refused or fails for some other reason, the corresponding message simply remains in the outbox and no further action is taken.

4.4.3 GET Requests

The primary means of communication used to submit messages from the sender to the receiver is the TRY request introduced in section 4.4.2. However, this request may fail for a variety of reasons: The receiving process may not yet have started, establishing a new connection or submitting the data may time out, or the message may simply be too large to currently fit into the receivers inbox. In these cases, the receiver can also assume the role of the client and retrieve the message from the sender via a GET request, reversing the direction in which messages are transported.

Figure 4.8 illustrates the communication flow of a successful GET request: The receiver sends the identifier of the message it is interested in to the sender and the sender responds with a successful status code and the actual message. However, it is of course possible that the receiving process needs a specific message before the sending process has reached the point where this message is generated and submitted to the backend. In this case, the designated sender will fail to find the requested message in its outbox and tell the receiver so using a failure status code, as shown in figure 4.9.

It is worth noting that a GET request may fail for other reasons as well: For example the sender may have already successfully submitted the message with a TRY request to the receiver and consequently removed the message from its outbox, before it came around to processing the

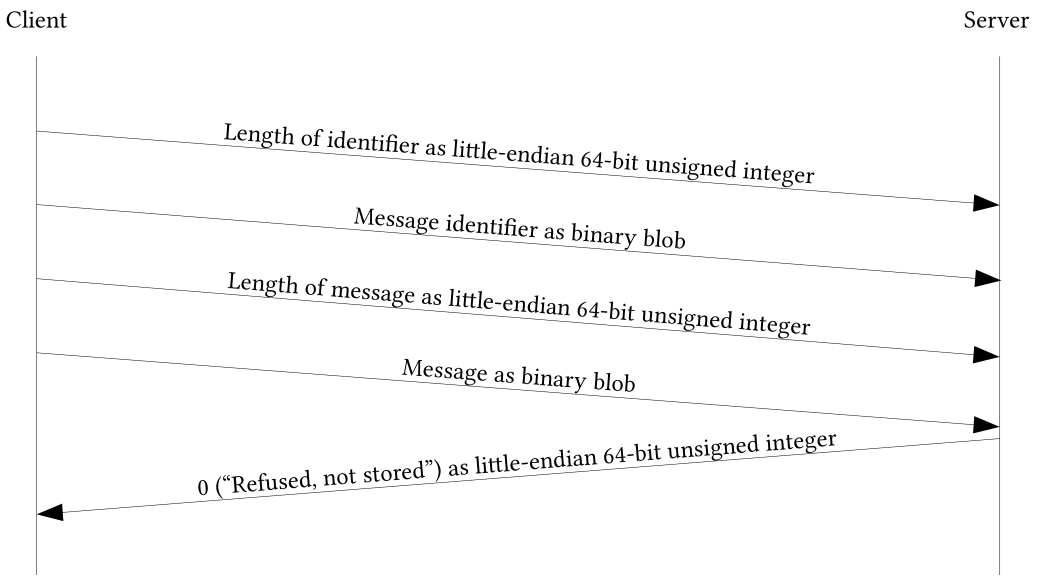

Client
Server
Length of identifier as little-endian 64-bit unsigned integer
Message identifier as binary blob
Length of message as little-endian 64-bit unsigned integer
Message as binary blob
0 (“Refused, not stored”) as little-endian 64-bit unsigned integer

Figure 4.7: Communication flow for a refused TRY request

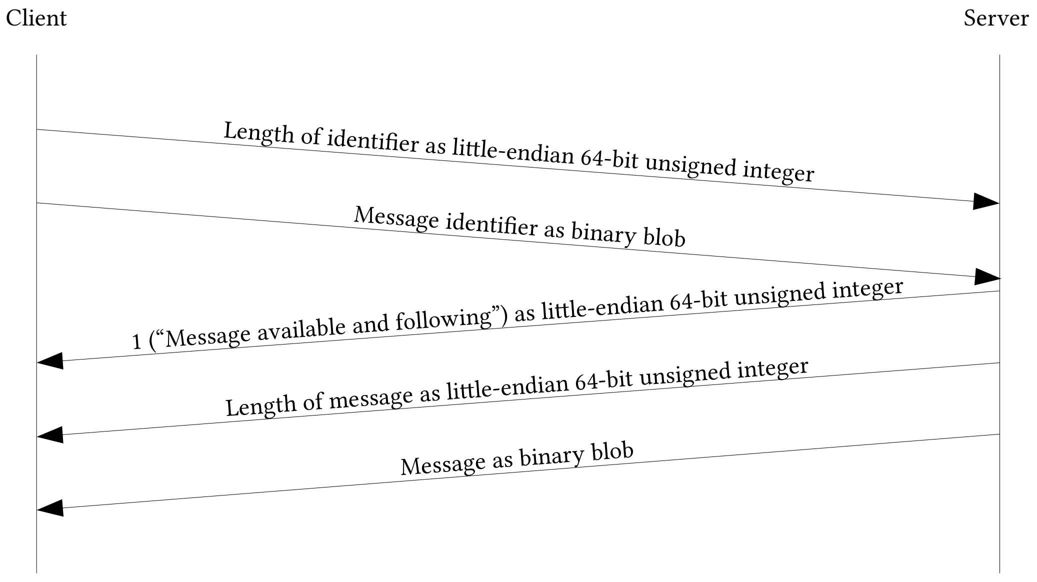

Client
Server
Length of identifier as little-endian 64-bit unsigned integer
Message identifier as binary blob
1 (“Message available and following”) as little-endian 64-bit unsigned integer
Length of message as little-endian 64-bit unsigned integer
Message as binary blob

Figure 4.8: Communication flow for an accepted GET request

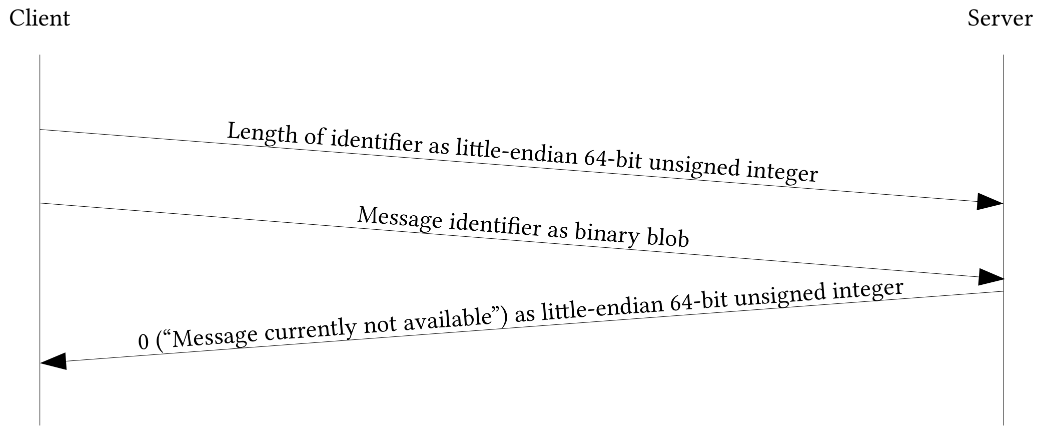

Client
Server
Length of identifier as little-endian 64-bit unsigned integer
Message identifier as binary blob
0 ("Message currently not available") as little-endian 64-bit unsigned integer

Figure 4.9: Communication flow for an refused GET request

GET request. It is also possible that the sender has sent multiple GET requests in the mean time which of course means that all but the first will surely fail as the sender removes the message from the outbox after successful delivery. In summary, a receiver must always check its inbox before sending a GET request and it must limit the rate at which these requests are sent.

4.5 Addressing the Challenges

In section 4.3 and 4.4, we have presented our basic design principles and our network protocol. It is now time to take a look back at section 4.2 to see how our design solves the challenges introduced there and what additional properties we gained from these design decisions.

4.5.1 Memory Exhaustion

The memory exhaustion introduced in section 4.2.1 is clearly solved by having maximum sizes for the inbox and outbox. Instead of consuming ever more memory upon a flood of incoming message, we will now simply refuse messages at some point and retrieve them later by having the receiver contact the sender, i.e. by optionally reversing the communication direction. If the outbox exceeds the maximum size, we can simply block in the next call attempting to send a new message (and thus requiring space in the outbox) until the messages in the outbox have been delivered so that enough space is available.

4.5.2 Connection Exhaustion

We do reuse TCP connections in order to avoid the high cost associated with establishing a new TCP connection whenever possible, but avoid the the connection exhaustion described in section 4.2.2 by limiting the maximum number of connections which may be open at any time, both for the client and server side (remember that every process acts as both client and server). Upon reaching the maximum number of connections, we simply close all the connections and then begin again to create or accept new connections.

While more sophisticated cache-eviction strategies would be possible, we decided not to implement them, since their efficiency highly depends on the communication profile of the application and we can not predict that as a library. It is worth noting that we deal gracefully with the situation where only one side decides to close a connection by handling such a connection like any other broken connection, i.e. by closing and removing it from the cache.

4.5.3 Unfair Connection Acceptance

Section 4.2.3 introduces the problem that with connection reuse, the server has to decide between accepting a new connection on the listening socket and handling input on an established connection in a manner which guarantees that every connection is eventually processed. We solved this by using a single FIFO-queue for both the listening and the connection sockets, always taking (and removing) the first ready socket from the queue, running the appropriate action (accept the new connection or process the input on the existing connection) and then adding the originally removed socket (and any new connection sockets) to the end of the FIFO queue. This scheme guarantees that every socket will eventually be the first ready socket in the queue and will thus be chosen to be processed next.

4.5.4 Message Identification

In section 4.2.4 we have determined that a basic requirement for sending and receiving messages in a buffered and possibly out-of-order fashion is having a way to uniquely identify a message across the network. As explained in the previous section, the MPI API provides no way for the application to submit such an identifier, so we have to generate them in the MPI emulation layer we are designing which is what the following code does:

```c
static GBytes* laik_tcp_minimpi_header (uint64_t generation, uint64_t type, uint64_t
    sender, uint64_t receiver, uint64_t tag) {
    laik_tcp_always (flows);

    // We are mutating a global variable here, so let's make sure that there's
    // only ever one thread reading and writing the flows hash table
    static Laik_Tcp_Lock lock;
    LAIK_TCP_LOCK (&lock);

    const uint64_t data[] = {
        GUINT64_TO_LE (generation),
        GUINT64_TO_LE (type),
        GUINT64_TO_LE (sender),
        GUINT64_TO_LE (receiver),
        GUINT64_TO_LE (tag),
        GUINT64_TO_LE (0),
    };

    GBytes* result = g_bytes_new (&data, sizeof (data));

    uint64_t* serial = g_hash_table_lookup (flows, result);

    if (serial) {
        ((uint64_t*) g_bytes_get_data (result, NULL))[5] = GUINT64_TO_LE (++*serial);
    } else {
        g_hash_table_insert (flows, g_bytes_ref (result), g_new0 (uint64_t, 1));
    }

    return result;
}
```

Listing 4.2: The full code of the function generating the message identifiers used throughout the entire TCP backend (located in the MPI compatibility layer, see figure 5.1)

This code is called by all the re-implemented MPI functions which want to send and/or receive data. For each call, the code generates a unique identifier based upon the input parameters and returns it as an opaque, binary blob. The meaning of the parameters is as follows:

- `generation` identifies the number of times the application has called `MPI_Comm_split()` (or `MPI_Comm_dup()`) to modify its communicator object. The original communicator object `MPI_COMM_WORLD` (i.e. the communicator object referring to the set of all processes originally created) has a value of 0, communicator objects directly derived from this one have a value of 1 and so on. Since `MPI_Comm_split()` (and `MPI_Comm_dup()`) generate a new communicator object by splitting the old one into disjunct subsets, this *generation counter* uniquely identifies the communicator object, at least within the process group defined by it.
- `type` states what kind of communication this identifier represents, i.e. whether it is data belonging to a `MPI_Send()`/`MPI_Recv()` call, data exchanged during a reduction initiated by `MPI_Reduce()` or `MPI_Allreduce()`, or perhaps the metadata required for computing the new communicator in an `MPI_Comm_split()` call. This field allows these different kinds of communications to be independent and not interfere with each other.
- `sender` and `receiver` are set to the rank of the sender and receiver within the currently used communicator object. Since both have to be specified, this implies that sending/receiving to/from multiple peers will not work, so for example `MPI_Recv()` calls with the sender specified as `MPI_ANY_SOURCE` are not supported.
- The `tag` parameter directly corresponds to the `tag` parameter in `MPI_Send()` and `MPI_Recv()` calls; other forms of communication simply set this field to zero.

With these parameters, we can uniquely identify a *message flow*, i.e. an ordered set of messages between a specific sender and receiver (within a specific communicator) and with a specific purpose:

$$flow = (generation,\ type,\ sender,\ receiver,\ tag) \tag{4.2}$$

The function shown above then consults a hash table to look up that specific flow and determine how often it was already called for it. The result is a serial number which uniquely identifies a particular message with its flow, leading to the final identifier:

$$identifier = (flow,\ serial) = (generation,\ type,\ sender,\ receiver,\ tag,\ serial) \tag{4.3}$$

Now, if for example a sender with rank 2 issues a total of 3 `MPI_Send()` calls on the `MPI_COMM_WORLD` communicator with a receiver rank of 5 and a tag value of 7, the identifiers created would look like this:

$$identifier_0 = (0, type_{send/receive}, 2, 5, 7, 0) \tag{4.4}$$
$$identifier_1 = (0, type_{send/receive}, 2, 5, 7, 1) \tag{4.5}$$
$$identifier_2 = (0, type_{send/receive}, 2, 5, 7, 2) \tag{4.6}$$

Assuming there is a matching receiver which also issues 3 `MPI_Recv()` calls with the corresponding parameters, it would generate the same list of identifiers, thus allowing the sender and receiver to uniquely identify each message exchanged between them.

Finally, it is worth nothing that the header generation function above takes a global lock before modifying the hash table containing the per-flow serial numbers and that different communication types are cleanly separated by the `type` field. In combination, this allows the program to run different kinds of communication in parallel; i.e. the program could for example issue a `MPI_Send()`, a `MPI_Recv()` and a `MPI_Reduce()` call in separate threads at the same time and have the right thing happen.

4.5.5 Implicit Global Serialization

With the message identifier generation algorithm introduced in section 4.5.4, it is possible to run a batch of `MPI_Send()` calls in parallel to other code, as long as that other code never calls `MPI_Send()`. This works because the identifier generation function is *thread-safe* and only depends on the inputs provided to the individual `MPI_Send()` calls. This allows the backend to run the necessary `MPI_Send()` calls in one asynchronous helper thread, while handling the `MPI_Recv()` calls in the main thread. In combination with the message buffering in the inbox, this allows the backend to avoid the global serialization problem described in section 4.2.5, as no `MPI_Send()` call ever depends on an unrelated `MPI_Recv()` call to complete first.

4.5.6 Peer Discovery and Rank Assignment

As described in section 4.3, peer discovery and rank determination uses a configuration file and the basic semantics of binding a socket to a name: This solves the problem introduced in section 4.2.6 by delegating it to an outside entity (who- or whatever writes the configuration file). We think that this is more elegant than having a wrapper program and/or a dedicated management process, since it easily allows running the TCP backend in new environments, because the configuration file is simple enough that it can be constructed by e.g. a trivial shell script which contains whatever domain-specific knowledge is required by the execution environment.

4.5.7 Fault Tolerance

The final problem introduced in section 4.2.7 was whether we could use the message identifiers we would need anyway for adding fault-tolerance to our backend. This turned out to be very simple: Since the maximum size of the outbox needed to be configurable anyway, we decided to interpret a size of -1 (or equivalently `SIZE_MAX`) as *infinite*, and handle that as a special case when removing messages from the outbox, by turning that particular function into a no-op. This means that the receiver of a message can send infinitely many GET requests for that message and they will all succeed, even if the receiving process was restarted in the mean time. Furthermore, since the configuration file (which is also responsible for peer discovery), is refreshed periodically, it is actually possible to redirect the members of a process group towards a new address on the fly. This allows a failed process to be restarted on a new node.

5 Implementation

With the design introduced in chapter 4, we are now ready to implement a new TCP based backend for LAIK. This chapter describes the challenges we encountered during the implementation and then gives a short overview of the architecture of the new TCP backend. Finally we demonstrate how to use the new backend, including making use of the built-in fault tolerance.

5.1 Challenges

Early on during the implementation, we encountered three major problems with LAIK itself which we had to solve before we could begin working on the code for the new backend:

1. LAIK did not use any library for basic data structures such as mappings or lists, but instead implemented these data structures itself. Since these implementations were not generalized, but specific to their particular purpose, we would either have to re-implement our own versions of these data structures or choose a suitable third-party library to base our code own. Furthermore, LAIK had no code for error handling or thread management, and we would almost certainly need both to implement a fault-tolerant communication backend based on TCP sockets.
2. In fact, LAIK only had one real dependency at the time, namely an MPI installation for its existing MPI backend. There were some optional leaf components which did use other libraries such as libpapi and libmosquitto, but LAIK did neither build nor test these components per default. Consequently LAIK only had a very basic build system consisting of a few Makefiles and a small Python script used to turn some features on or off, depending on the environment found on the host system. It seemed clear that adding a new backend to this build system (including test and installation support) would be non-trivial, especially if that new backend would use a third-party library not presently supported by the build system.
3. The fact that LAIK had only a very basic build system, meant that a lot of the code present in the project was either not compiled at all on a default run, or was not properly tested: For example, LAIK does support a plugin-mechanism and even contains a few example plugins, but these were not compiled per default. Furthermore, while there is a dummy backed (supporting only a single process) to test LAIK's API for its backends, this code was effectively unreachable if LAIK was compiled with support with MPI. Finally, the Makefile based build system also failed to define a consistent set of compiler flags for the project which meant that only parts of the code compiled when we enabled the `-Wall` and `-Werror` compiler flags.

In order to tackle these problems, we added a CMake based build system to LAIK which had the ability to build all the code contained in the repository if requested (and properly error out if the host system lacked certain dependencies) and use a single set of compiler flags everywhere. With that in place, we turned on the `-pedantic`, `-Wall`, `-Werror`, and `-Wextra` compiler flags and began to fix the detected errors. While most errors were minor, we did find a few real bugs which more than justified the effort put into this.

We then needed to chose a suitable third-party library providing us with the basic data structures and functionality needed for writing a non-trivial network application. We looked at Qt and Boost, but ultimately went for GLib because Qt and Boost are both C++ libraries and LAIK is (at least for now) still a C-only project.

With the new build system in place and the library chosen, we went ahead and implemented the design shown in chapter 4. We also extended LAIK core so that the backend for an application linked against `liblaik` could be dynamically chosen at runtime using the environment variable LAIK_BACKEND. We chose the configuration file (used for configuring a few performance options and for peer discovery) to be a "keyfile" [7] which is be refreshed every second, so adjusting the configuration during runtime is possible.

5.2 Architecture

Figure 5.1 shows the architecture of the new backend. When comparing this with the architecture of the existing MPI backend as seen figure 1.1, we can see that the API spoken between the LAIK backend and our implementation is actually still (a subset of) the MPI API. This decision made incremental development much easier, since we could initially re-use the existing code for LAIK's MPI backend and had a stable and well-document API to implement.

However, we eventually replaced the code taken from the existing MPI backend with our own version of a LAIK backend, because the existing code suffered from a variant of the implicit global serialization issue described in section 4.2.5. Our own version of the LAIK backend solves this issue using the approach described in section 4.5.5, but still interfaces with the actual implementation using the MPI API.

5.3 Demonstration

5.3.1 Regular Mode

We first want to demonstrate the basic workflow of using the TCP backend using the configuration file shown below:

```
[general]
# nothing here

[addresses]
rank0 = localhost 2000
rank1 = localhost 2001
rank2 = localhost 2002
rank3 = localhost 2003
```

Listing 5.1: A simple configuration file for the TCP backend without any special settings

We also need a small script which is responsible for setting up the environment and starting the required number of processes. This script contains the following:

```
#!/bin/sh -eu

export LAIK_BACKEND='tcp'
export LAIK_TCP_CONFIG='config.txt'

./jac3d 500 &
./jac3d 500 &
./jac3d 500 &
./jac3d 500 &
```

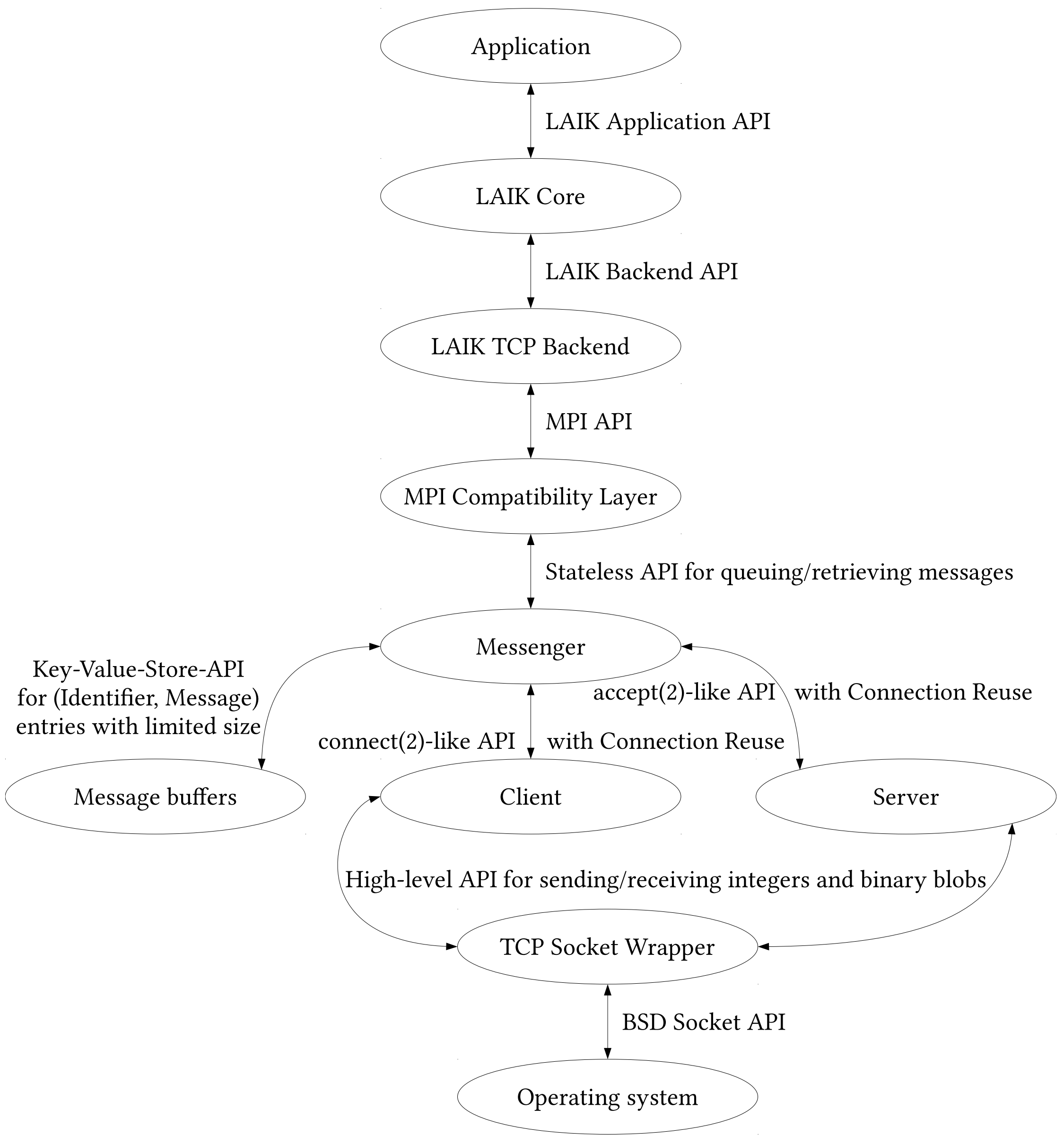

Application
LAIK Application API
LAIK Core
LAIK Backend API
LAIK TCP Backend
MPI API
MPI Compatibility Layer
Stateless API for queuing/retrieving messages
Messenger
Key-Value-Store-API for (Identifier, Message) entries with limited size
accept(2)-like API with Connection Reuse
connect(2)-like API with Connection Reuse
Message buffers
Client
Server
High-level API for sending/receiving integers and binary blobs
TCP Socket Wrapper
BSD Socket API
Operating system

Figure 5.1: Stack overview of LAIK when the new TCP backend is used

```

wait
```

Listing 5.2: This script demonstrates how to run an example program using the TCP backend

Running the script gives us this output:

```
$ time ./run.sh
500 x 500 x 500 cells (mem 2000.0 MB), running 50 iterations with 4 tasks
Residuum after  1 iters: 377335777.221728
Residuum after 11 iters: 506561.854361
Residuum after 21 iters: 99032.165751
Residuum after 31 iters: 54811.260127
Residuum after 41 iters: 35891.442200

real 0m16.160s
user 1m0.756s
sys 0m1.630s
$
```

Listing 5.3: This output is generated when running the example program with the TCP backend in use

We can see that the script starts four instances of a program from LAIK's example suite, that they properly discover each other using the addresses provided in the configuration file, and finally that the distributed computation works as expected.

5.3.2 Failure Detection

Next, we want to demonstrate that the TCP backend can correctly detect when a process fails to respond and abort the other processes after a configurable timeout. For this we modify the configuration file from the previous section by limiting the maximum number of send and receive attempts to 100, with a 0.1 second delay between each attempt, effectively giving us a 10 second timeout[1]:

```
[general]
send_attempts = 100
send_delay = 0.1
receive_attempts = 100
receive_delay = 0.1

[addresses]
rank0 = localhost 2000
rank1 = localhost 2001
rank2 = localhost 2002
rank3 = localhost 2003
```

Listing 5.4: A configuration file for the TCP backend with explicit limits on the amount of send/receive attempts

We also modify our script to abort one of the processes after 10 seconds:

```
#!/bin/sh -eu

export LAIK_BACKEND='tcp'
export LAIK_TCP_CONFIG='config.txt'

./jac3d 500 &
./jac3d 500 &
./jac3d 500 &
```

[1]These settings are actually the default and only shown here for demonstration purposes. The reasoning behind having a timeout enabled per default, is that our TCP backend has no central manager, so that a crashed or unreachable node would otherwise lead to the other processes waiting forever.

```
timeout 10 ./jac3d 500 &

wait
```

Listing 5.5: This script runs the example program using the TCP backend, but purposefully causes one process to fail

Now we can run the script:

```
$ time ./run.sh
500 x 500 x 500 cells (mem 2000.0 MB), running 50 iterations with 4 tasks
Residuum after  1 iters: 377335777.221728
Residuum after 11 iters: 506561.854361
Residuum after 21 iters: 99032.165751
[LAIK TCP Backend] Aborting, the contents of the error stack follow:
 => Domain laik_tcp_backend_exec encountered error #0: Reduce operation failed
 => Domain laik_tcp_backend_reduce encountered error #2: Failed to receive reduction
     input from task 3
 => Domain laik_tcp_backend_push_code encountered error #0: An MPI operation failed,
     details below
 => Domain laik_tcp_minimpi_recv encountered error #1: Failed to receive message from
     task 3
 => Domain laik_tcp_messenger_get encountered error #0: Maximum number of attempts
     exceeded while attempting to receive message from rank 3

[LAIK TCP Backend] Aborting, the contents of the error stack follow:
 => Domain laik_tcp_backend_exec encountered error #0: Reduce operation failed
 => Domain laik_tcp_backend_reduce encountered error #2: Failed to receive reduction
     input from task 3
 => Domain laik_tcp_backend_push_code encountered error #0: An MPI operation failed,
     details below
 => Domain laik_tcp_minimpi_recv encountered error #1: Failed to receive message from
     task 3
 => Domain laik_tcp_messenger_get encountered error #0: Maximum number of attempts
     exceeded while attempting to receive message from rank 3

[LAIK TCP Backend] Aborting, the contents of the error stack follow:
 => Domain laik_tcp_backend_exec encountered error #0: Reduce operation failed
 => Domain laik_tcp_backend_reduce encountered error #2: Failed to receive reduction
     input from task 3
 => Domain laik_tcp_backend_push_code encountered error #0: An MPI operation failed,
     details below
 => Domain laik_tcp_minimpi_recv encountered error #1: Failed to receive message from
     task 3
 => Domain laik_tcp_messenger_get encountered error #0: Maximum number of attempts
     exceeded while attempting to receive message from rank 3


real 0m20.013s
user 0m38.262s
sys 0m1.401s
$
```

Listing 5.6: The output shows that the TCP backend correctly detects the failure after a timeout and terminates the remaining processes

As expected, the remaining processes detect that the terminated processed fails to respond and consequently abort with an error after the timeout expires. It is worth noting that the total execution time is roughly 20 seconds which corresponds to the 10 seconds after which the fourth process is terminated, plus the 10 seconds it takes the other processes to reach the timeout.

5.3.3 Failure Recovery

In the previous section we have purposefully terminated one of the processes to test that the remaining processes properly abort after a timeout. Now we want to test whether the fault tolerance of the TCP backend works. We change the configuration file so that sending and receiving messages are retried indefinitely and that sent messages are never removed from the outbox (by setting its maximum size to −1 which is interpreted as "infinity"):

```
[general]
send_attempts = -1
receive_attempts = -1
outbox_size = -1

[addresses]
rank0 = localhost 2000
rank1 = localhost 2001
rank2 = localhost 2002
rank3 = localhost 2003
```

Listing 5.7: The configuration file for the TCP backend with fault tolerance enabled

Next we modify the script to not only terminate a process after 10 seconds, but to also spawn a replacement process then:

```
#!/bin/sh -eu

export LAIK_BACKEND='tcp'
export LAIK_TCP_CONFIG='config.txt'

./jac3d 500 &
./jac3d 500 &
./jac3d 500 &

if ! timeout 10 ./jac3d 500; then
 echo "Process␣died,␣restarting"
 ./jac3d 500
fi &

wait
```

Listing 5.8: A script running an example program using the TCP backend, but purposefully failing and restarting one process

This gives us the following output:

```
$ time ./run.sh
500 x 500 x 500 cells (mem 2000.0 MB), running 50 iterations with 4 tasks
Residuum after  1 iters: 377335777.221728
Residuum after 11 iters: 506561.854361
Residuum after 21 iters: 99032.165751
Process died, restarting
Residuum after 31 iters: 54811.260127
Residuum after 41 iters: 35891.442200

real 0m21.374s
user 1m6.426s
sys 0m2.097s
$
```

Listing 5.9: The output shows that one process was killed and successfully restarted

We can see that the distributed computation takes significantly longer to complete (21 seconds compared to the 16 seconds from section 5.3.1), but gives the same result, so our approach for bringing fault-tolerance to LAIK seems to work as expected.

5.3.4 Failure Recovery with Address Change

For the final demonstration, we want to showcase the ability of the new TCP backend to replace a failed process with a new process reachable via a different address. The initial configuration file is identical to the one used in the previous section:

```
[general]
send_attempts = -1
receive_attempts = -1
outbox_size = -1

[addresses]
rank0 = localhost 2000
rank1 = localhost 2001
rank2 = localhost 2002
rank3 = localhost 2003
```

Listing 5.10: The configuration file for the TCP backend with fault tolerance enabled

The script is also mostly identical to the one used in the previous section with two important differences:

1. Before starting the fourth process, we wait for one second to make sure that the first three addresses listed in the configuration file are already taken and the fourth process has to take the fourth address.
2. After terminating the fourth process we change the configuration file so that the fourth address uses a different port (3333 instead of 2003), and only then spawn the replacement process.

```
#!/bin/sh -eu

export LAIK_BACKEND='tcp'
export LAIK_TCP_CONFIG='config.txt'

./jac3d 500 &
./jac3d 500 &
./jac3d 500 &

# Make sure the first three instance are started up and took rank 0,1, and 2
sleep 1

if ! timeout 10 ./jac3d 500; then
 echo "Process␣died,␣restarting␣on␣different␣port"
 sed -i 's/rank3 = localhost 2003/rank3 = localhost 3333/' 'config.txt'
 ./jac3d 500
fi &

wait
```

Listing 5.11: A script running an example program using the TCP backend, but purposefully failing a process and then restarting it with a different address

Running this script gives us the following output:

```
$ time ./run.sh
500 x 500 x 500 cells (mem 2000.0 MB), running 50 iterations with 4 tasks
Residuum after  1 iters: 377335777.221728
Residuum after 11 iters: 506561.854361
Residuum after 21 iters: 99032.165751
Process died, restarting on different port
Residuum after 31 iters: 54811.260127
Residuum after 41 iters: 35891.442200

real 0m19.190s
user 1m1.699s
sys 0m1.890s
$
```

Listing 5.12: The output shows that one process was killed and successfully restarted; the remaining processes correctly picked up the new address of the started process

Everything still works, even though we now effectively moved one of the processes during the distributed computation. This combines the feature of periodically reloading the configuration file with the optional fault tolerance of the TCP backend.

6 Evaluation

With the new backend in place, we now want to evaluate its performance, compared to the existing MPI backend. For this, we first introduce the system on which we performed the evaluation and the programs used as test cases. Then, we explain how we measured the execution time of the test cases and describe the runtime environments these test cases were run in. Next, we detail the test variables and metric, present the results and finally discuss them shortly.

6.1 Test System

We wanted a test system with enough nodes so that the communication cost required to synchronize the processes running on these nodes would be significant without the option for an MPI implementation to take advantage of a potentially faster non-network communication channel. The second requirement was that the system needed to be fast enough to process the domain-specific part of the application quickly, so that the computation time would not dominate the communication overhead. Third, as we wanted to compare the performance of two competing implementations, the test system had to be otherwise unoccupied to avoid external factors such as other users or network traffic to influence the measurements.

The HimMUC cluster [3] built on our chair as a research project proved to be ideal for our purposes: It consists of two partitions, one containing 40 Raspberry Pi 3 units and the other one containing 40 ODroid C2 boards. Each partition is interconnected via a single Gigabit-Ethernet-Switch and both partitions are managed by an external virtual machine running the SLURM workload manager [28]. Using MPI applications (or LAIK applications with the existing MPI backend) is also possible via a custom OpenMPI 3.0.0 installation configured to work correctly with SLURM [43]. Finally, both partitions were fortunately otherwise unoccupied during the evaluation, reducing outside influence.

We chose to only use the ODroid C2 partition of the HimMUC since its nodes provide faster processors, more memory and faster network cards compared to the Raspberry Pi partition. This allowed us to run more complex test cases while making sure that the total execution time was not completely dominated by the computational effort required. The cluster can be seen in figure 6.1; each node contained the following hardware [3]:

- SoC: Amlogic S905
- CPU: 4 x ARM Cortex-A53, 1.5 GHz
- GPU: Mali-450
- RAM: 2GB (912 MHz)
- Network: 1 GBit Ethernet

Figure 6.1: The ODroid C2 partition of the HimMUC [3]

6.2 Test Cases

Since we built a new backend for the LAIK library, we needed applications using this library for running the evaluation. Unfortunately, as LAIK is still a very young project, there are only a few such applications available. Furthermore, LAIK does not yet provide a stable API, so the only applications which we could rely on to work correctly with the new TCP backend were the example applications contained in the LAIK project, as those were also used in LAIK's integration tests (which the new TCP backend passed reliably).

Unfortunately, the HimMUC cluster did not provide a recent version of all the necessary dependencies to build LAIK. Furthermore, while the management VM was an amd64 machine, the compute nodes were arm64 machines. We therefore decided to cross-build LAIK and the examples used for testing locally. This turned out to be relatively easy thanks to the new CMake based build system described in section 5.1. The details are available in section 9.1; however, it is worth noting that we took care to create a *Release* (i.e. fully optimized) build for the evaluation.

When formulating the exact list of test cases, we had two requirements: We first wanted to use the entire set of example applications provided by LAIK (as far as possible, unfortunately the examples not used in the integration tests all failed to run correctly on the HimMUC). Second, we wanted each test case to not complete almost immediately but still terminate within a reasonable time (we targeted roughly 5 seconds per run). In the end, we came up with the following 13 command lines as our test cases:

- `jac1d 125000`
- `jac2d 11000`
- `jac3d 375`
- `jac3d -g 375`
- `markov 400 4000`
- `markov2 400 4000`
- `markov2 -f 400 4000`
- `propagation2d 40 40`

- `spmv 5000`
- `spmv2 150 1500`
- `vsum 20000000`
- `vsum2`
- `vsum3`

6.3 Time Measurements

As mentioned in section 6.1, the HimMUC test system uses the SLURM workload manager to distribute jobs scheduled on the management VM to the compute nodes. Fortunately, the management VM and the compute nodes share a common NFS based file system, so allocating a few nodes from the cluster and then running the LAIK examples cross-built for arm64 on them is simple:

```
$ file lib/*
lib/liblaik.so:   symbolic link to ../laik/src/liblaik.so
lib/libmpi.so.20: broken symbolic link to /opt/openmpi/lib/libmpi.so
$ salloc --partition=odr --nodes=20 --ntasks-per-node=2
salloc: Granted job allocation 18656
salloc: Waiting for resource configuration
salloc: Nodes odr[01-20] are ready for job
$ LD_LIBRARY_PATH=~/lib LAIK_BACKEND=mpi srun laik/examples/jac3d 100
100 x 100 x 100 cells (mem 16.0 MB), running 50 iterations with 40 tasks
Residuum after  1 iters: 3088288.333333
Residuum after 11 iters: 11612.580828
Residuum after 21 iters: 3544.954272
Residuum after 31 iters: 1925.258713
Residuum after 41 iters: 1238.432564
$
```

Listing 6.1: An example of running a program from the LAIK example suite on the HimMUC

Now, if we want to compare the new TCP backend to the existing MPI backend, the obvious way is to measure the total execution time for an `srun` invocation on the management VM shown above. However, for this to work, we have to first determine how much overhead SLURM itself causes. For this, we start $5\ nodes \cdot 1\ processes/node = 5$ processes running a simple no-op command:

```
$ salloc --partition=odr --nodes=5 --ntasks-per-node=1
salloc: Granted job allocation 18652
salloc: Waiting for resource configuration
salloc: Nodes odr[01-05] are ready for job
$ for i in 1 2 3 4 5; do time srun true; done

real 0m0.294s
user 0m0.020s
sys 0m0.004s

real 0m0.265s
user 0m0.012s
sys 0m0.012s

real 0m0.267s
user 0m0.024s
sys 0m0.000s

real 0m0.262s
user 0m0.020s
sys 0m0.004s

real 0m0.270s
user 0m0.020s
sys 0m0.004s
```

```
$
```

Listing 6.2: A simple no-op command is run a number of times on the HimMUC while the network is not under any load

The results look promising: Assuming that the `true` command takes almost no time, the overhead caused by SLURM seems to be relatively small and constant, around 0.2 to 0.3 seconds per `srun` invocation. This overhead should not matter much if we make sure that the real test cases used later on take significantly more time to complete.

Unfortunately, the situation changes dramatically if the network is put under heavy load: To demonstrate this, we first start an `iperf` server on the management VM and then spawn $30\ nodes \cdot 4\ processes/node = 120$ processes running the corresponding `iperf` client command. Then, we again start $5\ nodes \cdot 1\ processes/node = 5$ processes running the `true` command:

```
$ ./iperf.amd64 --server --time=300 1>/dev/null 2>/dev/null &
[1] 4345
$ srun --partition=odr --nodes=30 --ntasks-per-node=4 ./iperf.arm64 --client
    10.42.1.253 --dualtest --time=300 1>/dev/null 2>/dev/null &
[2] 4398
$ salloc --partition=odr --nodes=5 --ntasks-per-node=1
salloc: Granted job allocation 18654
$ for i in 1 2 3 4 5; do time srun true; done

real 0m1.874s
user 0m0.020s
sys 0m0.004s

real 0m1.836s
user 0m0.024s
sys 0m0.004s

real 0m7.229s
user 0m0.020s
sys 0m0.004s

real 0m1.802s
user 0m0.016s
sys 0m0.012s

real 0m2.242s
user 0m0.020s
sys 0m0.008s
$
```

Listing 6.3: A simple no-op command is run a number of times on the HimMUC while the network is under heavy load

The result clearly indicate that measuring the invocation time of the `srun` command on the management VM will make getting usable results difficult at best: Not only has the overhead increased drastically (to typically about 1.8 to 2.3 seconds), it has also become a lot more unstable, with one invocation taking as much as 7.2 seconds. We were able to reproduce these “jumps” in the overhead reliably under heavy network load. Furthermore we even sporadically encountered timeouts in the communication between the SLURM instances on the management VM and the compute nodes, causing the scheduled job to fail entirely.

As measuring the total execution time of the `srun` command on the management VM was not feasible, we built our own timing and synchronization solution consisting of a small “client” wrapper script (see section 9.5.7) which the compute nodes would run in place of the real test cases and a “server” script (see section 9.5.6) the management VM would run to synchronize and time the test runs.

The basic idea was that before and after executing the real test case, the clients would connect to the server and block until the server closed the connection which it would do by simply terminating once the expected number of connections (one for each processes spawned on the compute nodes) had been accepted. Since the server would print a timestamp just before exiting this served both as a means to prevent some processes from starting (much) earlier than others as well as a way to record the start/stop time.

6.4 Environments

With the startup synchronization and time measurement issues addressed, we now needed to decide on the runtime *environments* to benchmark. We chose the following:

6.4.1 Environment 1: MPI

LAIK's existing MPI backend would be benchmarked by the custom OpenMPI 3.0.0 implementation installed on the HimMUC, without any special runtime parameters or environment variables (see section 9.5.2 for details).

6.4.2 Environment 2: TCP

LAIK's new TCP backend would be tested with its default parameters (see section 9.3 for details), with the sole exception of the maximum limit of message receive attempts which was increased from its default value of 100 to 1000. In combination with the default value for the receive retry delay of 0.1 seconds, this simply increased the total message receive timeout from 10 to 100 seconds, so that even long-running test-cases would not encounter this timeout. The precise detail are available in section 9.5.1

6.4.3 Environment 3: TCP with Master Reduction (TCP-M)

Per default, the TCP backend computes reductions by having the input processes (those providing input to the reduction) send their data to *all* the output processes (those interested in the result of the reduction). This environment is identical to the TCP environment described in section 6.4.2, except that reductions are computed using a dedicated master processes (see 9.5.3 for details) which receives the input from all input processes, computes the result, and sends it back out the all the other output processes (the master processes is chosen so that it always is an output processes).

This modification reduces the amount of messages required for a reduction with N *input processes* and M output processes from $N * M$ to $N + M$, i.e. from $O(n^2)$ to $O(n)$. However, it also blocks *all* the output processes until the dedicated master processes has received *all* the inputs, computed the reduction result, and sent this result back to *all* the output processes. With this environment, we therefore hope to determine which of the two reduction strategies is better, depending on the test case and the number of processes involved.

6.4.4 Environment 4: TCP with Extra Resources (TCP+)

The default limits for the resources used by the TCP limit are quite conservative: Its buffers for incoming and outgoing messages (called the inbox and outbox) are limited to 16 MiB, both the client and server side may not keep more than 16 connections open concurrently, and the socket backlog used for listening sockets is set to just 64 connections (see 9.3 for a full list of the default values). This environment is identical to the TCP environment described in section 6.4.2, except that the buffers may now grow to 256 MiB, the connection limit has been bumped to 400, and the socket backlog is now also 400 connections (see section 9.5.4 for details).

These new limits are still well below the limits imposed by the hardware or the operating system, even on systems such as the ODroid C2 boards used here. However, they are big enough to potentially cause problems in some situations (such as having other resource-hungry programs running in parallel), so they can not be made the default. In summary, this environment should give us an idea of the performance the TCP backend is able to achieve under ideal conditions.

6.5 Test Variables and Metric

In addition to the 13 test cases described in section 6.2 and the 4 environments described in section 6.4, we also varied the number of ODroid C2 nodes used (from 5 to 35) and the number of processes per node (from 1 to 4), giving us a total of $13 \cdot 4 \cdot 31 \cdot 4 = 6448$ data points to collect. Each data point would represent the time spent on the distributed computation, measured using the approach described in section 6.3. Furthermore, for each data point we collected $n = 10$ samples and calculated the average $\overline{x}$ and the sample standard deviation σ:

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{6.1}$$

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2} \tag{6.2}$$

Unfortunately, the results gathered in this way sporadically contained unexpectedly high averages and huge sample standard deviations. We suspect that the underlying problem which caused the jumps in execution time encountered in section 6.3 is either a fundamental network issue, affecting all communications on the cluster, or at least also affects our own time stamping mechanism. Regardless, in order to get usable results, we decided to filter the data points after the initial test run and regenerate all the data points with a sample standard deviation exceeding $\max\{1, 0.10 \cdot \overline{x}\}$ seconds using the script shown in section 9.5.8.

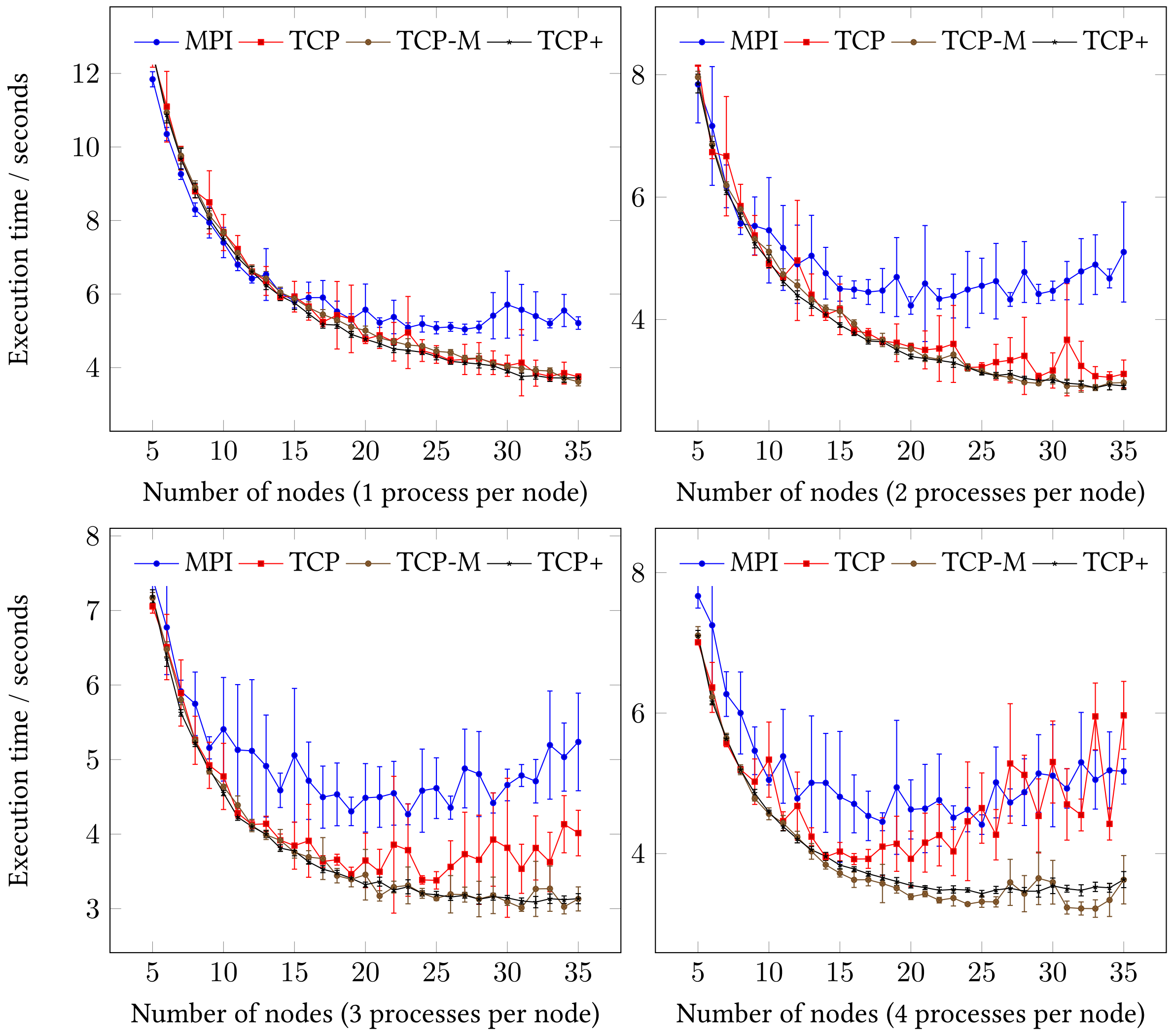
6.6 Results

This section presents the results of evaluation. For every test case, we first give a short introduction into what the program requires the backend to do using the analysis of the test cases we have conducted in section 9.2. We then present the resulting graphs and shortly discuss them.

6.6.1 jac1d 125000

Looking at section 9.2.1, we see that this test case mostly issues send/receive operations to the backend and only a few reductions. All send/receive operations concern messages with a size of 8 bytes, and the backend typically gets two such operations per invocation. Overall, we expect the TCP backend to not perform very well here because of the very small messages. However, the total number of messages is only 900, so the performance hit compared to the MPI backend should not be too bad.

Figure 6.2: Evaluation of `jac1d 125000` using 5-35 nodes with 1-4 processes per node

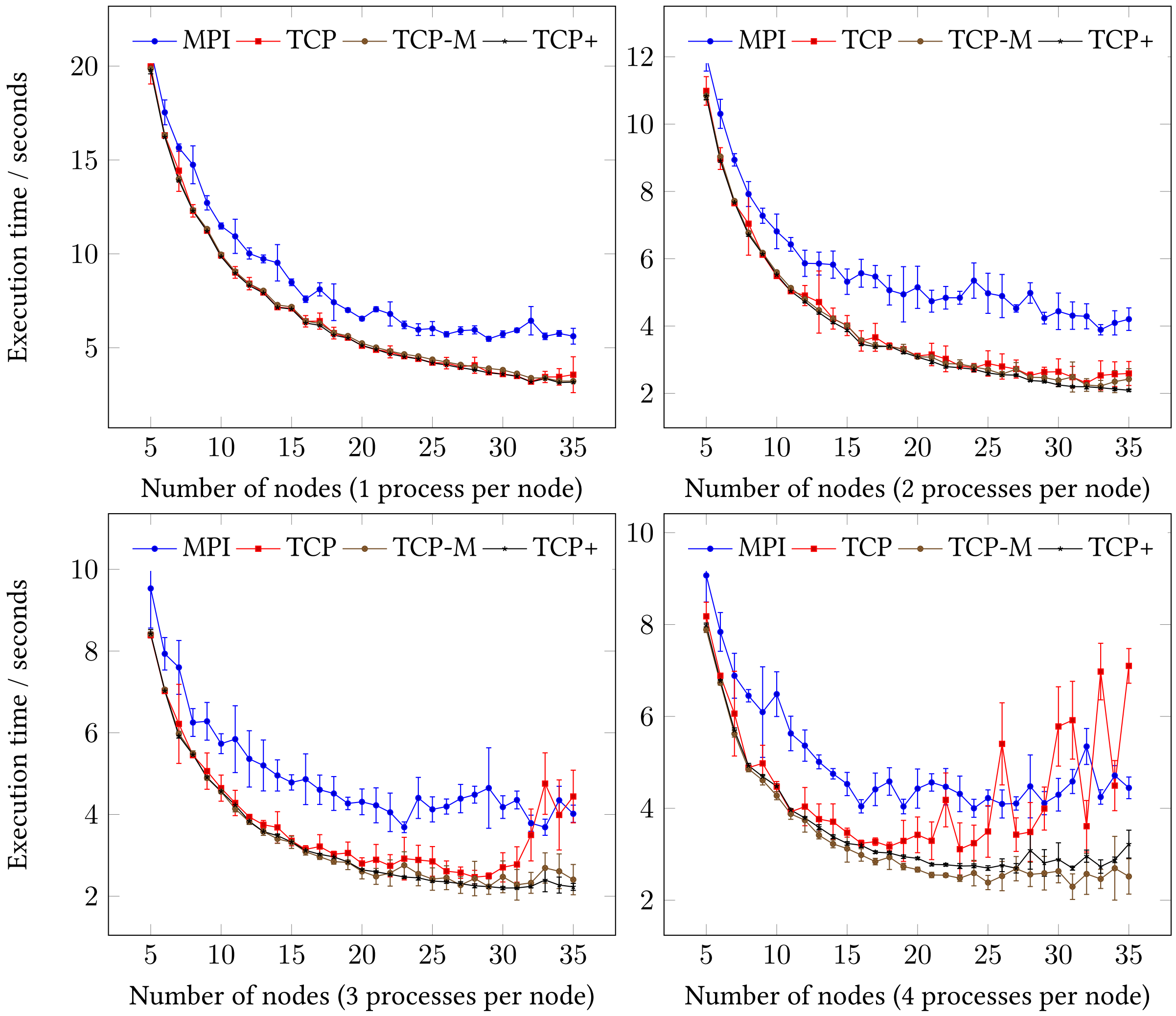


Despite the small message sizes, the TCP backend almost always outperforms the MPI backend here. Furthermore, while the MPI backend actually begins to take more time to complete the computation once more than 20 nodes are used, the TCP backend scales much better, with only the graphs representing 3 and 4 processes per node showing a similar effect. However, this only applies to the default configuration, the TCP-M and TCP+ variants are both able to benefit from more nodes almost everywhere, although the benefit per additional node decreases.

6.6.2 jac2d 11000

In contrast to the previous test case, the analysis in section 9.2.2 suggests that this test case is much better suited for the TCP backend: The backend is frequently called with multiple send/receive operations, and those overwhelmingly consists of messages in the order of several kilobytes. As previously, there were only a few reductions. We therefore expect that the TCP backend can perform about as well as the MPI backend here, if not better.

Figure 6.3: Evaluation of `jac2d 11000` using 5-35 nodes with 1-4 processes per node

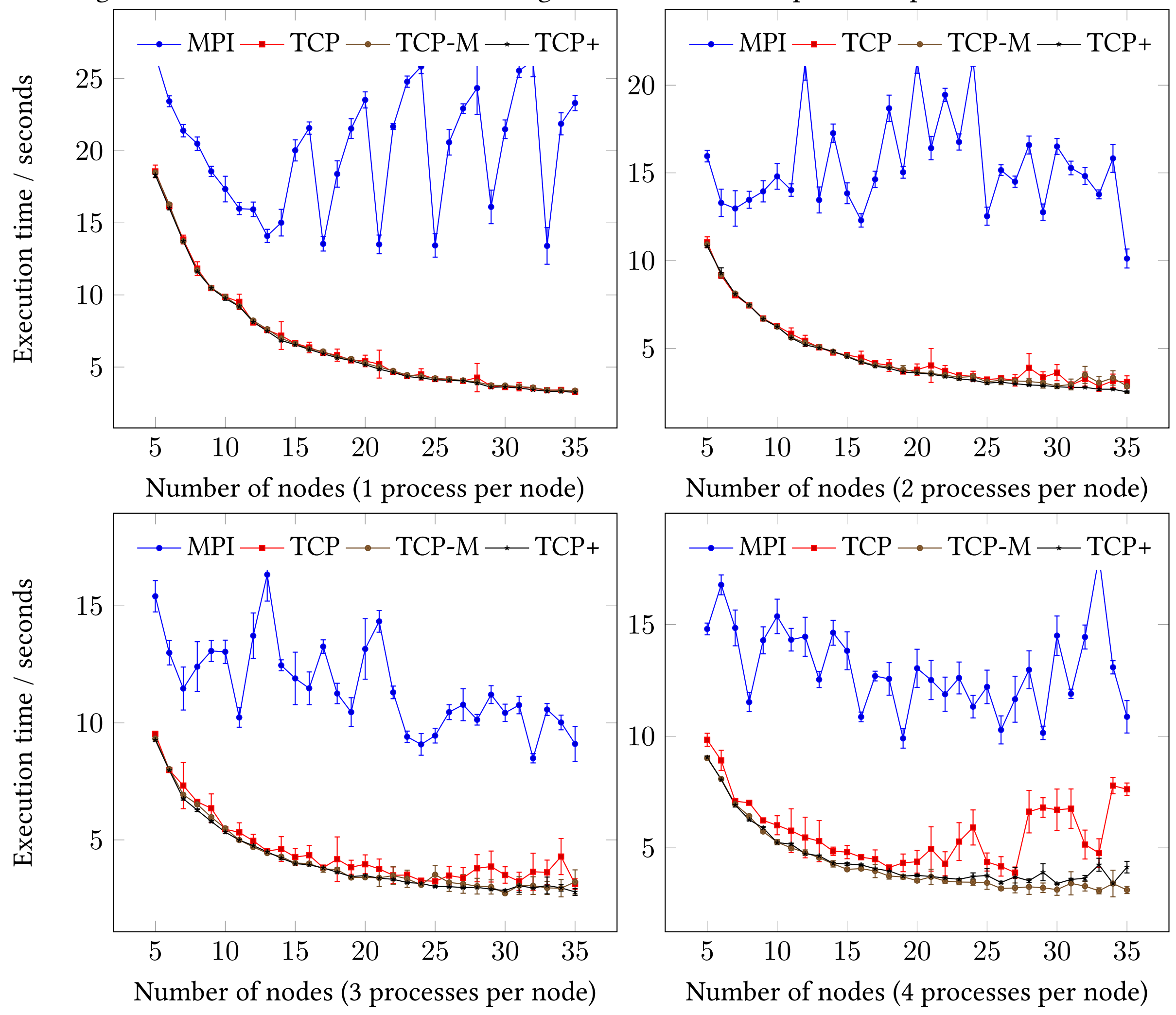


We see a similar result as in the previous section here, the TCP backend is better almost everywhere, except when many nodes and three or more processes per node are used. The TCP-M and TCP+ variants however still beat the MPI backend even in these cases. We suspect that the default configuration (which sends/receives $O(n^2)$ messages per reduction) reaches the connection limit with enough processes involved, and that either using a $O(n)$ reduction algorithm or a higher limit avoid the issue.

6.6.3 jac3d 375

This test case is very similar to the previous one, except that is even better suited for the TCP backend: Section 9.2.3 shows that the program submits even more send/receive operations per invocation to the backend, and those operations consist of messages of frequently up to several hundred kilobytes. Again, there are almost no reductions.

Figure 6.4: Evaluation of `jac3d 375` using 5-35 nodes with 1-4 processes per node

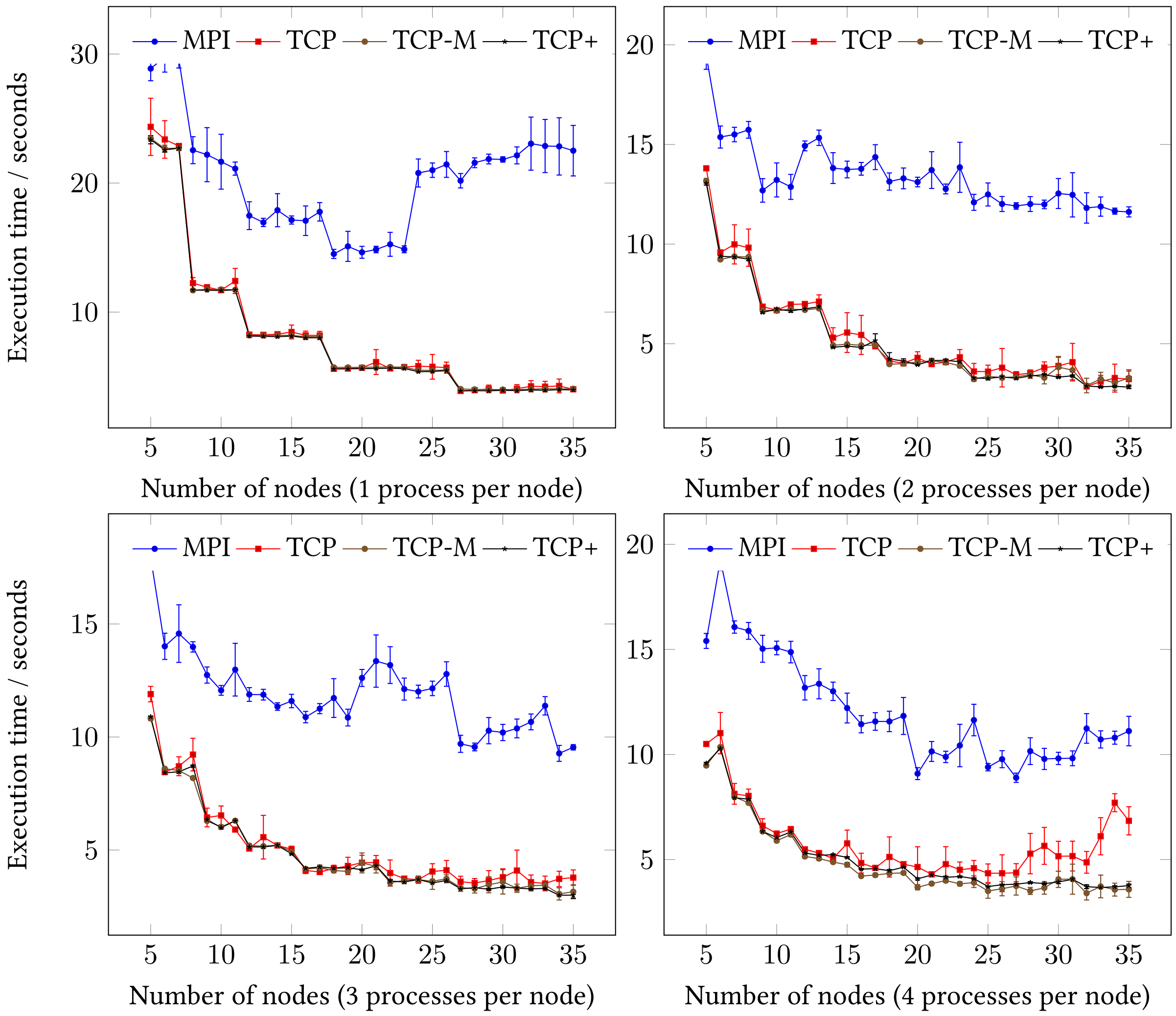


In essences, the behavior of the TCP backend is identical to the previous two test case here, but the MPI backend behaves strangely: There are clear and repeated jumps in the graph and overall the MPI backend takes significantly more time to complete the computation, regardless of the specific test parameters. We think that this test case triggers a particularly bad variant of the problem described in section 4.2.5, as the MPI backend contains a variant of the code shown there.

6.6.4 jac3d -g 375

The only difference here compared to the previous test case is the extra `-g` parameter used in the command line. This seems to make the program issue less varied backend invocations as shown in section 9.2.4, but overall the analysis from the previous test case is still valid: Multiple send/receive operations per backend invocation, large message sizes and only a few reductions.

Figure 6.5: Evaluation of `jac3d -g 375` using 5-35 nodes with 1-4 processes per node



In contrast to the previous test case, there are now clear steps in the graphs, but the overall assessment still holds: The TCP backend is much faster than the MPI backend, and both the TCP-M and TCP+ variant are faster than the TCP backend running in its default configuration when many processes are involved.

6.6.5 markov 400 4000

The analysis in section 9.2.5 shows that this test case almost always calls the backend with 9 send/receive operations, each consisting of messages with a size of 320 bytes. There are no reductions submitted to the backend. Overall, we expect that this test case should perform well using the TCP backend, as there are many messages to be sent in parallel, and the messages do not have a negligible size.

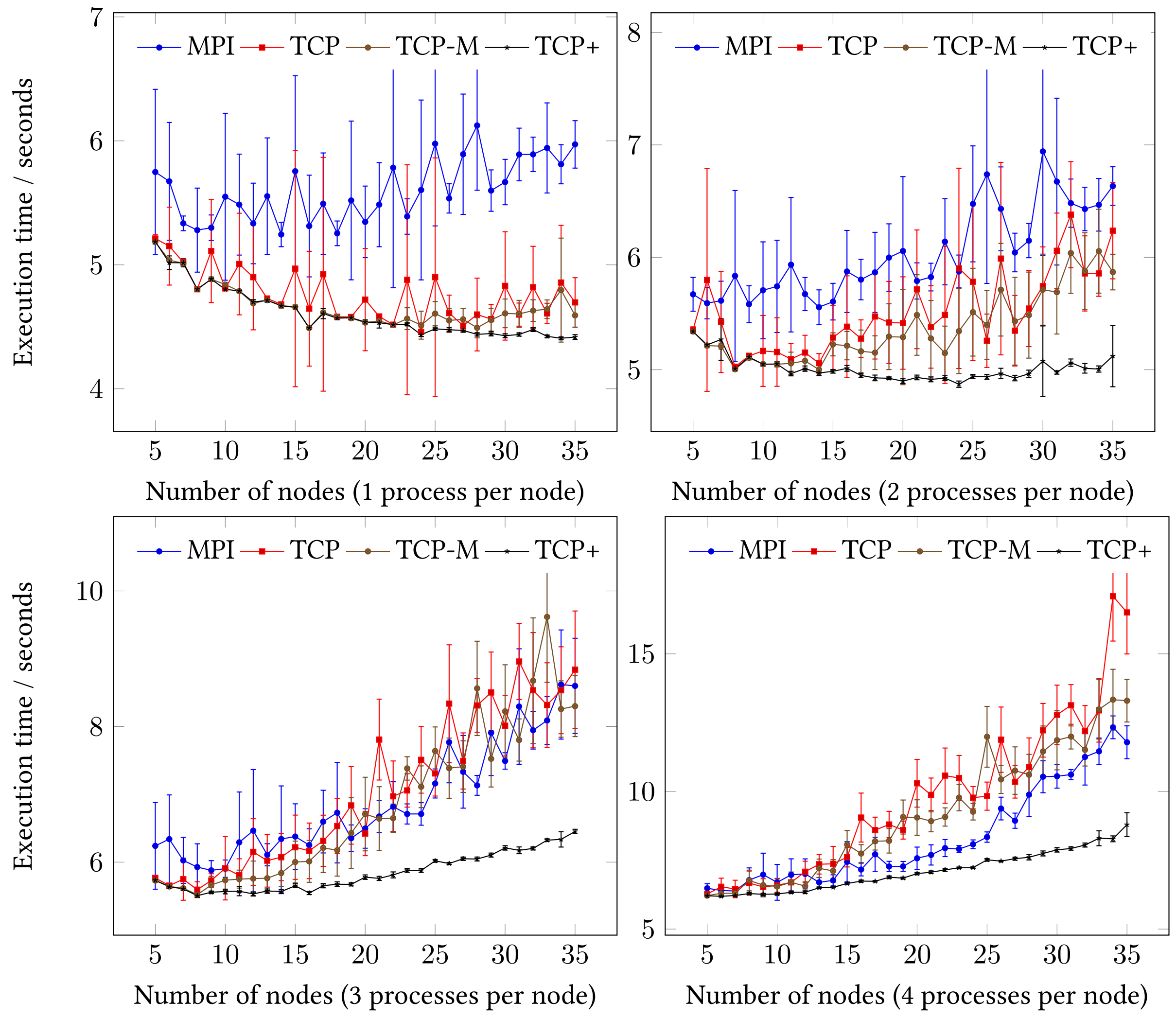


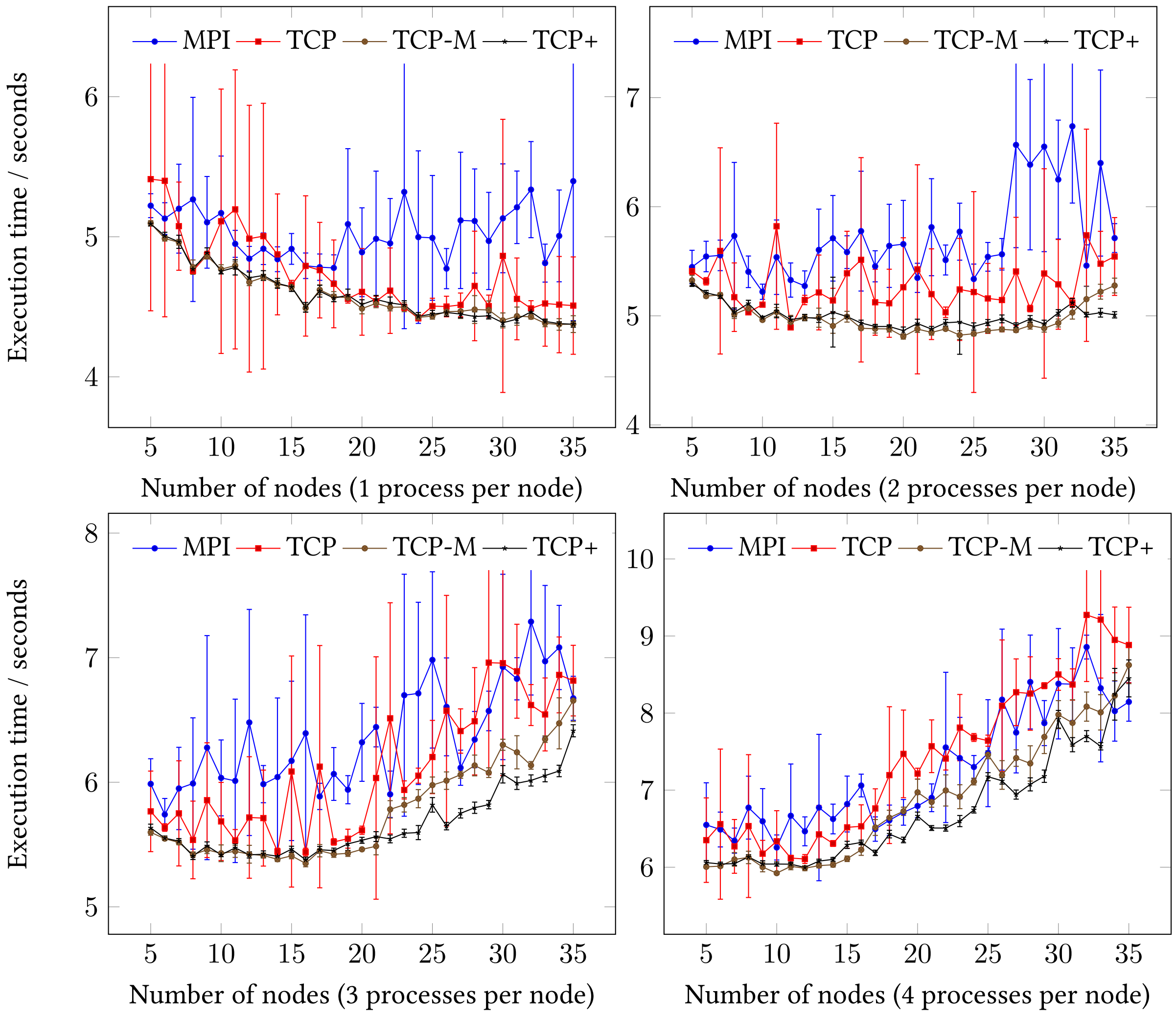
Figure 6.6: Evaluation of `markov 400 4000` using 5-35 nodes with 1-4 processes per node

The results are interesting: For 1 and 2 processes per node, the TCP backend outperforms the MPI backend in all variants, for 3 processes per node all environments except TCP+ are roughly equivalent, and for 4 processes per node, the MPI backend is faster than the TCP backend in its default configuration, but slower than TCP+. We therefore think that the underlying problem of the TCP backend here is the relatively low default connection limit which means that with many processes it is possibly that connections get evicted from the connection cache before they can be reused.

6.6.6 markov2 400 4000

This test just uses reductions according to the analysis in section 9.2.6. Most of the time, the backend gets 10 reductions per invocation, with each reduction usually concerning 320 bytes. In total, this usage profile should not favor the TCP backend, as it cannot run reductions in parallel, in contrast to send/receive operations.

Figure 6.7: Evaluation of `markov2 400 4000` using 5-35 nodes with 1-4 processes per node

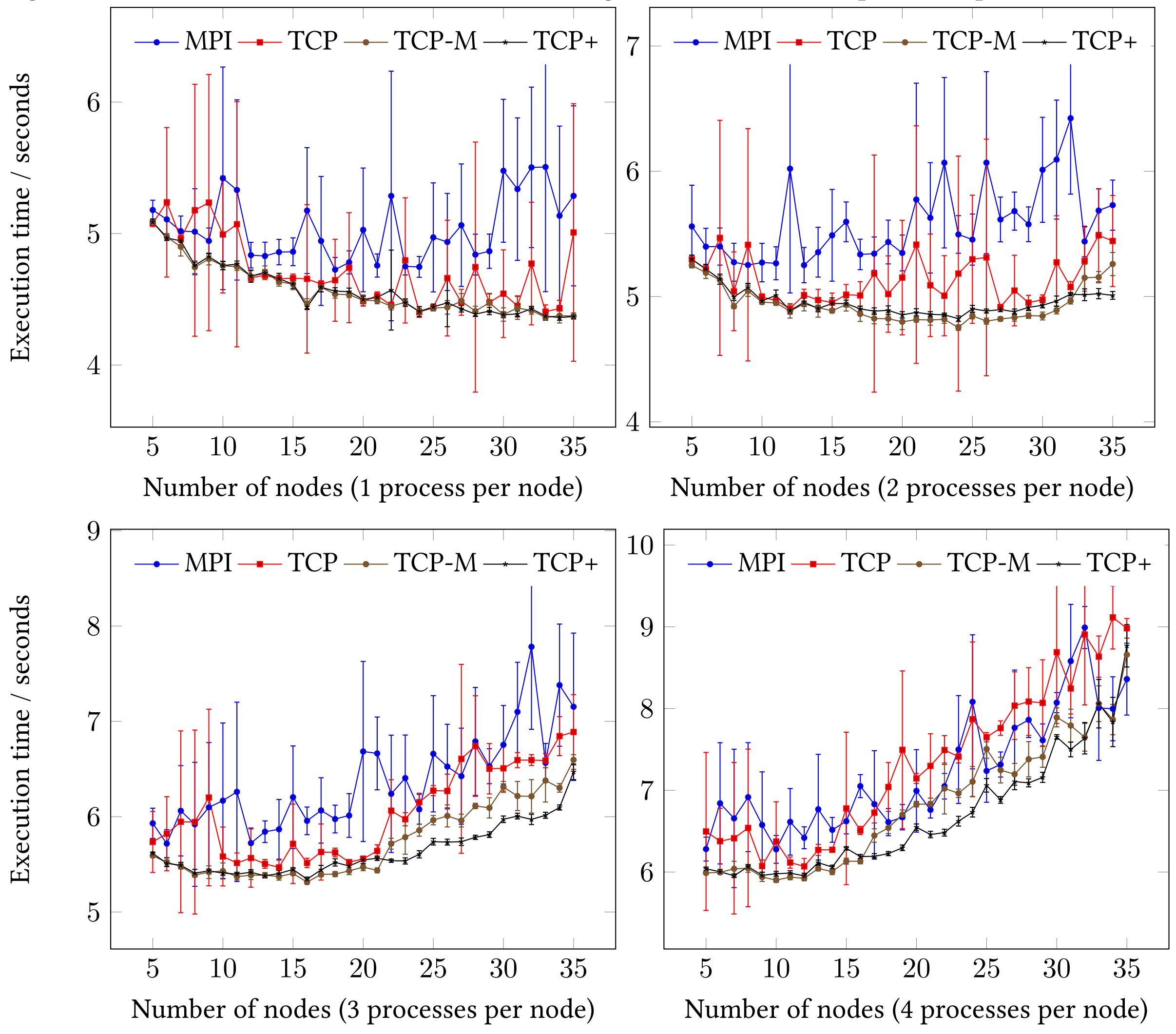


In short, the result mirror those from the previous test case: The TCP backend performs better than MPI with a lower process group size, but loses that advantage once enough nodes or processes per node are used. The TCP-M and TCP+ variants still seem to be slightly better than MPI everywhere, but not by much. However, given the application analysis described above, this is not surprising: Small reductions appear to be the Achilles heel of the TCP backend, probably because they are not run in parallel.

6.6.7 markov2 -f 400 4000

The analysis in section 9.2.7 shows that this test case behaves identically to the previous test case which is not surprising given that we only added the `-f` flag to the command line. Therefore, we expect the TCP backend to perform about as well as in the previous section.

Figure 6.8: Evaluation of `markov2 -f 400 4000` using 5-35 nodes with 1-4 processes per node

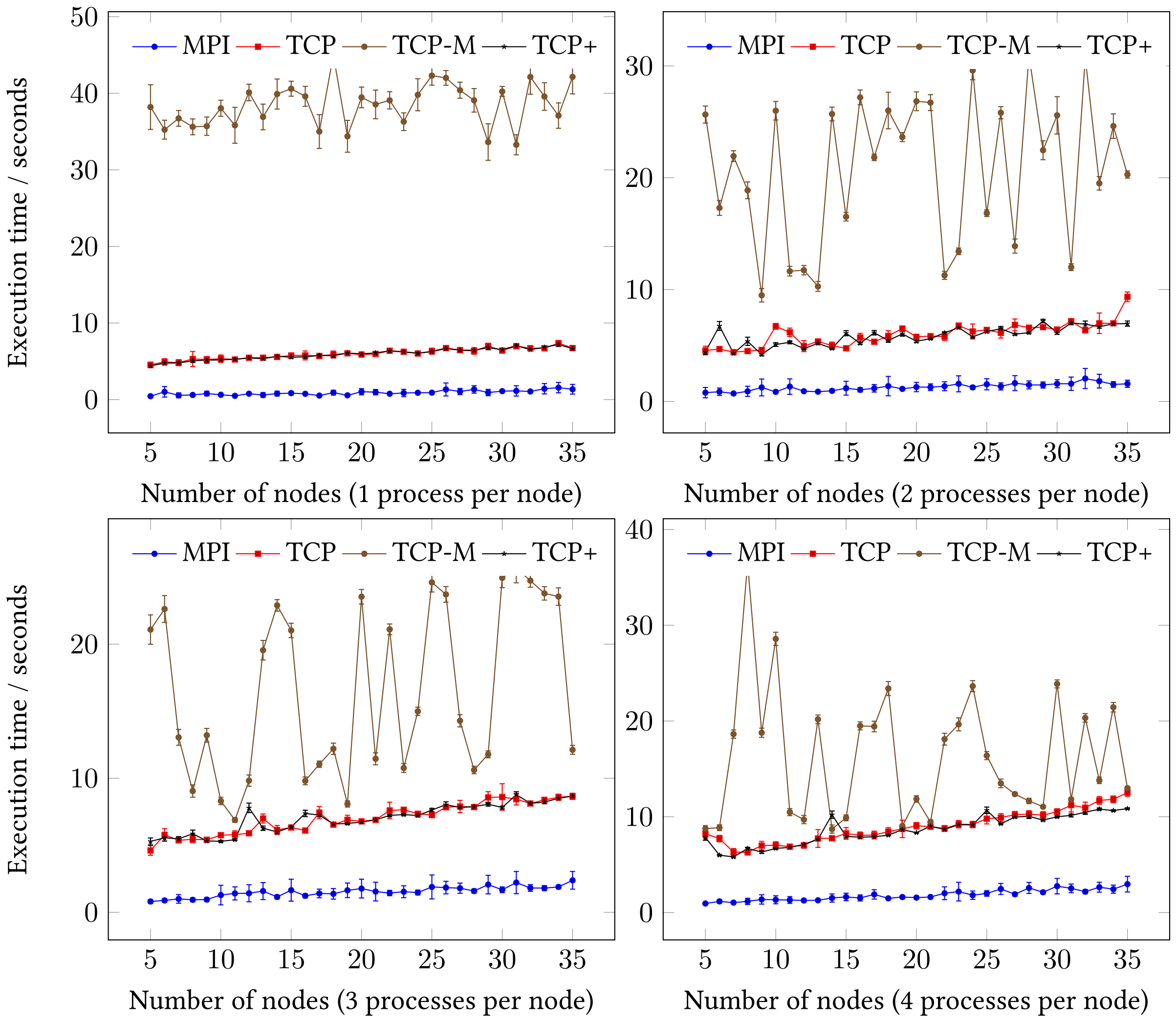


The result from last `markov` test case unsurprisingly mirror the results from the previous two: TCP in the default configuration is better than MPI, but not by much and only if the number of processes is not too high, and TCP+ delivers equal or better performance everywhere.

6.6.8 propagation2d 40 40

Looking at the analysis in section 9.2.8, we can see that this test case exclusively issues reduce operations to the backend, typically 42 or 83 per backend invocation. However, the size of the data to reduce is usually just 8 bytes, and the TCP backend does not support running reductions in parallel. In summary, we expect the TCP backend to perform very badly here, since the overhead caused by our protocol (see section 4.4) is significant here, and the backend needs to exchange more than 27000 messages to run the requested reductions.

Figure 6.9: Evaluation of `propagation2d 40 40` using 5-35 nodes with 1-4 processes per node

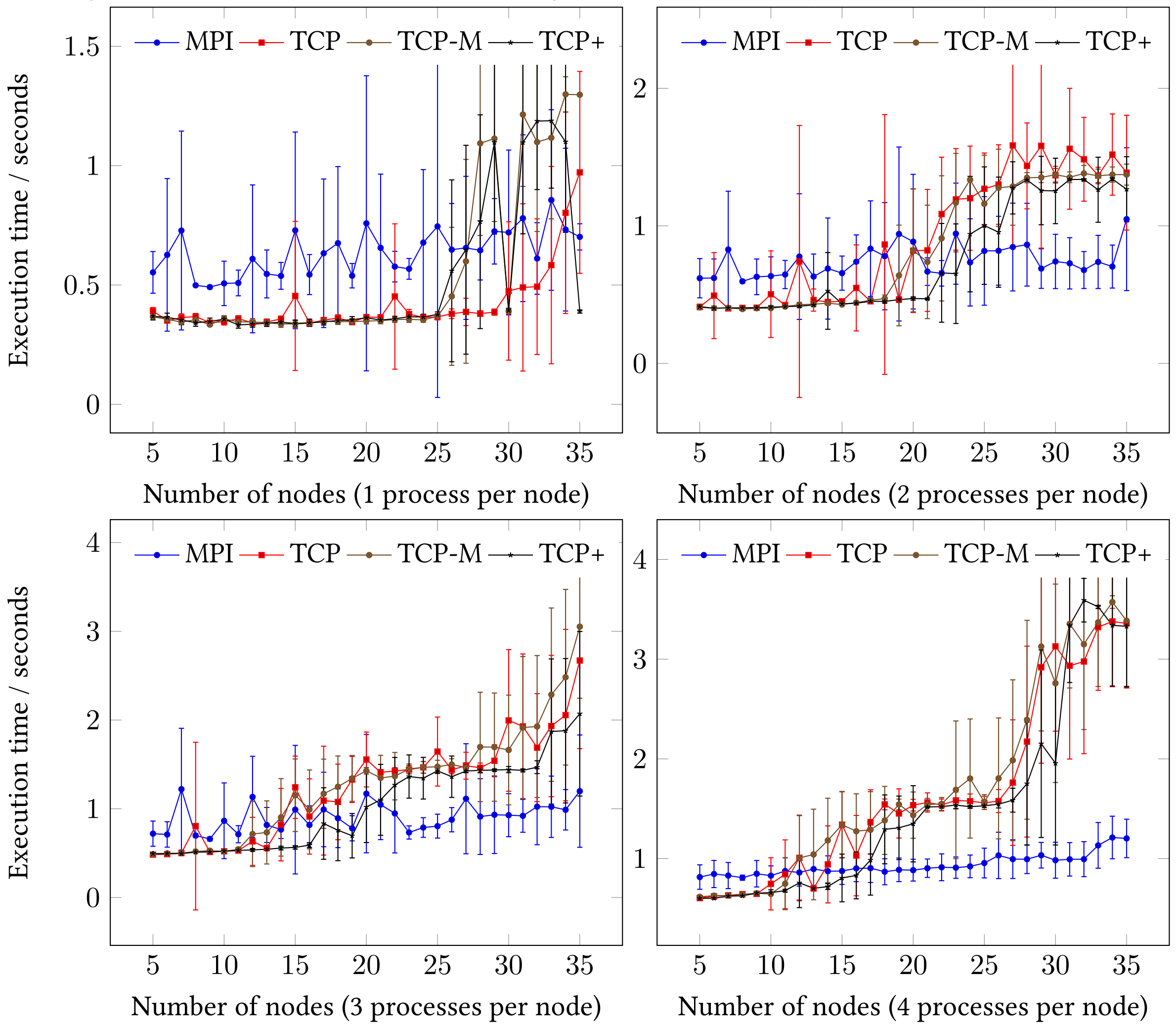


Finally, some interesting results! First, the TCP backend consistently performs worse than the MPI backend; in fact it takes at least three times as long everywhere. Second, in the previous test cases the TCP-M variant was always better than the default configuration which is not surprising given the reduction in exchanged messages from $O(n^2)$ to $O(n)$. However, in this case the quadratic algorithm (TCP) consistently beats the linear algorithm (TCP-M), and by no small amount. Looking back at the analysis above, this seems to suggest that for the specific case of many small reductions, the quadratic algorithm may actually be better.

6.6.9 spmv 5000

The analysis in section 9.2.9 shows one thing above all else: This test case barely utilizes the backend. It is worth noting that we noticed this when selecting the test cases and tried mitigating this problem by adjusting the command line arguments to cause the program to run longer and issue more work to the backend. However, we almost always encountered out-of-memory (OOM) errors when using different parameters. If the results are significant here, we expect the TCP backend to not perform very well, since the little work the backend gets almost always consists of one send/receive or reduce operation per backend invocation, meaning no parallelization is possible.

Figure 6.10: Evaluation of `spmv 5000` using 5-35 nodes with 1-4 processes per node

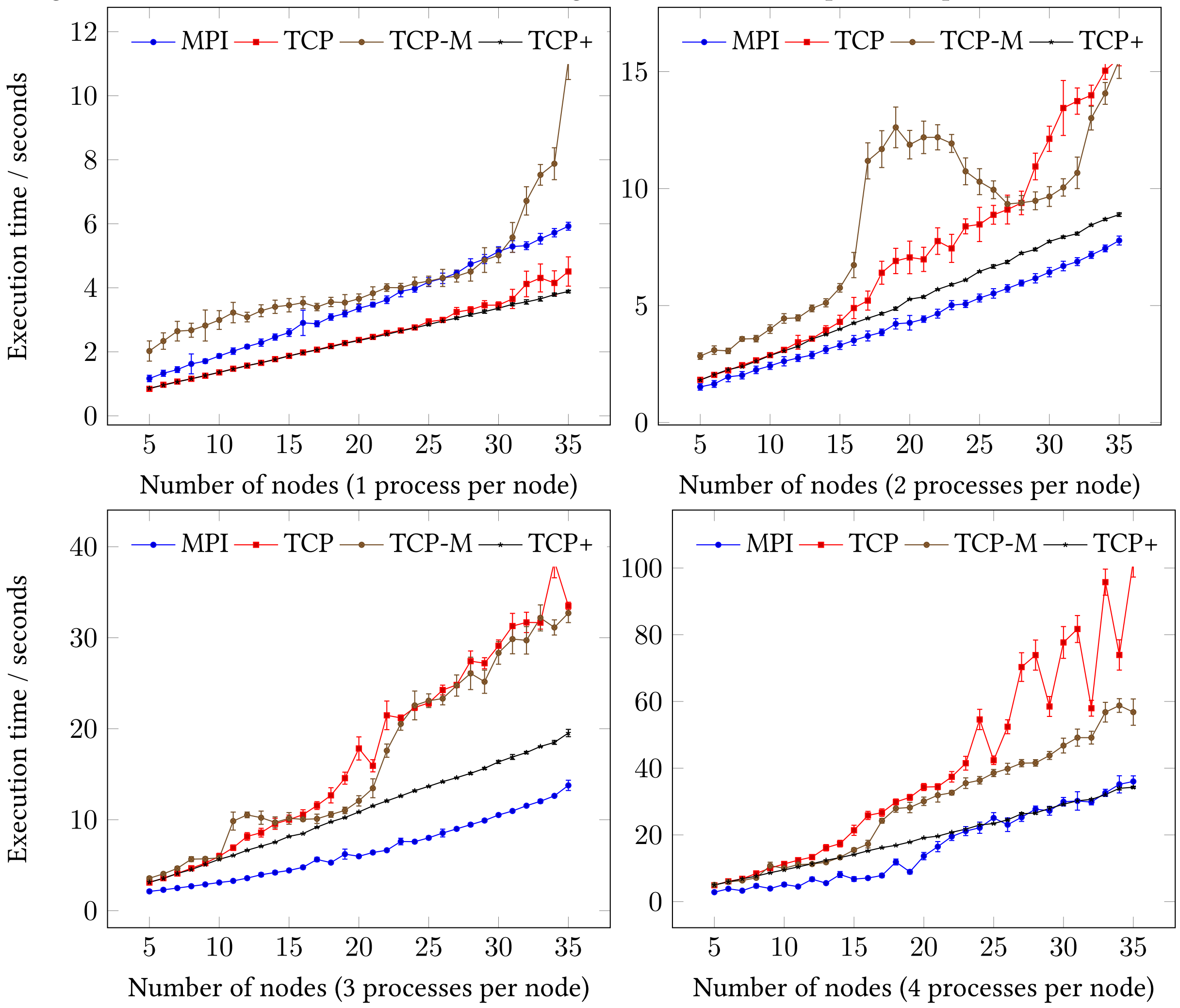


This test case appears to be another example of the TCP backend performing worse than the MPI backend, at least once enough processes are involved, but we expected as much given the analysis described above. However, the lower total execution time compared to the other test cases in combination with the relatively high sample standard deviation means that we probably should not give this result too much weight.

6.6.10 spmv2 150 1500

This test is is interesting: The analysis in section 9.2.10 shows that the program typically submits 9 send/receive operations to the backend which should allow for good parallelization. Furthermore, the message sizes are non-negligible, ranging from several hundred to several thousand bytes. However, on nearly half the backend invocations, the program submits no send/receive operations at all, but only a single reduction consisting of just 8 bytes. So, half of the time this program is well suited for the TCP backend, and the other half it is not. Therefore, it seems there is no clear prediction possible here regarding the performance to expect.

Figure 6.11: Evaluation of `spmv2 150 1500` using 5-35 nodes with 1-4 processes per node

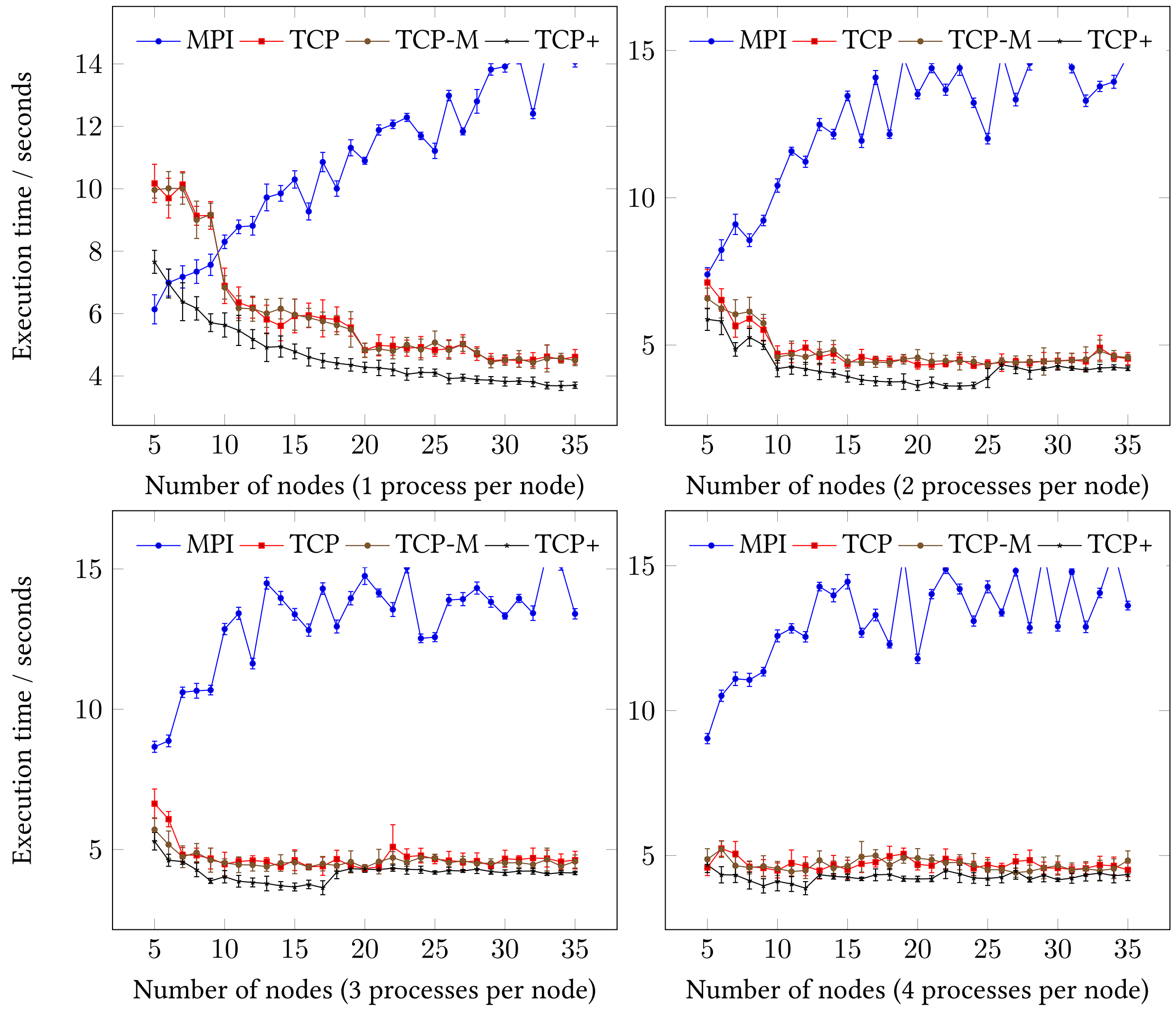


This is another result which is very interesting: First, the TCP backend appears to be both significantly better and significantly worse, depending the number of processes per node. However, we think that this is simply caused by the fact that more processes per node also means more processes in total. Second, the linear reduction algorithm (TCP-M) also performs surprisingly badly here, compared to default TCP environment. The TCP+ variant (which also uses the quadratic reduction algorithm) even outperforms it everywhere and is the closest to the clearly superior MPI backend. In total, we can conclude that there are clearly usage profiles where the quadratic reduction algorithm is better, and that having a higher limit on concurrently open connections is obviously also beneficial.

6.6.11 vsum 20000000

The test case presented here is another example of a program which only calls into the backend very infrequently: The analysis in section 9.2.11 shows a total of just 40 invocations to send/receive or reduce data. Among these, the bulk of the work given to the backend are send/receive operations with message sizes ranging from several hundred kilobytes to well over a megabyte. However, there are also 10 invocations containing a single reduction over just 32 bytes. In total, we expect that the TCP backend can perform well here, given the characteristics of the send/receive operations.

Figure 6.12: Evaluation of `vsum 20000000` using 5-35 nodes with 1-4 processes per node



This test case clearly seems to favor the TCP backend: The MPI backend is slower by a factor of at least two, except for very small process group sizes. Furthermore, all TCP variants perform almost identically here, suggesting that a a higher connection limit in no way guarantees better performance.

6.6.12 vsum2

This test case is similar to the previous one: Section 9.2.12 shows mostly multiple send/receive operations per backend invocation, with message sizes ranging from several kilobytes to almost half a megabyte. There are again some reductions over just 32 bytes of data, but overall the performance of the TCP backend should still be reasonable here.

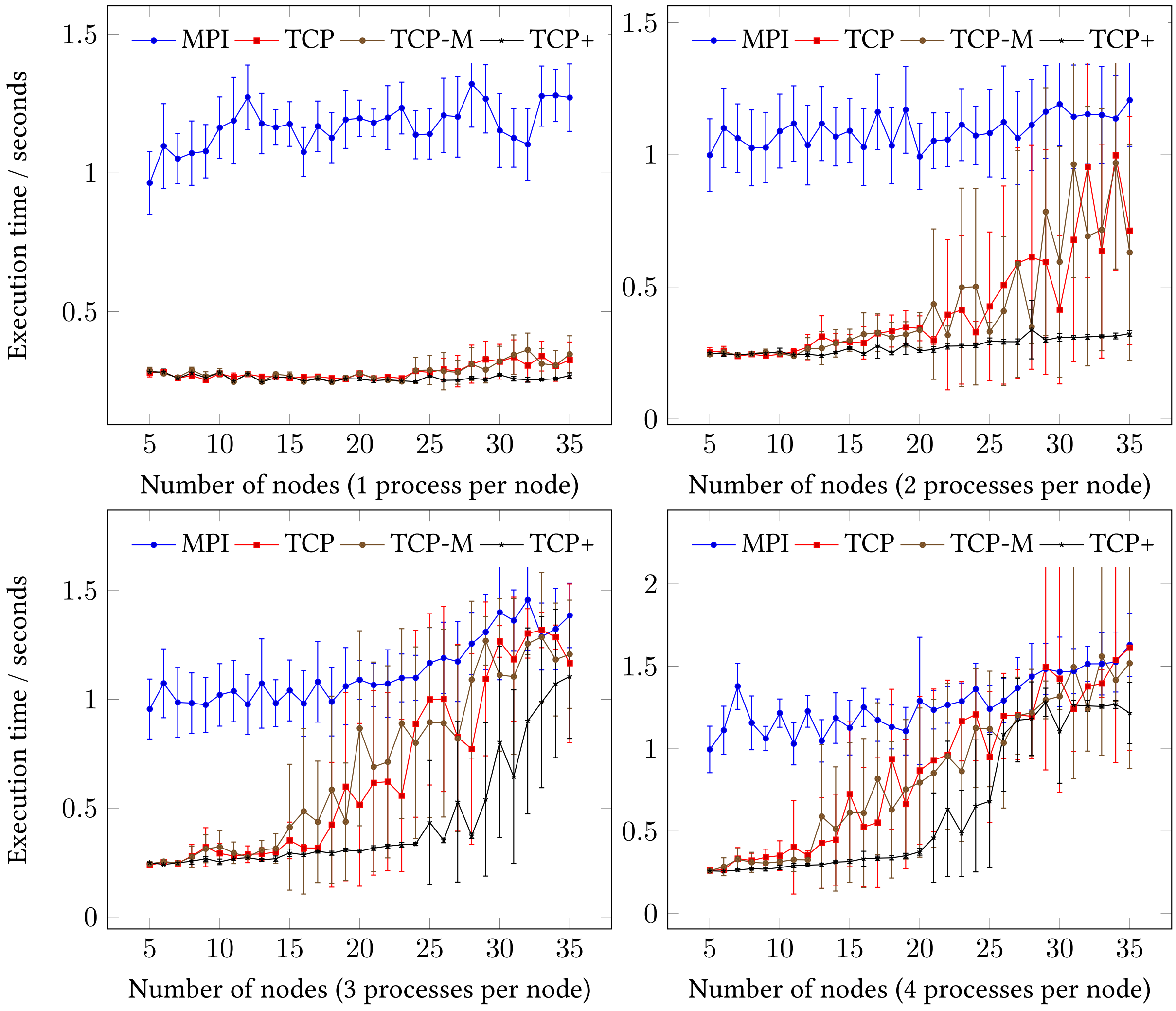


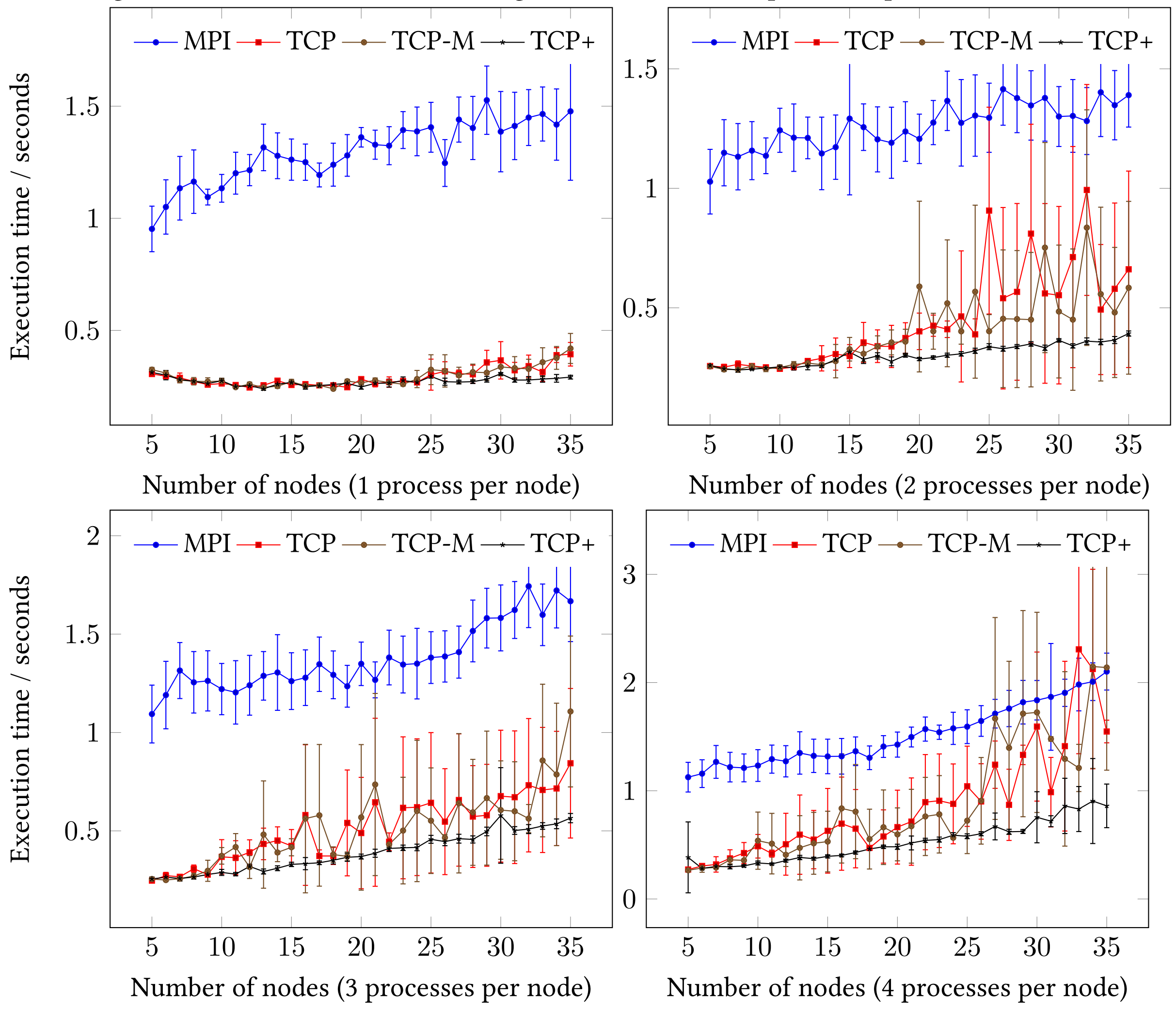
Figure 6.13: Evaluation of `vsum2` using 5-35 nodes with 1-4 processes per node

It should be noted that the `vsum2` program does not take any command line arguments, so we could not increase the total execution time to roughly match that seen in the other test cases. The results are still clear however: The TCP backend is much faster compared to the MPI backend, but not by as much as seen in the previous test case. Furthermore, the advantage of the TCP backend actually seems to be getting smaller with both the number of nodes and the number of processes per node, suggesting that larger tests might actually return results with the MPI backend on top.

6.6.13 vsum3

The analysis in section 9.2.13 show that the last case is similar to the previous two: Mostly send/receive operations with each backend invocation containing multiple such operations. The only difference is that now the message sizes range from several kilobytes to almost a megabyte, but in the end this should not affect the results much, the TCP backend should still perform well here.

Figure 6.14: Evaluation of `vsum3` using 5-35 nodes with 1-4 processes per node



Just like the previous test case, the `vsum3` program takes no arguments which is why the total execution time is lower than desired. The results are otherwise very similar to the previous test case: TCP is much faster, but its advantage shrinks with bigger process group sizes. However, in contrast to the previous test case, the TCP+ variant does not appear to be suffering from this problem, as its distance to the MPI backend is more or less consistent everywhere.

6.7 Discussion

The results presented in section 6.6 show a few interesting results which we want to summarize here shortly:

1. The new TCP backend appears to be faster than the existing MPI backend in most of the cases. This is surprising given that the MPI backend indirectly uses the code from OpenMPI (see section 6.1) which is certainly better optimized compared to our relatively simple, new TCP backend. The only good explanation we can offer for this, is that as described in section 4.5.5 our LAIK backend submits send operations messages in parallel to the corresponding receiving operations to the MPI compatibility layer (see figure 5.1), thus avoiding the problem described in section 4.2.5. In contrast the MPI backend is affected by this problem, as it contains code similar to the code shown in the referenced section.

2. The few cases were the TCP backend was significantly slower than the MPI backend all made heavy use of reductions (which the TCP backend does not run in parallel) over very small data buffers. In these cases, the overhead of the network protocol described in 4.4 and of the underlying TCP/IP stack becomes significant. Looking back at section 3.2, it might be a good idea to handle small reductions specially, e.g. by using UDP for these.

3. We expected that the TCP-M environment described in section 6.4.3 would always perform better than the default TCP environment, given that it reduces the number of messages required for a reduction from $O(n^2)$ to $O(n)$. However, at least for very small reduction sizes, this assumption does not appear to hold.

In total, we are very pleased with the results: Despite the fact that the new TCP backend was implemented without special attention to optimizing the performance besides the points described in chapter 4, it still can compete with the existing MPI backend in most test cases, often times even outperforming it. Furthermore, the results have also shown us clearly that the backend LAIK uses makes a significant difference in total execution time, regardless of the test case. From this, we conclude that optimizing the backends is an important component in bringing LAIK on par with existing inter-process communication solution such as plain MPI.

7 Future Work

In this chapter we want to introduce some ideas for future work resulting from the knowledge gained when designing, implementing and evaluating the TCP backend. We will first discuss possible improvements to LAIK's backend API and then explain how the performance of the new TCP backend could be further improved. Finally, we will motivate why our approach to *message identification* could be an interesting way of adding fault tolerance transparently to MPI applications running in co-scheduling environments.

7.1 Improvements to LAIK's Backend API

In section 4.1 we have explained why we initially did not use LAIK's backend API to interface with LAIK, but rather chose to re-implement a subset of the MPI API. Unfortunately, the points listed there are still largely valid today. While it is not surprising that an API design with only one real implementation turns out to be less than ideal, we think that now that there is a second, independent implementation, it is time to revisit that API design. In our opinion, the current API is too complicated and does too much, which makes it both difficult to use and difficult to implement.

We therefore suggest to redesign the API around a basic, synchronous send/receive design: Essentially the API should just contain the following functions:

```c
int laik_backend_init (void);
int laik_backend_finish (void);
int laik_backend_get_size (void);
int laik_backend_get_rank (void);
int laik_backend_send (int receiver, void* buffer, size_t length);
int laik_backend_receive (int sender, void* buffer, size_t length);
```

The first two functions would start/stop the backend, the next two determine the size of the process group and the local rank, and the last two functions can be used to synchronously send and receive binary blobs to/from a specific process (the backend may implement some buffering to make the actual transmissions asynchronous). This design differs in a few crucial points from the currently existing API:

1. LAIK would no longer inform the backend when it wants to make changes to the process group. Instead, it would determine the initial size and rank once on startup, and then always use those when talking to the backend. If LAIK wants to modify the process group for some reason it can still do so, but it has to translate the new values for size and rank back to the old (global) values expected by the backend. This modification would allow both making LAIK more flexible with regard to the modifications it can make to the process group as well as relieve the backend of the burden of supporting these changes.
2. There are no reduction functions present anymore. While this may seem like a glaring omission at first, LAIK *does* already contain a distributed software reduction algorithm in the MPI backend which could be extracted and generalized to use the send/receive primitives shown above. As LAIK supports a much wider variety of reductions compared to e.g. MPI already, chances are that this software reduction algorithm would be used most

of the time anyway. However, it is of course still possible to add an optional extension to the backend API for reductions directly supported by the backend. But given the fact that the types of reductions supported natively by a backend will vary widely (think for example of shared-memory based backends vs. network based backends), it does not make sense to have reduction support in the core backend API.

3. The send/receive functions accept/return simple binary blobs instead of run-time typed data buffers. We think that this is appropriate, given that there are probably not many HPC clusters with nodes of different architectures. However, if LAIK still wants to support such setups (which we do not think it should), it can easily marshal the data itself before handing it off to the backend if necessary. This should also not be much of a performance issue, given that different architectures typically imply different physical machines, which means message passing works by using some kind of interconnect that is likely much slower than the memory access required for data marshalling.
4. The `int` return values of the functions could be used by the backend to return detailed error codes as negative values (a technique common in the Linux kernel and related projects), something which is currently not possible at all: As of now, the only choice a backend has if it encounters an error is to terminate the whole program forcefully, which is less than ideal given that LAIK is a library. With proper error reporting, LAIK could either attempt to remedy the error itself (for example by retrying an operation) or properly report the problem to the main application.

The simplified API in combination with proper error reporting could also be used to extend LAIK with the ability to not only support *shrinking* the process group, but also *extending* it. While the former would be implemented by simply not calling the backend with the ranks of the removed processes anymore, the latter could be achieved using the following approach:

1. LAIK would regularly call `laik_backend_get_size()` to determine if the backend has noticed a new process which has joined the process group since the last check. If so, it would update its internal data structures and start distributing work to the new process.
2. LAIK would then start calling the `laik_backend_send()` and `laik_backend_receive()` functions with new receiver/sender ranks derived from the new process group size.

We realize that the new API design introduced above is heavily influenced by the TCP backend we have introduced in this work, just as the existing API design was heavily influenced by the existing MPI backend. However, given that we both implemented the current LAIK backend API and the subset of the MPI API used by LAIK's MPI backend, we are quite certain that a simpler, and less abstract backend interface would a long way in making LAIK easier to extend in the future.

7.2 Improvements to the TCP Backend

7.2.1 Reducing the Amount of System Calls

Section 4.4 explained the basic network protocol. The implementation follows the messages shown there closely, so each arrow in the diagrams corresponds to an individual `send` or `recv` system call. Since the TCP sockets used have the TCP_NODELAY option enabled as explained in section 4.3, this likely means that every such arrow actually corresponds to an individual TCP segment transported over the network. This is a performance problem: Both making a system call as well as sending data over the network should be done as infrequently as possible, as they are both expensive operations, especially if they are executed hundreds or thousands of times (section 9.2 shows that these numbers are realistic even in small use cases).

The obvious solution to this problem is to incrementally construct the messages to be sent in a buffer and then submit that buffer to the `send` function once the message is complete. However, this approach means that an additional copy of the message is created when sending which is not ideal. A better idea would be to use the TCP_CORK [15] socket option offered by recent versions of the Linux kernel to prevent the message from being sent out before it has been submitted to the operating system completely. Unfortunately, this approach still leaves the problem of using multiple system call per message unsolved.

We think that the ideal solution should use the scatter/gather I/O offered by the `sendmsg()` [21] and `recvmsg()` [25] system calls: The socket abstraction class shown in figure 5.1 should gain the ability to queue submitted buffers and only submit them to operating system once the calling code signals that the current message is complete. Implementing a corresponding solution for receiving a message unfortunately seems more complicated, since the socket abstraction class does not know enough about the network protocol to properly split up incoming messages. Furthermore, great care has to be taken to ensure that buffers submitted to the socket abstraction class are released once they are no longer needed, but not before.

7.2.2 Automatic Selection of the Reduction Strategy

In section 6.4.3 we have introduced an option of the new TCP backend which lowers the amount of messages generated by the reduction algorithm from $O(n^2)$ to $O(n)$ (where n is the size of the process group). Chapter 6 has shown that this modification is beneficial in almost all cases, except when frequently reducing very small buffers (see section 6.6.8).

We think that this shows a fundamental problem: Reducing using a dedicated master process lowers the amount of messages required, but introduces an additional round trip. Depending on the specific application and network in use this may lead to a net gain or loss in performance. It would be interesting to investigate the precise parameters where net gains turn into net losses and to use that knowledge to dynamically switch from the centralized to the distributed reduction algorithm, depending on the current reduction task and the observed environment.

7.2.3 Better Connection Cache Eviction Strategies

In order to solve the problem described in section 4.2.2, we had to limit the total number of connections being kept open concurrently (see section 4.5.2). This implies that we need some sort of eviction strategy to decide which connections to keep and which to drop once we reach the limit. Our implementation uses the simplest possible approach here and just drops all connections once the limit has been reached. While this approach has the advantage that different communication patters in different stages of the program execution can be handled well, it is clearly not ideal.

A possible improvement would be to investigate whether existing cache eviction algorithms (such as those used by CPU caches) could be applied here. However, we expect that the decision which of those algorithms to actually use is not trivial: Different applications may have wildly different communication patterns and an eviction algorithm that works well for one application may not do so for others. Overall, we suspect that solving this problem will require profiling the application dynamically during runtime (as mentioned in section 7.2.2), and then select an eviction algorithm suited for the specific communication pattern observed.

7.3 Fault Tolerance in MPI Using Unique Message Identifiers

Section 3.3 shows that bringing fault tolerance to MPI implementations or even the standard itself is an active research area. In this work, we have shown an approach for fault tolerance which is based on generating unique message identifiers (see section 4.5.4) and then adding the option to retain sent messages (see section 4.5.7) so they can be requested a second time later on (see section 4.4.3). However, we did this while still only using the MPI API to communicate between the backend used by LAIK and our actual implementation, as shown in figure 5.1.

Since we only used the MPI API to generate the message identifiers, it seems possible to bring a similar solution to existing MPI implementations without modifying the MPI standard at all. While our implementation has some obvious limitations (such as the fact that messages are retained in memory only), we think that the fundamental idea of having unique message identifiers and retransmission requests is solid and could be used to add fault tolerance transparently to existing MPI applications.

However, it is worth noting that this approach has two fundamental limitations. Since it relies on the fact that both the sender and the receiver can compute the message identifier independently and come to the same result, it will obviously not work for applications using MPI in a nondeterministic way (i.e. where the exact identity of the next message to receive is not fixed). Furthermore, without additional measures, restarting a failed process means that this process has to redo its entire part of the computation, likely causing the other processes to block on message reception until the replacement process has caught up with the rest of the process group. This implies that restarting a single failed process is no faster than restarting the entire process group when it comes to the total execution time. It is however much faster, when we only look at the total amount of CPU time used for the computation, so this approach might still be very interesting in co-scheduling environments.

8 Conclusion

We have designed, implemented, and evaluated a new backend for LAIK based on native TCP sockets. Using the results from the evaluation (as presented in chapter 6) we can now answer the *research question* from chapter 1, namely how the performance of our new TCP backend compares to that of the existing MPI backend in combination with a widely used MPI implementation: The new TCP backend delivers comparable performance in most test cases, with many test cases even showing significant performance improvements.

However, delivering competitive performance is not the only result from this work: The problems described in section 4.2 can serve as a blueprint for other LAIK backends to be designed in the future, at least if they are also based around send/receive primitives. In fact, the problem described in section 4.2.5 also affects the MPI backend and fixing it could potentially deliver dramatic performance increases here.

Besides performance considerations we also designed and implemented a working fault tolerance mechanism based on unique message identifiers. Sections 5.3.2, 5.3.3, and 5.3.4 show how this approach can help deal with failed processes transparently, i.e. without requiring any modification to the program or even the core components of LAIK. However, we want to stress that adding fault tolerance was not a priority when we started working on the new TCP backend, but rather came as a natural extension once we had the message identifiers from section 4.5.4 and the network protocol described in section 4.4 figured out.

Actually, we think that the code to generate the message identifiers (shown in full in section 4.5.4) is our most important result. This code lives inside our MPI compatibility layer (see figure 5.1), and consequently only has access to information any MPI implementation contains. It still is amazingly simple, as it essentially boils down to attaching serial numbers to messages within a common message flow by using a hash table. However, despite its simplicity it still allows making LAIK applications fault tolerant today which is a nice final result.

9 Appendix

9.1 Cross-building LAIK on amd64 for arm64

The following steps are based on [2] and describe how to cross-build LAIK on a Debian based amd64 system so that the resulting binaries can run on an arm64 system:

1. Enable Debian's multiarch support for arm64:

```
$ dpkg --add-architecture arm64
$ apt update
[...]
$
```

2. Install the necessary packages:

    - `crossbuild-essential-arm64`
    - `libglib2.0-dev:arm64`
    - `libgomp1:arm64`
    - `libmosquitto-dev:arm64`
    - `libopenmpi-dev:arm64`
    - `libpapi-dev:arm64`
    - `libprotobuf-c-dev:arm64`
    - `qemu-user`
    - `qemu-user-binfmt`

3. In LAIK's source tree, create a build directory, configure CMake, build and test:

```
$ mkdir cross-build
$ cd cross-build
$ PKG_CONFIG_PATH=/usr/lib/aarch64-linux-gnu/pkgconfig cmake -D CMAKE_C_COMPILER=
    aarch64-linux-gnu-gcc -D CMAKE_CXX_COMPILER=aarch64-linux-gnu-g++ -D
    CMAKE_BUILD_TYPE=Release -D skip-missing=off ..
[...]
$ make
[...]
$ ctest
[...]
$
```

9.2 Test Case Analysis

In order to get an overview of what each test case required the backend to do, we patched LAIK as shown in section 9.4.1 and then ran the script shown in section 9.4.3 (using the configuration file shown in section 9.4.2). This script started each test cases with 10 processes and then collected the generated data which is presented in the following sections.

9.2.1 jac1d 125000

Running the test case with the command line `jac1d 125000` using 10 processes caused the following things to happen in the backend, summarized across all processes:

- The backend was called a total of 560 times to send, receive, and/or reduce data, see figure 9.1, 9.2, and 9.3 for details about the kind of work submitted to the backend.
- The backend was called a total of 0 times because LAIK's view of the process group had changed, i.e. because one or more process were removed.
- The backend sent 900 chunks of data, see figure 9.4 for details about their sizes.
- The backend received 900 chunks of data, see figure 9.5 for details about their sizes.
- The backend reduced 60 chunks of data, see figure 9.6 for details about their sizes.

Figure 9.1: jac1d 125000: Number of times the backend was called with n send operations

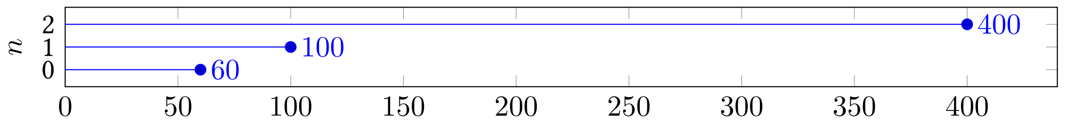


Figure 9.2: jac1d 125000: Number of times the backend was called with n receive operations

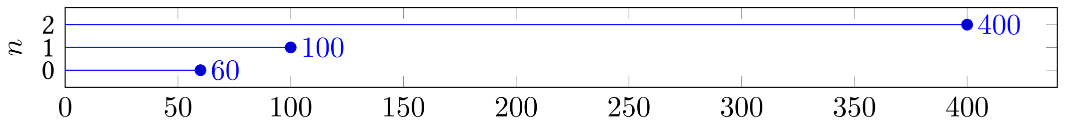


Figure 9.3: jac1d 125000: Number of times the backend was called with n reduce operations

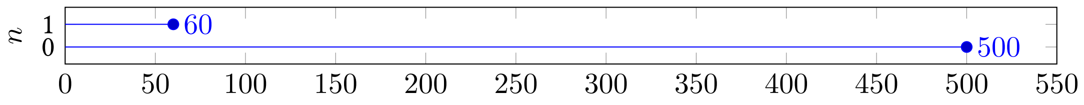


Figure 9.4: jac1d 125000: Number of times the backend sent n bytes

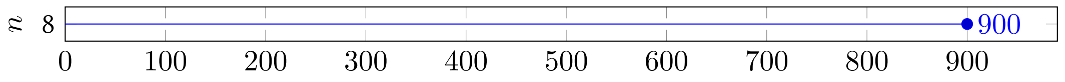


Figure 9.5: jac1d 125000: Number of times the backend received n bytes

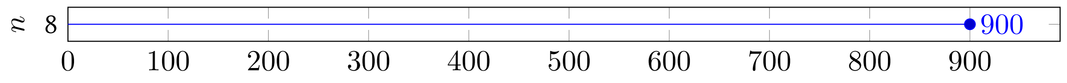
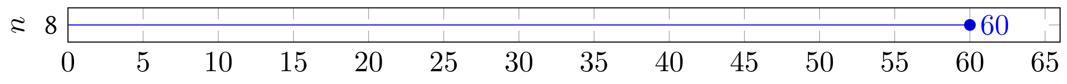


Figure 9.6: jac1d 125000: Number of times the backend reduced n bytes



9.2.2 jac2d 11000

Running the test case with the command line `jac2d 11000` using 10 processes caused the following things to happen in the backend, summarized across all processes:

- The backend was called a total of 550 times to send, receive, and/or reduce data, see figure 9.7, 9.8, and 9.9 for details about the kind of work submitted to the backend.
- The backend was called a total of 0 times because LAIK's view of the process group had changed, i.e. because one or more process were removed.
- The backend sent 1900 chunks of data, see figure 9.10 for details about their sizes.
- The backend received 1900 chunks of data, see figure 9.11 for details about their sizes.
- The backend reduced 50 chunks of data, see figure 9.12 for details about their sizes.

Figure 9.7: jac2d 11000: Number of times the backend was called with n send operations

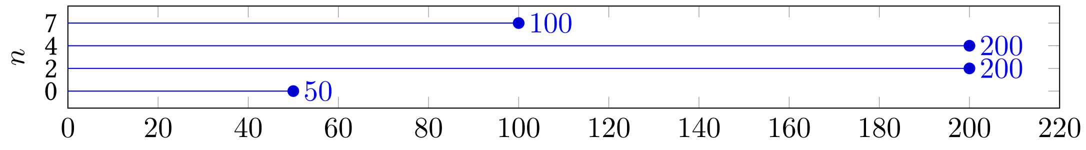


Figure 9.8: jac2d 11000: Number of times the backend was called with n receive operations

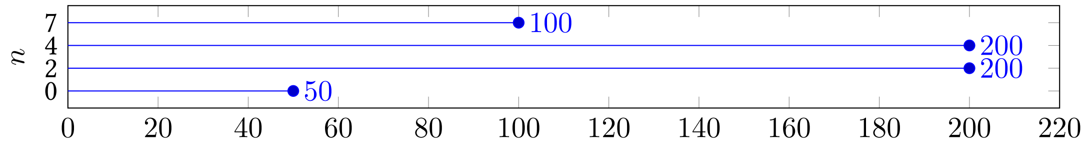


Figure 9.9: jac2d 11000: Number of times the backend was called with n reduce operations

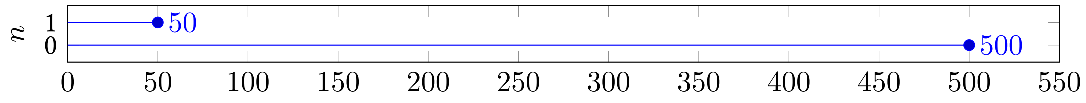


Figure 9.10: jac2d 11000: Number of times the backend sent n bytes

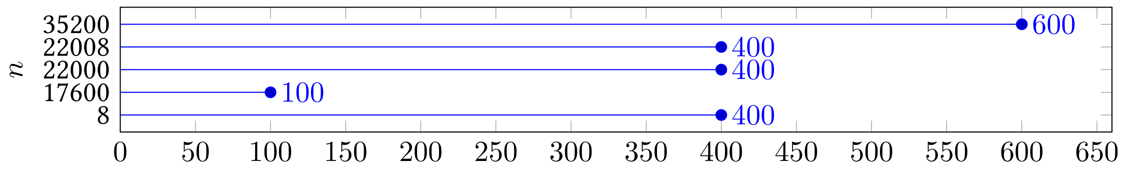


Figure 9.11: jac2d 11000: Number of times the backend received n bytes

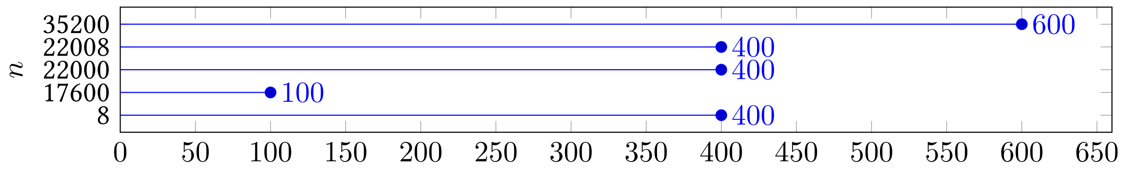
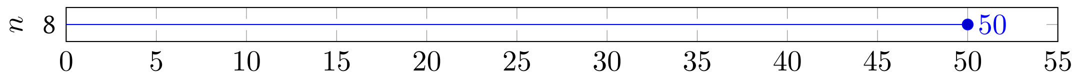


Figure 9.12: jac2d 11000: Number of times the backend reduced n bytes



9.2.3 jac3d 375

Running the test case with the command line `jac3d 375` using 10 processes caused the following things to happen in the backend, summarized across all processes:

- The backend was called a total of 550 times to send, receive, and/or reduce data, see figure 9.13, 9.14, and 9.15 for details about the kind of work submitted to the backend.
- The backend was called a total of 0 times because LAIK's view of the process group had changed, i.e. because one or more process were removed.
- The backend sent 3300 chunks of data, see figure 9.16 for details about their sizes.
- The backend received 3300 chunks of data, see figure 9.17 for details about their sizes.
- The backend reduced 50 chunks of data, see figure 9.18 for details about their sizes.

Figure 9.13: jac3d 375: Number of times the backend was called with n send operations

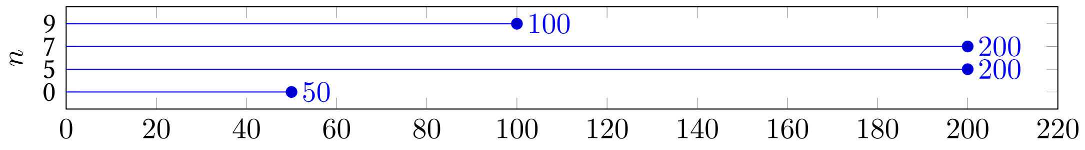


Figure 9.14: jac3d 375: Number of times the backend was called with n receive operations

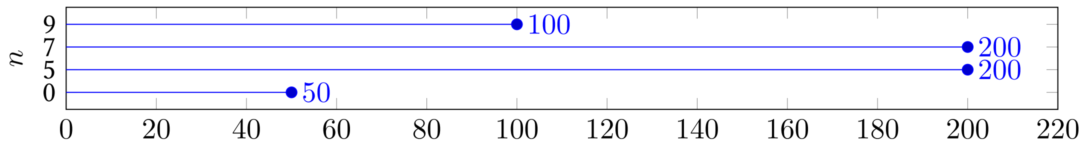


Figure 9.15: jac3d 375: Number of times the backend was called with n reduce operations

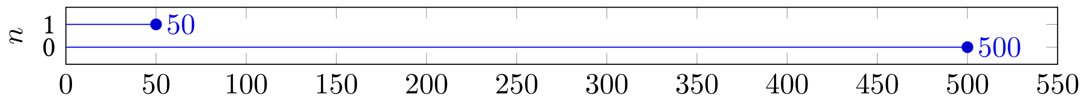


Figure 9.16: jac3d 375: Number of times the backend sent n bytes

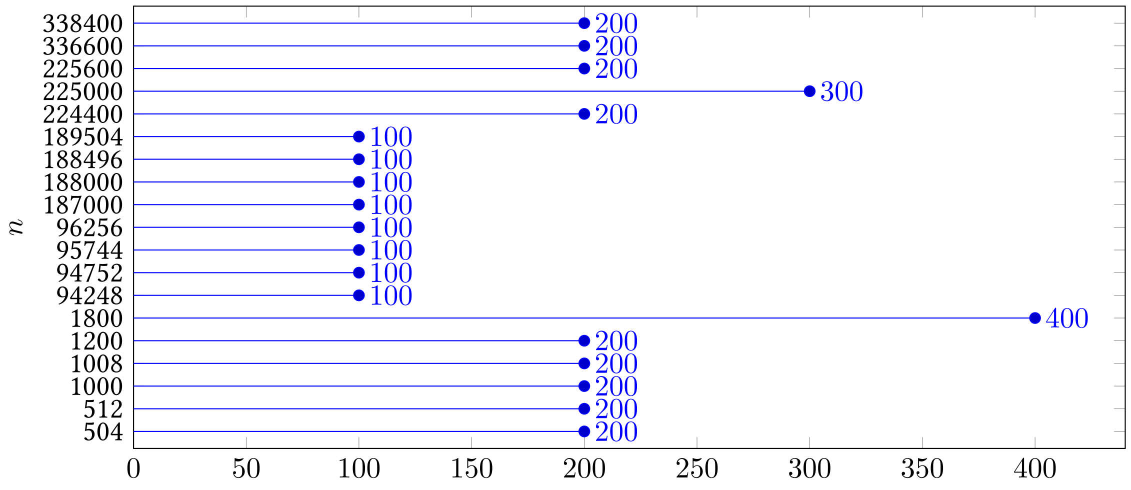


Figure 9.17: jac3d 375: Number of times the backend received n bytes

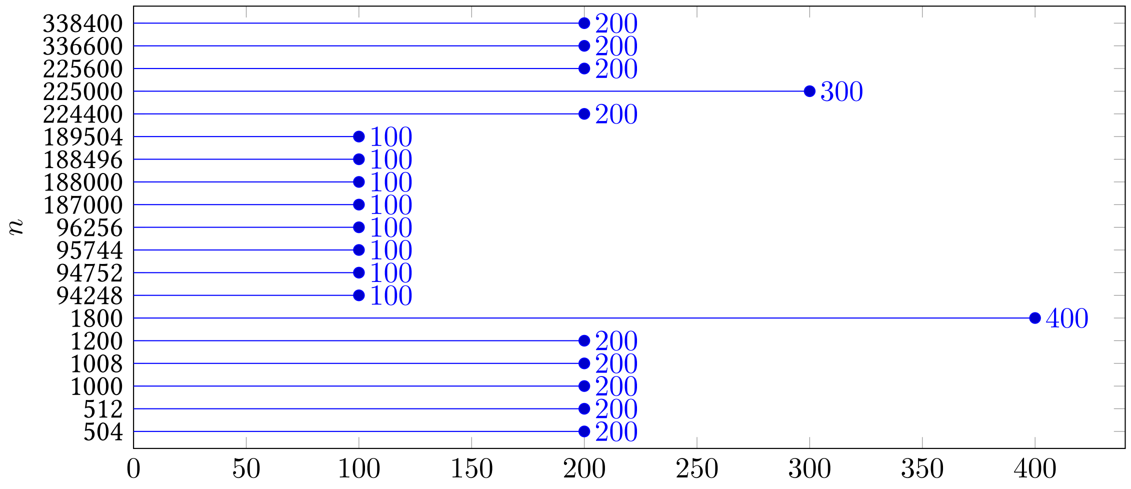
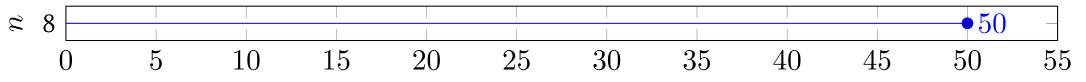


Figure 9.18: jac3d 375: Number of times the backend reduced n bytes



9.2.4 jac3d -g 375

Running the test case with the command line `jac3d -g 375` using 10 processes caused the following things to happen in the backend, summarized across all processes:

- The backend was called a total of 450 times to send, receive, and/or reduce data, see figure 9.19, 9.20, and 9.21 for details about the kind of work submitted to the backend.
- The backend was called a total of 0 times because LAIK's view of the process group had changed, i.e. because one or more process were removed.
- The backend sent 2800 chunks of data, see figure 9.22 for details about their sizes.
- The backend received 2800 chunks of data, see figure 9.23 for details about their sizes.
- The backend reduced 50 chunks of data, see figure 9.24 for details about their sizes.

Figure 9.19: jac3d -g 375: Number of times the backend was called with n send operations

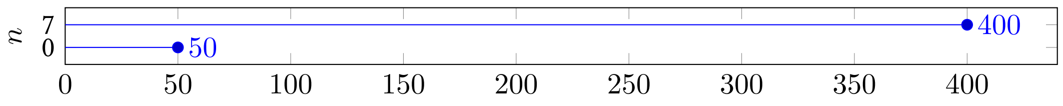


Figure 9.20: jac3d -g 375: Number of times the backend was called with n receive operations

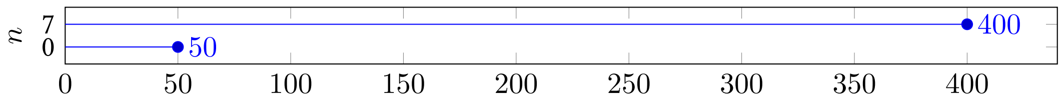


Figure 9.21: jac3d -g 375: Number of times the backend was called with n reduce operations

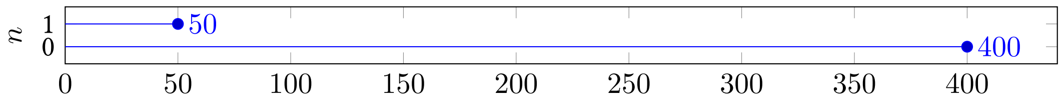


Figure 9.22: jac3d -g 375: Number of times the backend sent n bytes

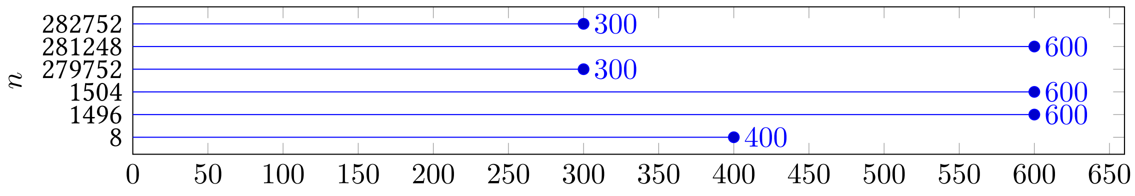


Figure 9.23: jac3d -g 375: Number of times the backend received n bytes

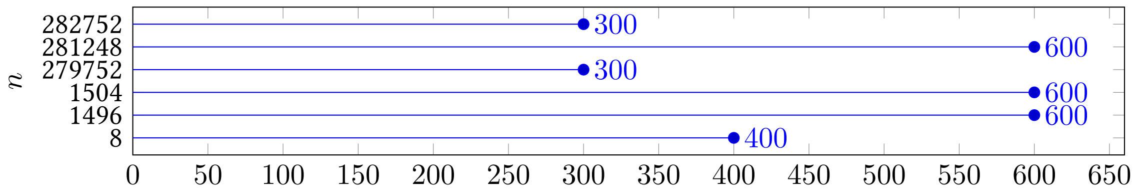
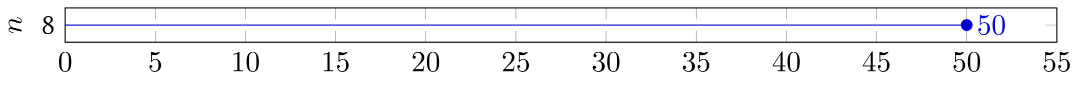


Figure 9.24: jac3d -g 375: Number of times the backend reduced n bytes



9.2.5 markov 400 4000

Running the test case with the command line `markov 400 4000` using 10 processes caused the following things to happen in the backend, summarized across all processes:

- The backend was called a total of 330 times to send, receive, and/or reduce data, see figure 9.25, 9.26, and 9.27 for details about the kind of work submitted to the backend.
- The backend was called a total of 0 times because LAIK's view of the process group had changed, i.e. because one or more process were removed.
- The backend sent 2727 chunks of data, see figure 9.28 for details about their sizes.
- The backend received 2727 chunks of data, see figure 9.29 for details about their sizes.
- The backend reduced 0 chunks of data, see figure 9.30 for details about their sizes.

Figure 9.25: markov 400 4000: Number of times the backend was called with n send operations

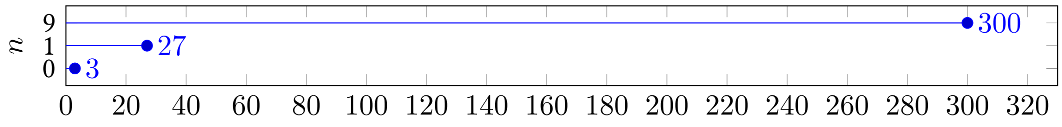


Figure 9.26: markov 400 4000: Number of times the backend was called with n receive operations

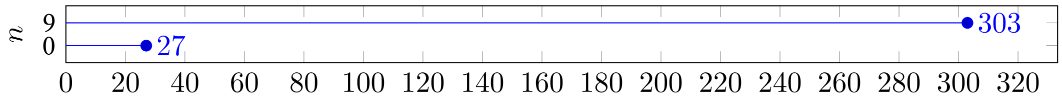


Figure 9.27: markov 400 4000: Number of times the backend was called with n reduce operations

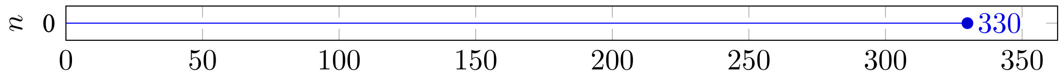


Figure 9.28: markov 400 4000: Number of times the backend sent n bytes

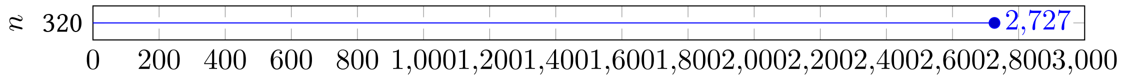


Figure 9.29: markov 400 4000: Number of times the backend received n bytes

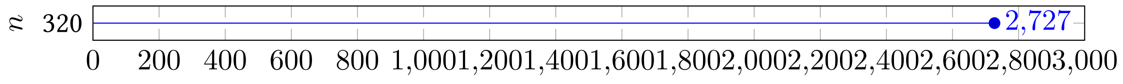
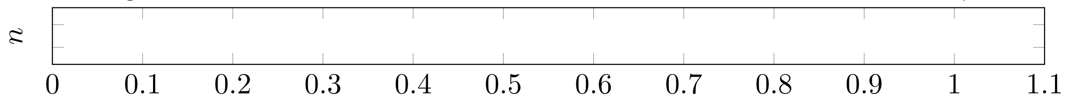


Figure 9.30: markov 400 4000: Number of times the backend reduced n bytes



9.2.6 markov2 400 4000

Running the test case with the command line `markov2 400 4000` using 10 processes caused the following things to happen in the backend, summarized across all processes:

- The backend was called a total of 100 times to send, receive, and/or reduce data, see figure 9.31, 9.32, and 9.33 for details about the kind of work submitted to the backend.
- The backend was called a total of 0 times because LAIK's view of the process group had changed, i.e. because one or more process were removed.
- The backend sent 0 chunks of data, see figure 9.34 for details about their sizes.
- The backend received 0 chunks of data, see figure 9.35 for details about their sizes.
- The backend reduced 910 chunks of data, see figure 9.36 for details about their sizes.

Figure 9.31: markov2 400 4000: Number of times the backend was called with n send operations

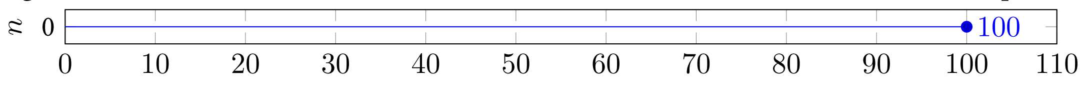


Figure 9.32: markov2 400 4000: Number of times the backend was called with n receive operations

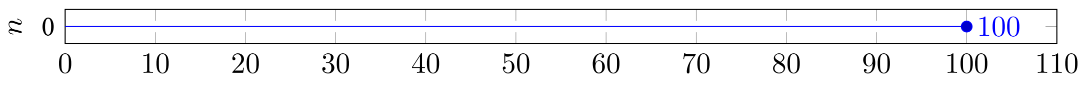


Figure 9.33: markov2 400 4000: Number of times the backend was called with n reduce operations

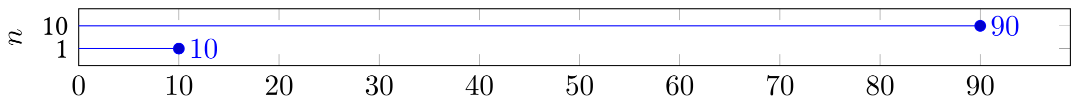


Figure 9.34: markov2 400 4000: Number of times the backend sent n bytes

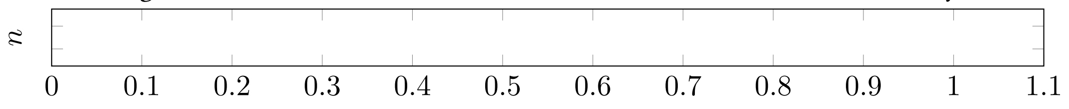


Figure 9.35: markov2 400 4000: Number of times the backend received n bytes

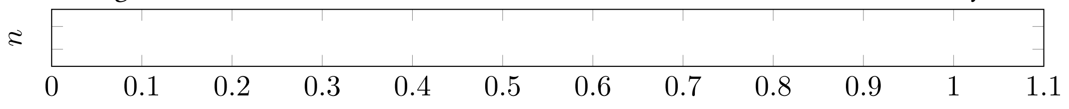
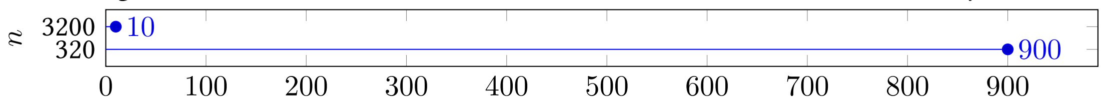


Figure 9.36: markov2 400 4000: Number of times the backend reduced n bytes



9.2.7 markov2 -f 400 4000

Running the test case with the command line `markov2 -f 400 4000` using 10 processes caused the following things to happen in the backend, summarized across all processes:

- The backend was called a total of 100 times to send, receive, and/or reduce data, see figure 9.37, 9.38, and 9.39 for details about the kind of work submitted to the backend.
- The backend was called a total of 0 times because LAIK's view of the process group had changed, i.e. because one or more process were removed.
- The backend sent 0 chunks of data, see figure 9.40 for details about their sizes.
- The backend received 0 chunks of data, see figure 9.41 for details about their sizes.
- The backend reduced 910 chunks of data, see figure 9.42 for details about their sizes.

Figure 9.37: markov2 -f 400 4000: Number of times the backend was called with n send operations

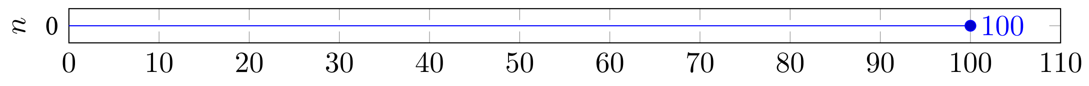


Figure 9.38: markov2 -f 400 4000: Number of times the backend was called with n receive operations

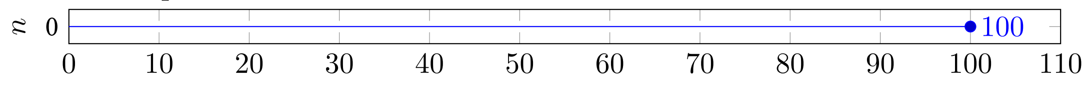


Figure 9.39: markov2 -f 400 4000: Number of times the backend was called with n reduce operations

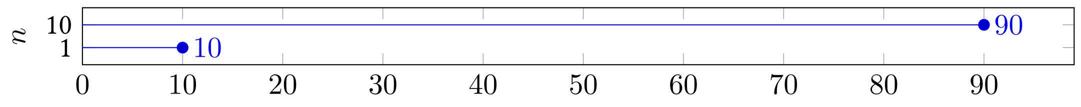


Figure 9.40: markov2 -f 400 4000: Number of times the backend sent n bytes

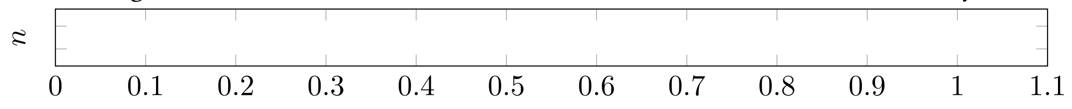


Figure 9.41: markov2 -f 400 4000: Number of times the backend received n bytes

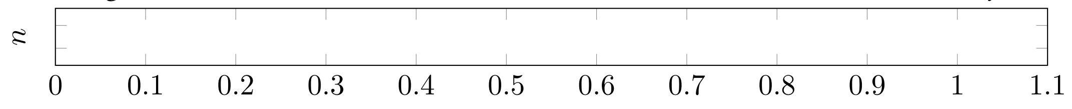
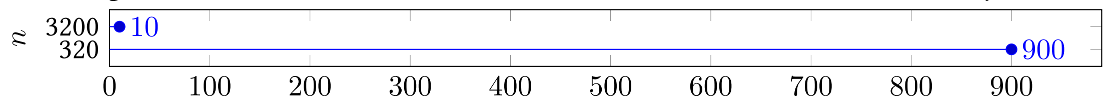


Figure 9.42: markov2 -f 400 4000: Number of times the backend reduced n bytes



9.2.8 propagation2d 40 40

Running the test case with the command line `propagation2d 40 40` using 10 processes caused the following things to happen in the backend, summarized across all processes:

- The backend was called a total of 420 times to send, receive, and/or reduce data, see figure 9.43, 9.44, and 9.45 for details about the kind of work submitted to the backend.
- The backend was called a total of 0 times because LAIK's view of the process group had changed, i.e. because one or more process were removed.
- The backend sent 0 chunks of data, see figure 9.46 for details about their sizes.
- The backend received 0 chunks of data, see figure 9.47 for details about their sizes.
- The backend reduced 27316 chunks of data, see figure 9.48 for details about their sizes.

Figure 9.43: propagation2d 40 40: Number of times the backend was called with n send operations

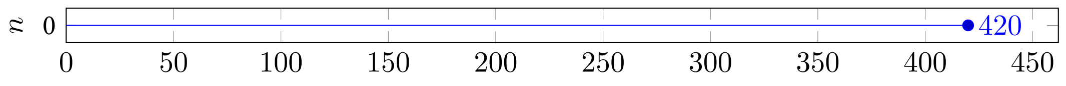


Figure 9.44: propagation2d 40 40: Number of times the backend was called with n receive operations

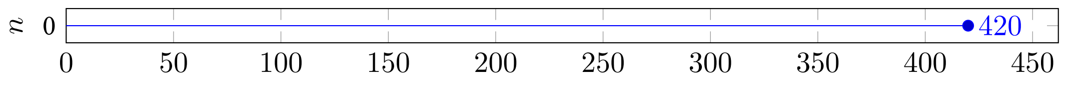


Figure 9.45: propagation2d 40 40: Number of times the backend was called with n reduce operations

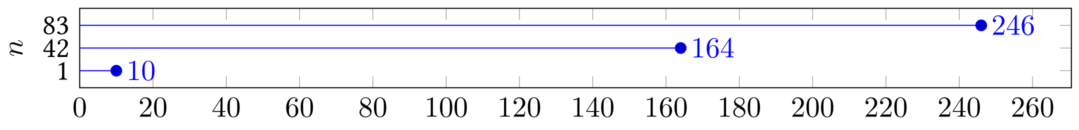


Figure 9.46: propagation2d 40 40: Number of times the backend sent n bytes

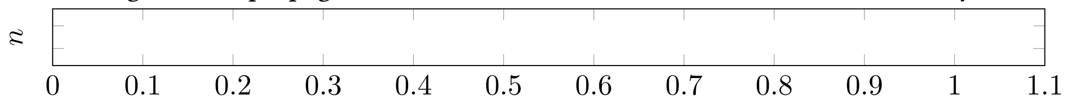


Figure 9.47: propagation2d 40 40: Number of times the backend received n bytes

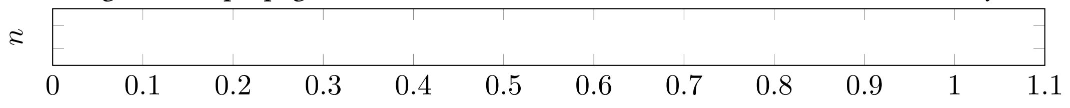
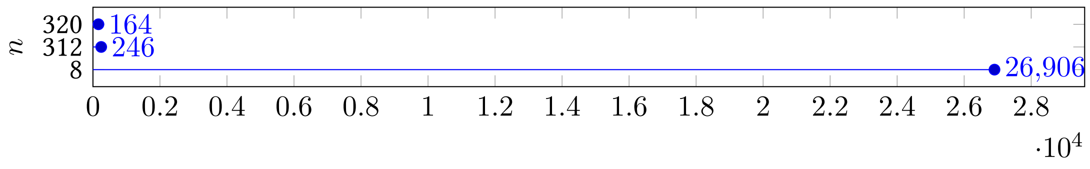


Figure 9.48: propagation2d 40 40: Number of times the backend reduced n bytes



9.2.9 spmv 5000

Running the test case with the command line `spmv 5000` using 10 processes caused the following things to happen in the backend, summarized across all processes:

- The backend was called a total of 20 times to send, receive, and/or reduce data, see figure 9.49, 9.50, and 9.51 for details about the kind of work submitted to the backend.
- The backend was called a total of 0 times because LAIK's view of the process group had changed, i.e. because one or more process were removed.
- The backend sent 9 chunks of data, see figure 9.52 for details about their sizes.
- The backend received 9 chunks of data, see figure 9.53 for details about their sizes.
- The backend reduced 10 chunks of data, see figure 9.54 for details about their sizes.

Figure 9.49: spmv 5000: Number of times the backend was called with n send operations

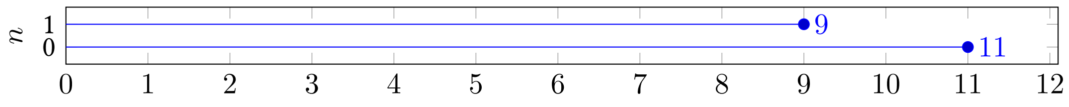


Figure 9.50: spmv 5000: Number of times the backend was called with n receive operations

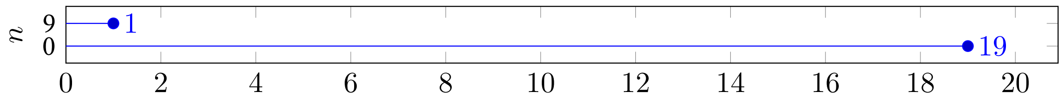


Figure 9.51: spmv 5000: Number of times the backend was called with n reduce operations

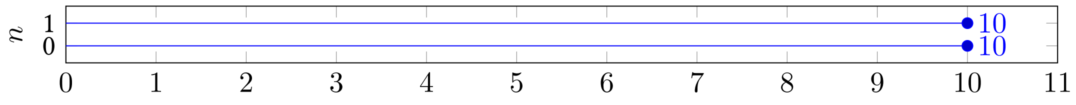


Figure 9.52: spmv 5000: Number of times the backend sent n bytes

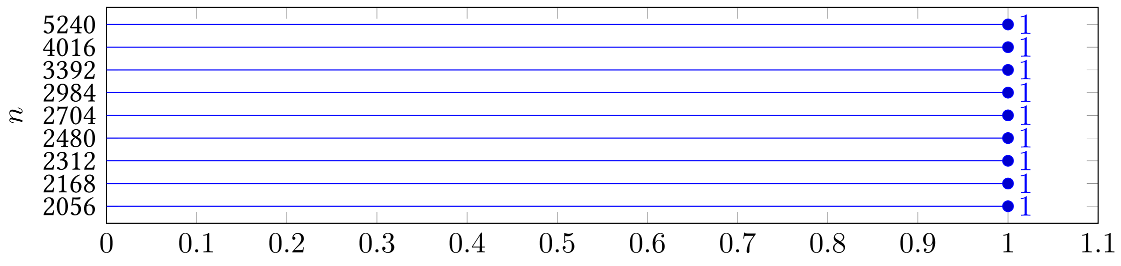


Figure 9.53: spmv 5000: Number of times the backend received n bytes

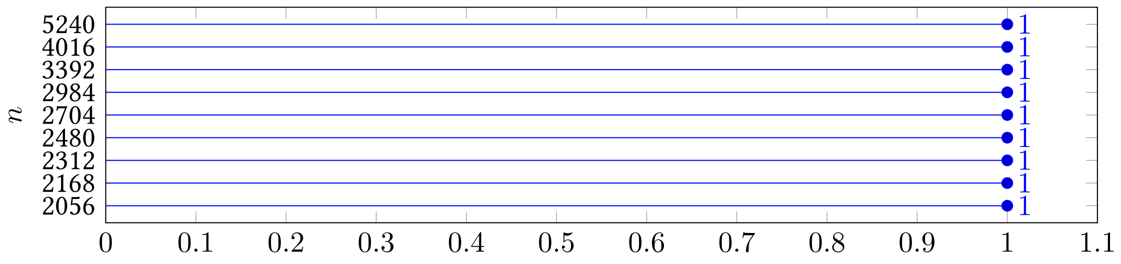
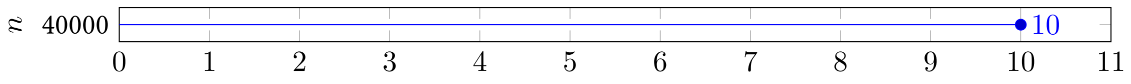


Figure 9.54: spmv 5000: Number of times the backend reduced n bytes



9.2.10 spmv2 150 1500

Running the test case with the command line `spmv2 150 1500` using 10 processes caused the following things to happen in the backend, summarized across all processes:

- The backend was called a total of 3020 times to send, receive, and/or reduce data, see figure 9.55, 9.56, and 9.57 for details about the kind of work submitted to the backend.
- The backend was called a total of 0 times because LAIK's view of the process group had changed, i.e. because one or more process were removed.
- The backend sent 13437 chunks of data, see figure 9.58 for details about their sizes.
- The backend received 13437 chunks of data, see figure 9.59 for details about their sizes.
- The backend reduced 1500 chunks of data, see figure 9.60 for details about their sizes.

Figure 9.55: spmv2 150 1500: Number of times the backend was called with n send operations

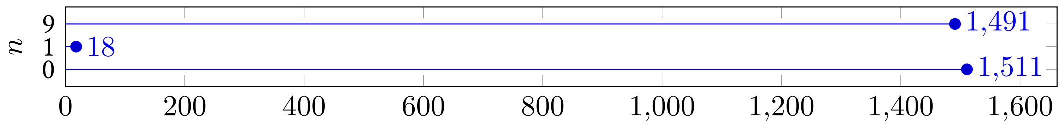


Figure 9.56: spmv2 150 1500: Number of times the backend was called with n receive operations

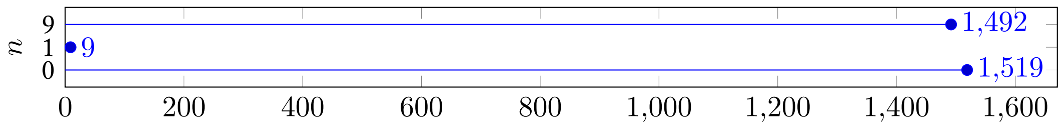


Figure 9.57: spmv2 150 1500: Number of times the backend was called with n reduce operations

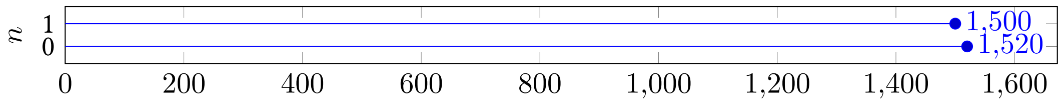


Figure 9.58: spmv2 150 1500: Number of times the backend sent n bytes

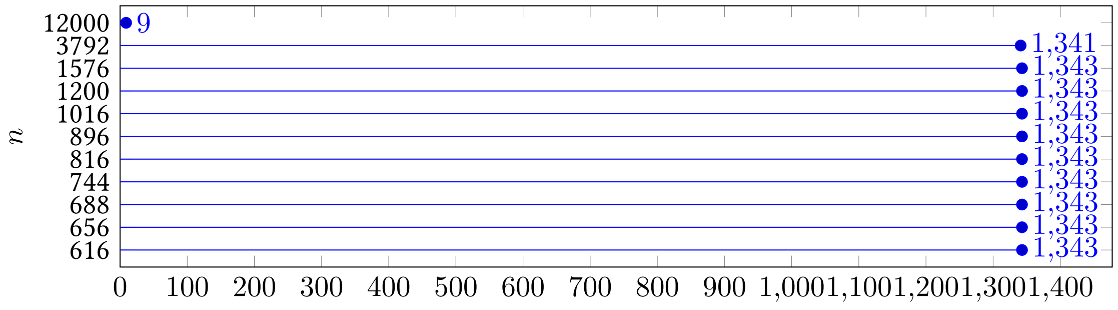


Figure 9.59: spmv2 150 1500: Number of times the backend received n bytes

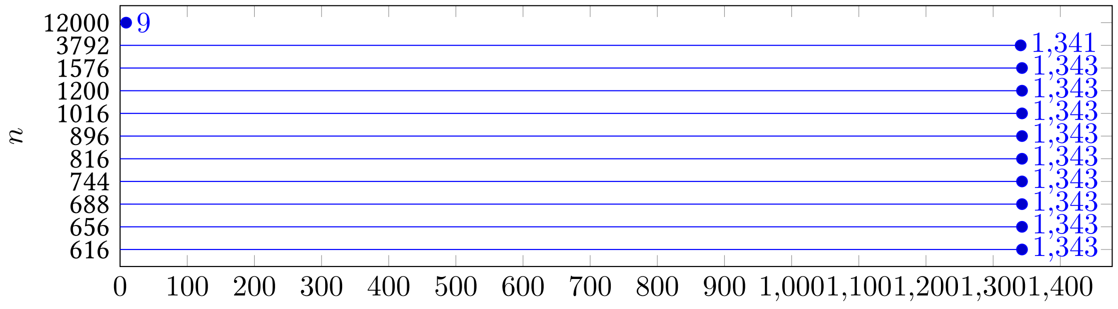
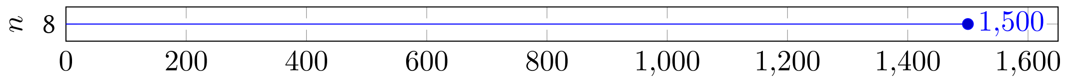


Figure 9.60: spmv2 150 1500: Number of times the backend reduced n bytes



9.2.11 vsum 20000000

Running the test case with the command line `vsum 20000000` using 10 processes caused the following things to happen in the backend, summarized across all processes:

- The backend was called a total of 40 times to send, receive, and/or reduce data, see figure 9.61, 9.62, and 9.63 for details about the kind of work submitted to the backend.
- The backend was called a total of 10 times because LAIK's view of the process group had changed, i.e. because one or more process were removed.
- The backend sent 42 chunks of data, see figure 9.64 for details about their sizes.
- The backend received 42 chunks of data, see figure 9.65 for details about their sizes.
- The backend reduced 10 chunks of data, see figure 9.66 for details about their sizes.

Figure 9.61: vsum 20000000: Number of times the backend was called with n send operations

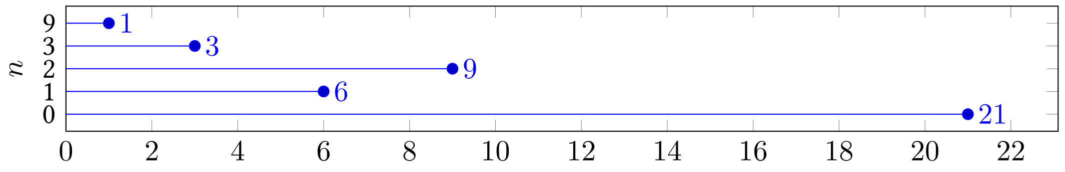


Figure 9.62: vsum 20000000: Number of times the backend was called with n receive operations

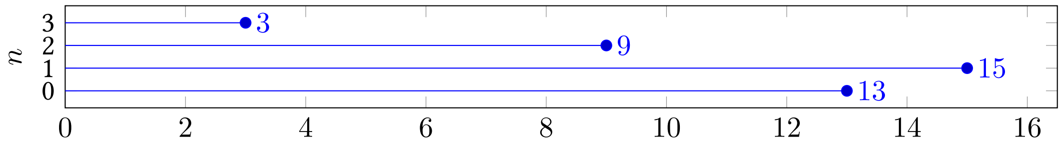


Figure 9.63: vsum 20000000: Number of times the backend was called with n reduce operations

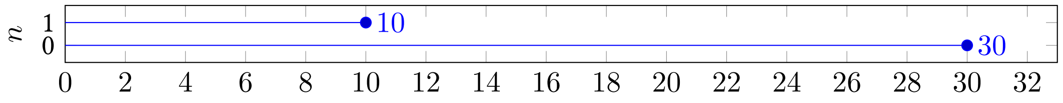


Figure 9.64: vsum 20000000: Number of times the backend sent n bytes

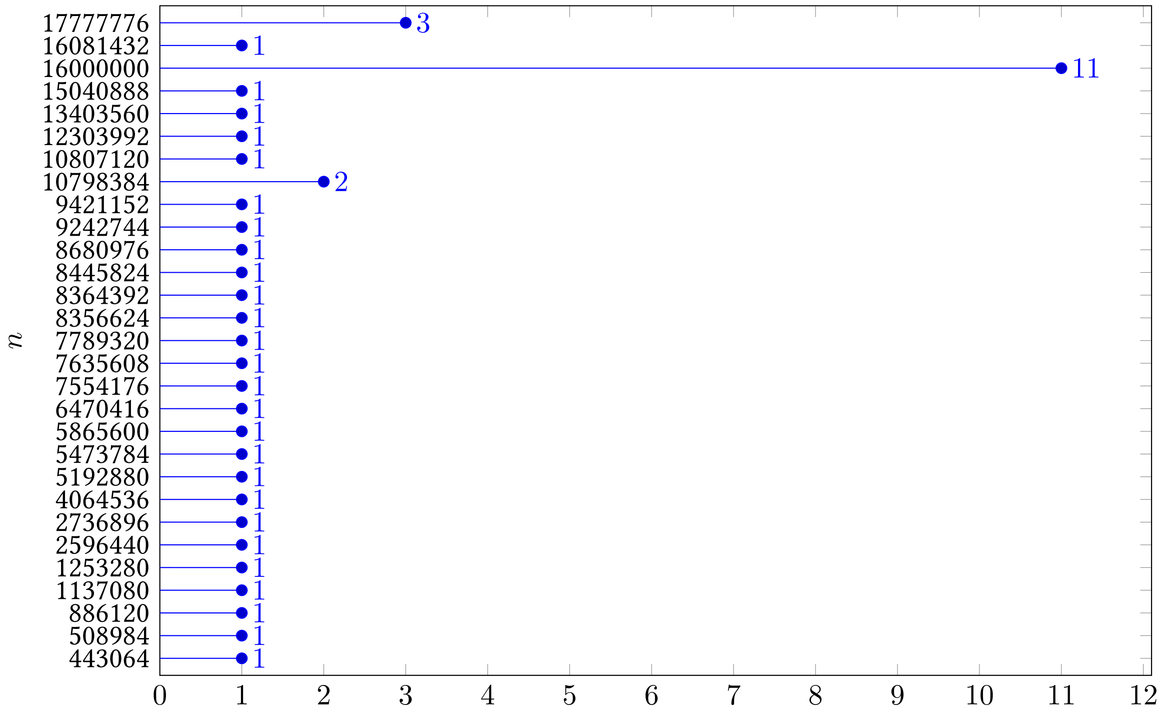


Figure 9.65: vsum 20000000: Number of times the backend received n bytes

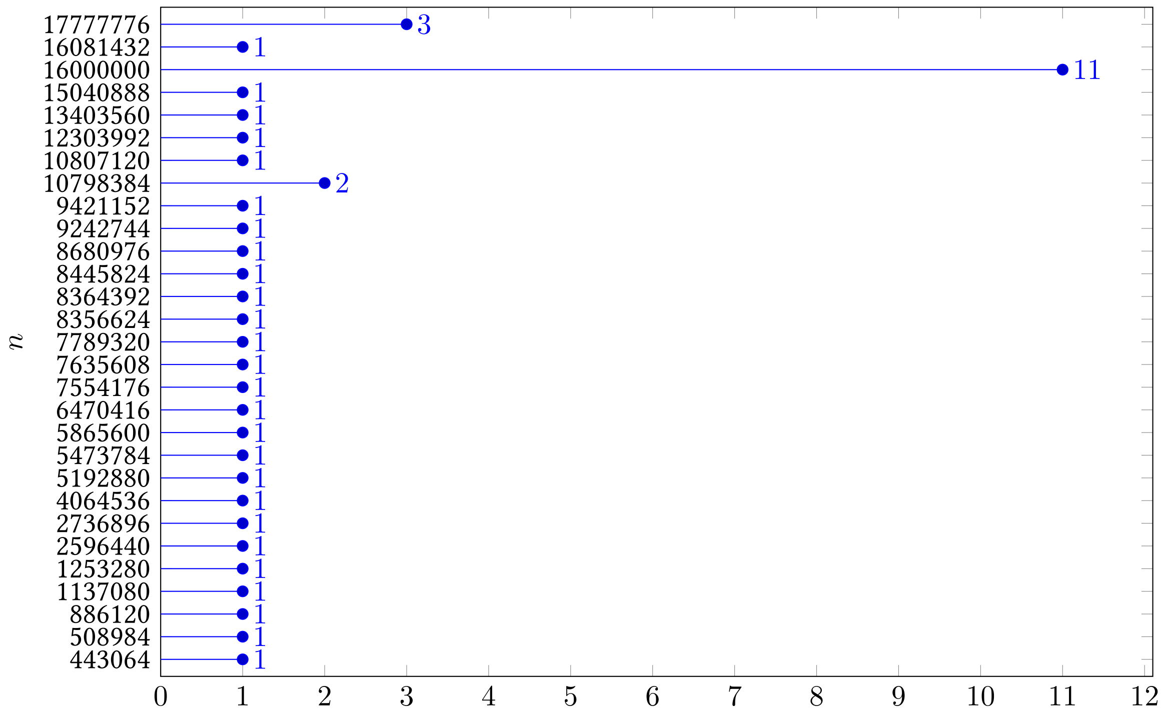
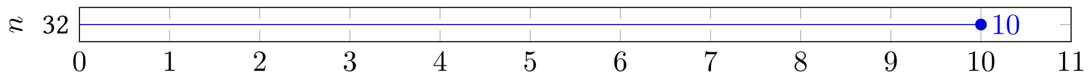


Figure 9.66: vsum 20000000: Number of times the backend reduced n bytes



9.2.12 vsum2

Running the test case with the command line `vsum2` using 10 processes caused the following things to happen in the backend, summarized across all processes:

- The backend was called a total of 40 times to send, receive, and/or reduce data, see figure 9.67, 9.68, and 9.69 for details about the kind of work submitted to the backend.
- The backend was called a total of 0 times because LAIK's view of the process group had changed, i.e. because one or more process were removed.
- The backend sent 88 chunks of data, see figure 9.70 for details about their sizes.
- The backend received 88 chunks of data, see figure 9.71 for details about their sizes.
- The backend reduced 10 chunks of data, see figure 9.72 for details about their sizes.

Figure 9.67: vsum2: Number of times the backend was called with n send operations

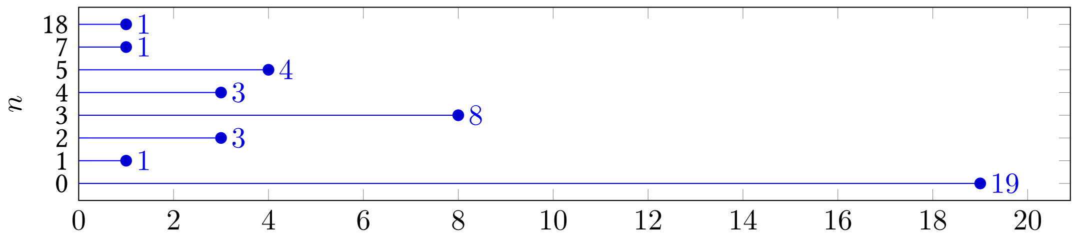


Figure 9.68: vsum2: Number of times the backend was called with n receive operations

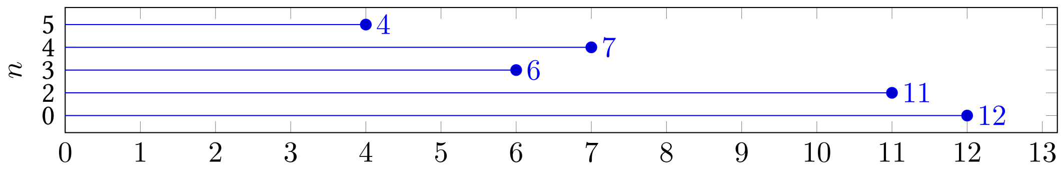


Figure 9.69: vsum2: Number of times the backend was called with n reduce operations

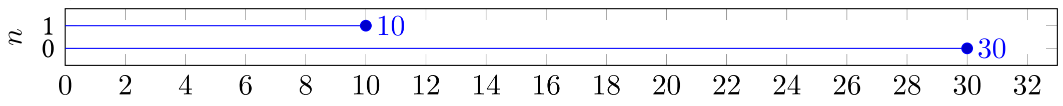


Figure 9.70: vsum2: Number of times the backend sent n bytes

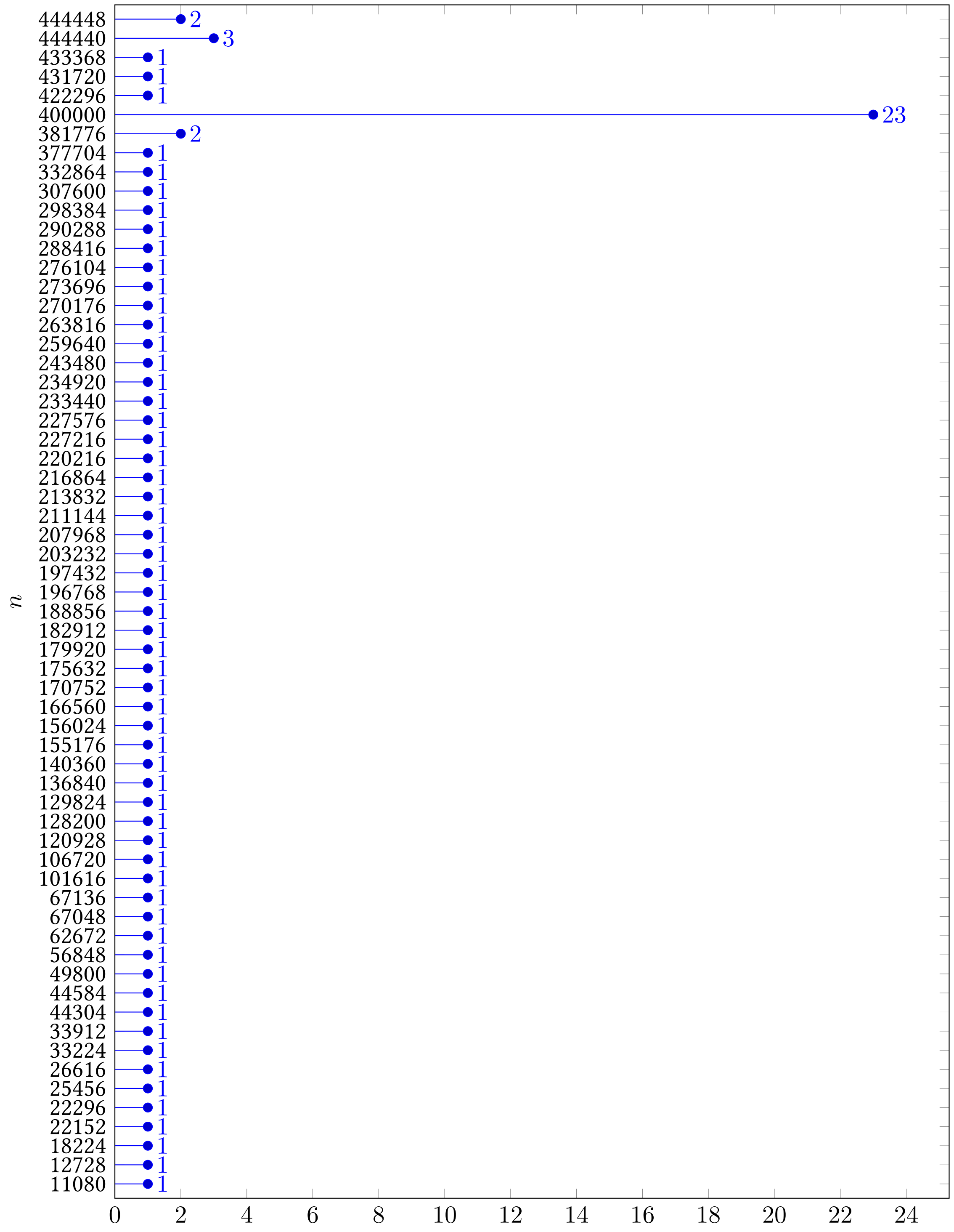


Figure 9.71: vsum2: Number of times the backend received n bytes

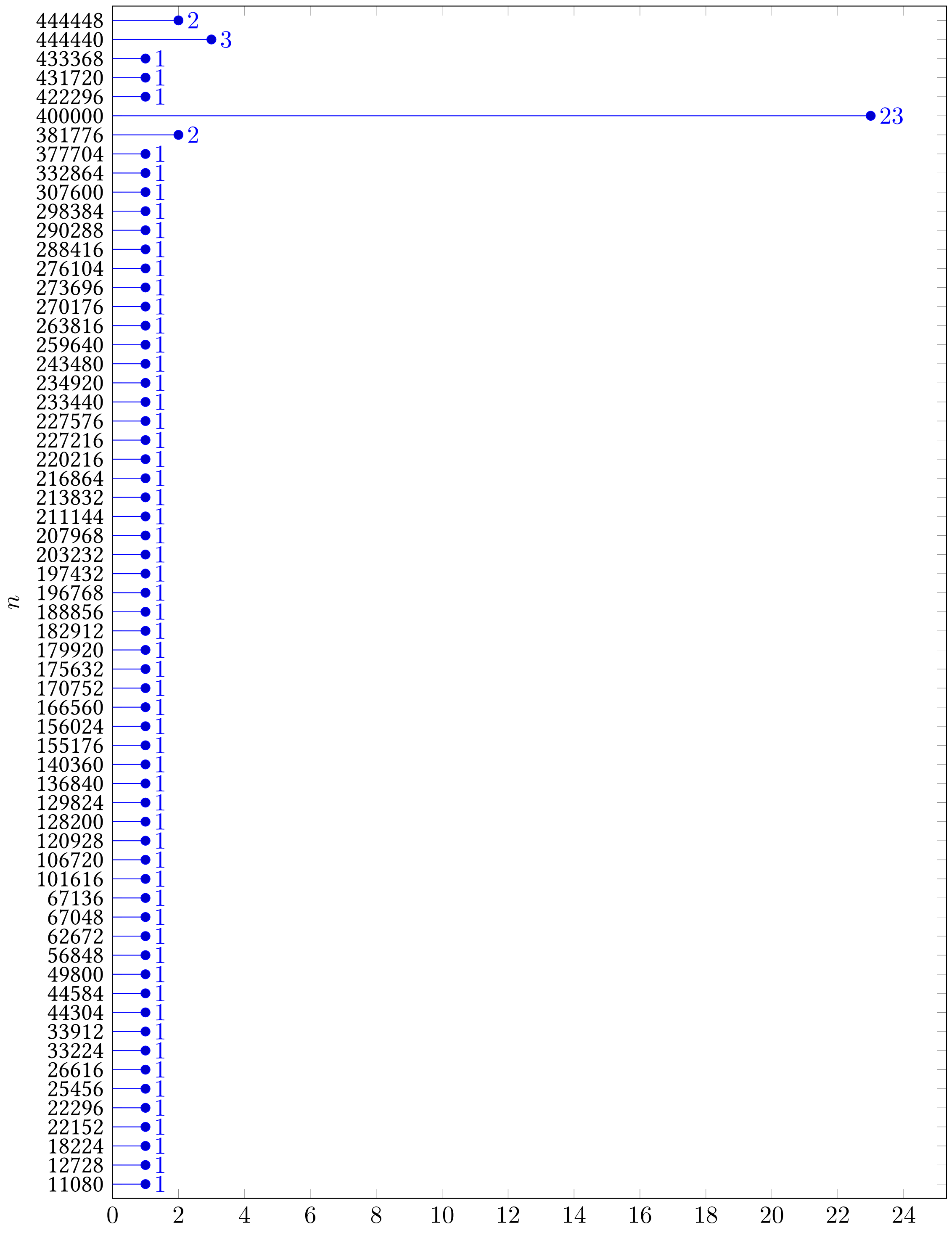
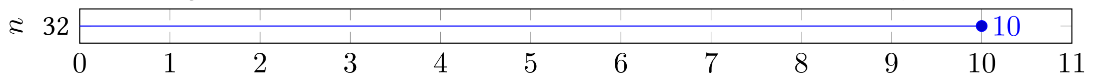


Figure 9.72: vsum2: Number of times the backend reduced n bytes



9.2.13 vsum3

Running the test case with the command line `vsum3` using 10 processes caused the following things to happen in the backend, summarized across all processes:

- The backend was called a total of 40 times to send, receive, and/or reduce data, see figure 9.73, 9.74, and 9.75 for details about the kind of work submitted to the backend.
- The backend was called a total of 10 times because LAIK's view of the process group had changed, i.e. because one or more process were removed.
- The backend sent 42 chunks of data, see figure 9.76 for details about their sizes.
- The backend received 42 chunks of data, see figure 9.77 for details about their sizes.
- The backend reduced 10 chunks of data, see figure 9.78 for details about their sizes.

Figure 9.73: vsum3: Number of times the backend was called with n send operations

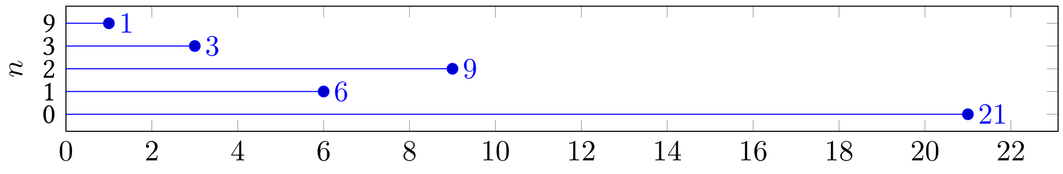


Figure 9.74: vsum3: Number of times the backend was called with n receive operations

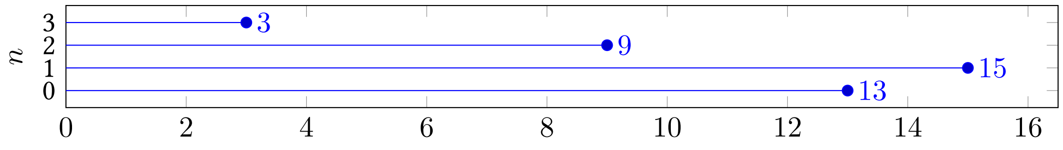


Figure 9.75: vsum3: Number of times the backend was called with n reduce operations

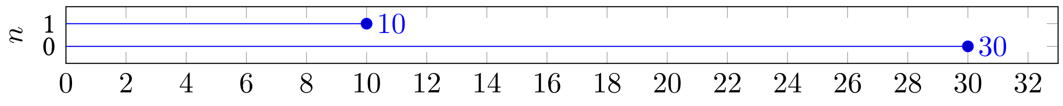


Figure 9.76: vsum3: Number of times the backend sent n bytes

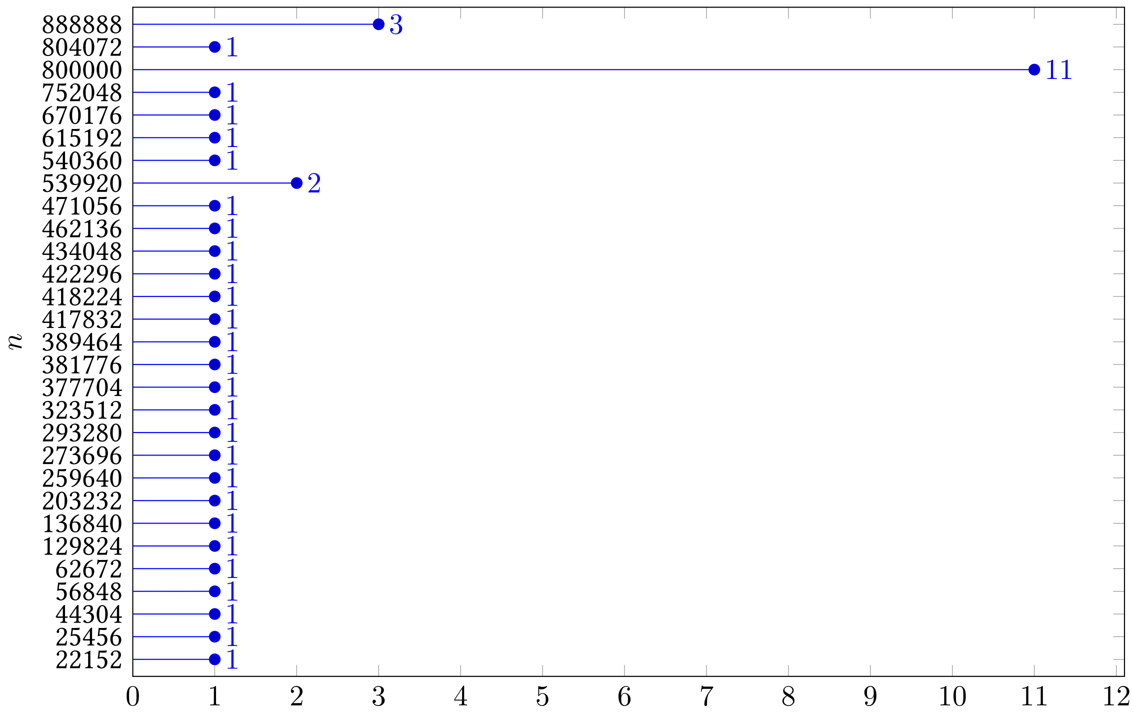


Figure 9.77: vsum3: Number of times the backend received n bytes

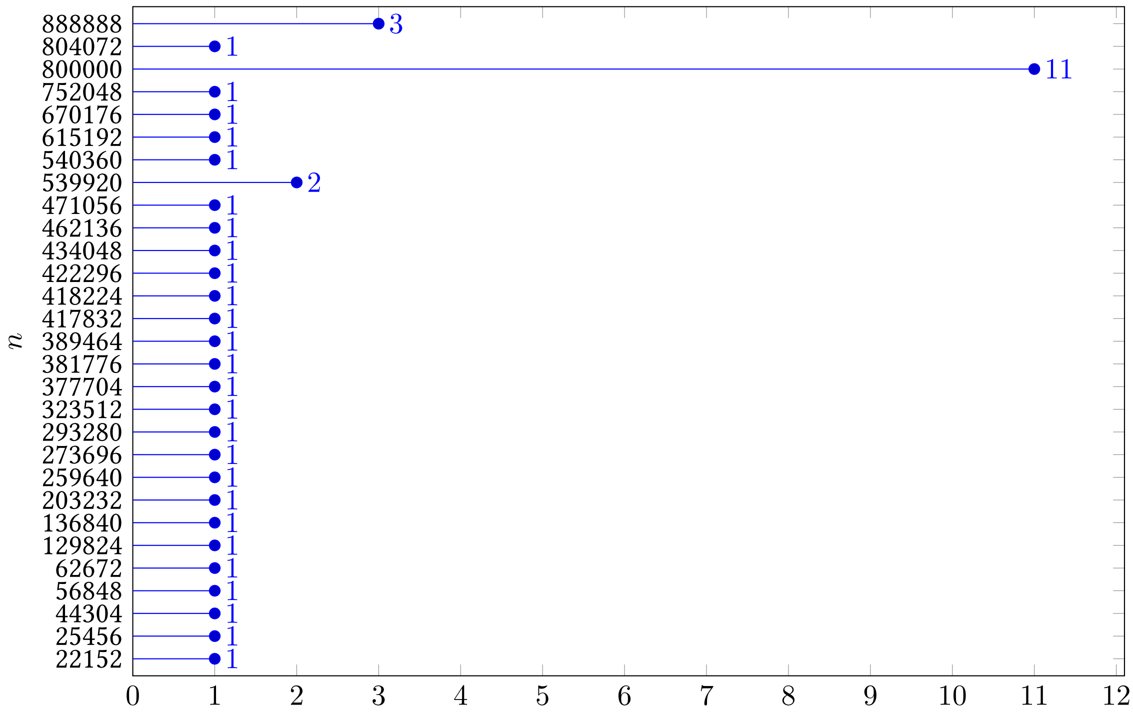
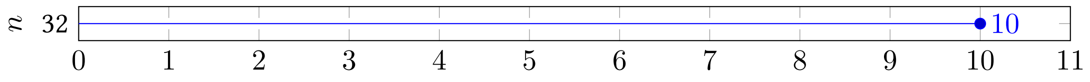


Figure 9.78: vsum3: Number of times the backend reduced n bytes



9.3 Example Configuration File for the TCP Backend

The following is the example configuration file for the TCP Backend. All commented-out values represent the default values, along with a short description.

```
[general]

# Whether the backend should do the sends in parallel to the receive operations
# backend_async_send = true

# Whether to attempt to use native MPI reductions if possible
# backend_native_reduce = false

# Whether reductions should be done on all output tasks instead of just one
# backend_peer_reduce = true

# How many connections to keep open concurrently
# client_connections = 16

# How many threads the client can use to send/receive messages
# client_threads = 4

# How many connections to keep open concurrently
# server_connections = 16

# How many threads the server can use to send/receive messages
# server_threads = 4

# How many connections the kernel is allowed to buffer for us
# socket_backlog = 64

# How long a socket may be unavailable for I/O before it is considered broken
# socket_timeout = 10.0

# How big our inbox may grow in bytes
# inbox_size = 16777216

# How big our outbox may grow in bytes
# outbox_size = 16777216

# How often a synchronous message send should be attempted giving up
# send_attempts = 100

# How long to wait after a failed synchronous message send before retrying
# send_delay = 0.1

# How often a message receive should be attempted before giving up
# receive_attempts = 100

# How long to wait for a message receive before sending an active request
# receive_timeout = 0.0

# How long to wait after the first active request before sending another
# receive_delay = 0.1

# Whether to to use asynchronous sends in the MPI_Comm_split operation
# minimpi_async_split = true;

[addresses]
# Where task 0 shall be located (TCP socket)
# 0 = localhost 4444

# Where task 1 shall be located (abstract UNIX socket)
# 1 = foo
```

9.4 Files Used in the Test Case Analysis

These files were used during the test case analysis in section 9.2.

9.4.1 analysis/analysis.diff

```
From 70563a8a5fb7d348cd842922cd25b7115ec8ef07 Mon Sep 17 00:00:00 2001
From: Alexander Kurtz <alexander@kurtz.be>
Date: Mon, 30 Apr 2018 14:16:48 +0200
Subject: [PATCH] src/backends/tcp: Use the statistics module for application
 profiling.

---
 src/CMakeLists.txt         |  2 +-
 src/backends/tcp/backend.c | 17 +++++++++++++++++
 2 files changed, 18 insertions(+), 1 deletion(-)

diff --git a/src/CMakeLists.txt b/src/CMakeLists.txt
index 76c919f..23cc726 100644
--- a/src/CMakeLists.txt
+++ b/src/CMakeLists.txt
@@ -120,7 +120,7 @@ if (tcp-backend)
             PRIVATE "GLIB_VERSION_MIN_REQUIRED=GLIB_VERSION_2_44"
             PRIVATE "USE_TCP"
             # PRIVATE "LAIK_TCP_DEBUG"
-            # PRIVATE "LAIK_TCP_STATS"
+            PRIVATE "LAIK_TCP_STATS"
         )

         target_link_libraries ("laik"
diff --git a/src/backends/tcp/backend.c b/src/backends/tcp/backend.c
index bf5c8ca..a7ff16b 100644
--- a/src/backends/tcp/backend.c
+++ b/src/backends/tcp/backend.c
@@ -27,6 +27,7 @@
 #include "debug.h"          // for laik_tcp_always
 #include "errors.h"         // for laik_tcp_errors_push, laik_tcp_errors_pre...
 #include "mpi.h"            // for MPI_Comm, MPI_Datatype, MPI_COMM_WORLD
+#include "stats.h"

 /* Internal structs */

@@ -119,6 +120,9 @@ static void laik_tcp_backend_receive
     g_autofree void*  buffer   = g_malloc (bytes);
     Laik_Index        start    = slice.from;

+    laik_tcp_stats_count ("laik_tcp_backend_receive_total");
+    laik_tcp_stats_count ("laik_tcp_backend_receive@%zu", bytes);
+
     // Make sure we aren't talking to ourselves
     laik_tcp_always (group->myid != sender);

@@ -195,6 +199,9 @@ static void laik_tcp_backend_send
     g_autofree void*  buffer   = g_malloc (bytes);
     Laik_Index        start    = slice.from;

+    laik_tcp_stats_count ("laik_tcp_backend_send_total");
+    laik_tcp_stats_count ("laik_tcp_backend_send@%zu", bytes);
+
     // Make sure we aren't talking to ourselves
     laik_tcp_always (group->myid != receiver);

@@ -348,6 +355,9 @@ static void laik_tcp_backend_reduce
     const size_t        elements     = op->slc.to.i[0] - op->slc.from.i[0];
     const size_t        bytes        = elements * data->elemsize;

+    laik_tcp_stats_count ("laik_tcp_backend_reduce_total");
+    laik_tcp_stats_count ("laik_tcp_backend_reduce@%zu", bytes);
+
     laik_tcp_always (group->myid >= 0);

     char* input_buffer = NULL;
@@ -600,6 +610,11 @@ static void laik_tcp_backend_exec

     const Laik_Group* group = data->activePartitioning->group;

+    laik_tcp_stats_count ("laik_tcp_backend_exec_total");
+    laik_tcp_stats_count ("laik_tcp_backend_exec_reductions@%d", transition->redCount)
    ;
+    laik_tcp_stats_count ("laik_tcp_backend_exec_receives@%d", transition->recvCount);
+    laik_tcp_stats_count ("laik_tcp_backend_exec_sends@%d", transition->sendCount);
+
     // Handle the reduce operations
     for (int i = 0; i < transition->redCount; i++) {
         const struct redTOp* op = transition->red + i;
@@ -729,6 +744,8 @@ static void laik_tcp_backend_finalize () {
 static void laik_tcp_backend_update_group (Laik_Group* group) {
     laik_tcp_always (group);

+    laik_tcp_stats_count ("laik_tcp_backend_update_group_total");
+
     g_autoptr (Laik_Tcp_Errors) errors = laik_tcp_errors_new ();

     // We are transitioning from an old (parent) group to a new (child) group,
--
2.17.0
```

9.4.2 analysis/config.txt

```
[general]
# nothing here

[addresses]
0 = localhost 2000
1 = localhost 2001
2 = localhost 2002
3 = localhost 2003
4 = localhost 2004
5 = localhost 2005
6 = localhost 2006
7 = localhost 2007
8 = localhost 2008
9 = localhost 2009
```

9.4.3 analysis/analyze.sh

```
#!/bin/bash -eu

run(){
    local index=''

    for index in  seq 0 9 ; do
        LAIK_BACKEND='tcp' LAIK_TCP_CONFIG='config.txt' "${@}" &
    done

    wait
}

gather(){
    local index=''

    for index in  seq 0 9 ; do
        cat -- "laik-tcp-stats-for-rank-${index}.txt"
    done | sort --field-separator='@' --numeric-sort --key='2'
}

summarize(){
    local oldkey=''
    local newkey=''
    local sum='0'

    while read newkey value; do
        # All values are integers, so drop the fractional part
        value="${value%%.*}"

        if [ "${oldkey}" = "${newkey}" ]; then
            sum="$((sum␣+␣value))"
        else
            if [ "${oldkey}" ]; then
                echo "${oldkey}␣${sum}"
            fi

            oldkey="${newkey}"
            sum="${value}"
        fi
    done

    echo "${oldkey}␣${sum}"
}

store(){
    local key=''
    local value=''
    local prefix=''
    local suffix=''
    local path=''

    while read key value; do
        case "${key}" in
            *@*)
                prefix="${key%@*}"
                suffix="${key##*@}"
                echo "${suffix}␣${value}" >> "${1}/${prefix}"
                ;;
            *)
                echo "${value}" > "${1}/${key}"
                ;;
        esac
    done
}

job(){
    run "./${@}"

    mkdir --parents -- "results/${*}"

    echo '0'               > "results/${*}/laik_tcp_backend_exec_total"
    echo '0'               > "results/${*}/laik_tcp_backend_send_total"
    echo '0'               > "results/${*}/laik_tcp_backend_receive_total"
    echo '0'               > "results/${*}/laik_tcp_backend_reduce_total"
    echo '0'               > "results/${*}/laik_tcp_backend_update_group_total"

    echo 'count frequency' > "results/${*}/laik_tcp_backend_exec_receives"
    echo 'count frequency' > "results/${*}/laik_tcp_backend_exec_reductions"
    echo 'count frequency' > "results/${*}/laik_tcp_backend_exec_sends"

    echo 'count frequency' > "results/${*}/laik_tcp_backend_receive"
    echo 'count frequency' > "results/${*}/laik_tcp_backend_reduce"
    echo 'count frequency' > "results/${*}/laik_tcp_backend_send"

    gather | summarize | store "results/${*}"
}

# Run the jobs
job jac1d 125000
job jac2d 11000
job jac3d 375
job jac3d -g 375 # Requested by Weidendorfer
job markov2 400 4000
job markov2 -f 400 4000 # Requested by Weidendorfer
job markov 400 4000
```

```
# job propagation1d # Throws assertion failures
job propagation2d 40 40
job spmv 5000 # 10000 is the biggest allowed, but throws OOM errors
job spmv2 150 1500
job vsum2 # Takes no parameters
job vsum3 # Takes no parameters
job vsum 20000000
```

9.5 Files Used in the Evaluation

These files were used during the evaluation in chapter 6.

9.5.1 evaluation/environments/tcp.sh

```
# Set the environment variables
export LAIK_BACKEND='tcp'
export LAIK_TCP_CONFIG="${root}/config.txt"

# Write the configuration file
cat > "${LAIK_TCP_CONFIG}" <<EOF
[general]
receive_attempts=1000

[addresses]
 cat -- "${tmp}/addresses"
EOF

# Make sure the configuration file is on disk to avoid NFS delays
sync
```

9.5.2 evaluation/environments/mpi.sh

```
# Set the environment variable
export LAIK_BACKEND='mpi'
```

9.5.3 evaluation/environments/tcp-master-reduction.sh

```
# Set the environment variables
export LAIK_BACKEND='tcp'
export LAIK_TCP_CONFIG="${root}/config.txt"

# Write the configuration file
cat > "${LAIK_TCP_CONFIG}" <<EOF
[general]
receive_attempts=1000
backend_peer_reduce=false

[addresses]
 cat -- "${tmp}/addresses"
EOF

# Make sure the configuration file is on disk to avoid NFS delays
sync
```

9.5.4 evaluation/environments/tcp-resources.sh

```
# Set the environment variables
export LAIK_BACKEND='tcp'
export LAIK_TCP_CONFIG="${root}/config.txt"

# Write the configuration file
cat > "${LAIK_TCP_CONFIG}" <<EOF
[general]
receive_attempts=1000
client_connections=400
server_connections=400
socket_backlog=400
outbox_size=268435456
inbox_size=268435456

[addresses]
 cat -- "${tmp}/addresses"
EOF
```

```

# Make sure the configuration file is on disk to avoid NFS delays
sync
```

9.5.5 evaluation/sample.sh

```
#!/bin/bash -eu

# Usage: calc <arithmetic expression>
# Passes its first argument to GNU bc and returns the result
calc(){
bc --mathlib <<EOF
${1}
EOF
}

# Usage cleanup <exit code>
# Sends SIGTERM to all children, waits for them and then exits
cleanup(){
    # Stop trapping EXIT so calling exit doesn't invoke us another time
    trap '' EXIT

    # Remove the temporary directory
    rm --force --recursive -- "${tmp}"

    # Send SIGTERM to all our children
    pkill --parent="$$" || true

    # Wait for all children to terminate
    wait

    # Exit with the requested exit code
    exit "${1}"
}

# Usage die <message>
# Prints a message to stderr and exits
die(){
    printf '%s\n' "${1}" >&2
    exit 1
}

# Set defaults for the test parameters
ENVIRONMENT="${ENVIRONMENT-tcp}"
RESULT="${RESULT-/dev/stdout}"
SAMPLES="${SAMPLES-5}"

# Make sure we are running under salloc
if ! [ "${SLURM_NTASKS+x}" ]; then
    die "SLURM_NTASKS␣not␣defined,␣did␣you␣run␣this␣script␣under␣salloc␣with␣--ntasks-per-node?"
fi

# Determine the directory this script is in
root="`dirname␣--␣"${0}"`"

# Get a temporary directory
tmp="`mktemp␣--directory`"

# Make sure we are never leaving any child processes around and clean up our tmp
trap 'cleanup $?'  EXIT
trap 'cleanup 129' HUP
trap 'cleanup 130' INT
trap 'cleanup 143' TERM

# Generate the address list
srun sh -c 'echo "${SLURM_NODEID}-${SLURM_LOCALID}=`hostname␣--ip-address`␣$((2000␣+␣SLURM_LOCALID))"' > "${tmp}/addresses"

# Run the environment-specific setup step
. "${root}/environments/${ENVIRONMENT}.sh"
```

```

# Take ${SAMPLES} measurements
times=""
for sample in  seq "${SAMPLES}" ; do
    # Start the synchronization servers
    "${root}/syncd.py" '10.42.1.253' '3000' "${SLURM_NTASKS}" > "${tmp}/start" & start_pid="${!}"
    "${root}/syncd.py" '10.42.1.253' '4000' "${SLURM_NTASKS}" > "${tmp}/stop"  & stop_pid="${!}"

    # Run the main job
    srun --kill-on-bad-exit -- " realpath␣--␣"${root}/run.sh" " "${@}"

    # Wait for the synchronization servers
    wait "${start_pid}"
    wait "${stop_pid}"

    # Add the new time to the list
    start=" cat␣--␣"${tmp}/start" "
    stop=" cat␣--␣"${tmp}/stop" "
    time=" calc␣"${stop} - ${start}" "
    times="${times}␣${time}"

    # Report the progress
    printf "[%s]␣Sample␣%02d/%02d␣is␣%f␣seconds\n" "${0##*/}" "${sample}" "${SAMPLES}" "${time}"
done

# Calculate the average
average='0'
for time in ${times}; do
    average=" calc␣"${average} + ${time}" "
done
average=" calc␣"${average} / ${SAMPLES}" "

# Calculate the standard deviation
deviation='0'
for time in ${times}; do
    deviation=" calc␣"${deviation} + (${time} - ${average})^2" "
done
deviation=" calc␣"sqrt (${deviation} / (${SAMPLES} - 1))" "

# Store the result
printf '%f %f\n' "${average}" "${deviation}" > "${RESULT}"
```

9.5.6 evaluation/syncd.py

```
#!/usr/bin/env python3

import socket
import sys
import time

# Get the parameters
host  = sys.argv[1]
port  = int (sys.argv[2])
count = int (sys.argv[3])

# Create a server socket
server = socket.socket (socket.AF_INET, socket.SOCK_STREAM)
server.setsockopt (socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.bind ((host, port))
server.listen (512)

# Accept the requested number of connections and store them in a list so they
# don't get garbage-collected and thus closed prematurely
connections = []
for index in range (count):
    connection, address = server.accept ()
    connections.append (connection)

```

```
# Print the monotonic time
print (time.monotonic ())

# Exit, thus closing all the connections
```

9.5.7 evaluation/run.sh

```
#!/bin/bash -eu

# If LAIK_TCP_CONFIG is set, put it in the cache
if [ "${LAIK_TCP_CONFIG+x}" ]; then
    cat -- "${LAIK_TCP_CONFIG}" > /dev/null
fi

# Put the binary and all its dependencies in the cache
LD_TRACE_LOADED_OBJECTS='1' "${1}" > /dev/null

# Wait until the start server has accepted and closed our connection
while ! cat < '/dev/tcp/10.42.1.253/3000' > '/dev/null'; do true; done

# Run the task
LD_LIBRARY_PATH="${HOME}/lib" "${@}" 0<'/dev/null' 1>'/dev/null'

# Wait until the stop server has accepted and closed our connection
while ! cat < '/dev/tcp/10.42.1.253/4000' > '/dev/null'; do true; done
```

9.5.8 evaluation/filter.sh

```
#!/bin/bash -eu

# Determine the directory this script is in
root=" dirname␣--␣"${0}" "

for result in "${root}"/results/*/*/*/*/*; do
    read avg dev < "${result}"

    if [ " echo␣"${dev} > 1.0"␣|␣bc␣-l " = '1' ] && [ " echo␣"${dev} > 0.10 * ${avg}"␣|␣bc␣-l " = '1' ]; then
        rm --verbose -- "${result}"
    fi
done
```

9.5.9 evaluation/main.sh

```
#!/bin/bash -eu

job(){
    for partition in 'odr'; do
        for processes in 1 2 3 4; do
            for ENVIRONMENT in 'mpi' 'tcp' 'tcp-master-reduction' 'tcp-resources' 'tcp-serial'; do
                local prefix="${root}/results/${partition}/${processes}/${ENVIRONMENT}/${*}"
                mkdir --parents -- "${prefix}"

                local series="${prefix}.txt"
                echo "nodes_time_error" > "${series}"

                for nodes in  seq 5 35 ; do
                    RESULT="${prefix}/${nodes}.txt"

                    if [ -e "${RESULT}" ]; then
                        printf '[%s] Reusing %s\n' "${0##*/}" "${RESULT}"
                    else
                        printf '[%s] Creating %s\n' "${0##*/}" "${RESULT}"
                        salloc \
                            --partition="${partition}" \
                            --nodes="${nodes}" \
                            --ntasks-per-node="${processes}" \
                            --time='120' \
                            --kill-command='TERM' \
                            "${root}/sample.sh" \
                            "${HOME}/laik/examples/${@}"
                    fi

                    echo "${nodes}␣ cat␣--␣"${RESULT}" " >> "${series}"
                done
            done
        done
    done
}

# Determine the directory this script is in
root=" dirname␣--␣"${0}" "

# Export the parameters
export ENVIRONMENT='invalid'
export RESULT='invalid'
export SAMPLES='10'

# Run the jobs
job jac1d 125000
job jac2d 11000
job jac3d 375
job jac3d -g 375 # Requested by Weidendorfer
job markov2 400 4000
job markov2 -f 400 4000 # Requested by Weidendorfer
job markov 400 4000
# job propagation1d # Throws assertion failures
job propagation2d 40 40
job spmv 5000 # 10000 is the biggest allowed, but throws OOM errors
job spmv2 150 1500
job vsum2 # Takes no parameters
job vsum3 # Takes no parameters
job vsum 20000000
```

10 References

[1] Greg Brockman. 2013. Speed Up SSH by Reusing Connections. Retrieved May 12, 2018 from https://puppet.com/blog/speed-up-ssh-by-reusing-connections

[2] Debian Project. 2016. CrossToolchains. Retrieved May 12, 2018 from https://wiki.debian.org/CrossToolchains

[3] Alexis Engelke. 2018. HimMUC Cluster. Retrieved May 12, 2018 from https://www.lrr.in.tum.de/en/projekte/forschungsprojekte/himmuc/

[4] R. Fielding and J. Reschke. 2014. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. Internet Requests for Comments; RFC Editor. Retrieved May 12, 2018 from https://www.rfc-editor.org/rfc/rfc7230.txt

[5] Message Passing Interface Forum. 1994. *MPI: A Message-Passing Interface Standard - Version 1.0*. Retrieved May 12, 2018 from https://www.mpi-forum.org/docs/mpi-1.0/mpi-10.ps.Z

[6] Message Passing Interface Forum. 2015. *MPI: A Message-Passing Interface Standard - Version 3.1*. Retrieved May 12, 2018 from https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[7] GNOME Project. 2018. Key-value file parser: GLib Reference Manual. Retrieved May 12, 2018 from https://developer.gnome.org/glib/stable/glib-Key-value-file-parser.html

[8] William Gropp and Ewing Lusk. 2004. Fault Tolerance in Message Passing Interface Programs. *International Journal of High Performance Computing Applications* 18, 3 (August 2004), 363–372. Retrieved May 12, 2018 from https://www.mcs.anl.gov/~lusk/papers/fault-tolerance.pdf

[9] Torsten Hoefler, Mirko Reinhardt, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. 2006. Low Overhead Ethernet Communication for Open MPI on Linux Clusters. CSR-06, 06 (July 2006). Retrieved May 12, 2018 from https://htor.inf.ethz.ch/publications/index.php?pub=31

[10] Joshua Hursey, Jeffrey M. Squyres, Timothy Mattox, and Andrew Lumsdaine. 2007. The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. *2007 IEEE International Parallel and Distributed Processing Symposium* (2007), 1–8. Retrieved May 12, 2018 from https://ieeexplore.ieee.org/abstract/document/4228333/

[11] Joshua Hursey, Jeffrey Squyres, and Andrew Lumsdaine. 2006. A checkpoint and restart service specification for Open MPI. (July 2006). Retrieved May 12, 2018 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.64.7661&rep=rep1&type=pdf

[12] IEEE. 2017. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008): IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications*. IEEE. Retrieved May 12, 2018 from https://standards.ieee.org/findstds/standard/1003.1-2017.html

[13] Intel. 2018. Fault Tolerance Support. Retrieved May 12, 2018 from https://software.intel.com/en-us/mpi-developer-reference-linux-fault-tolerance-support

[14] LAIK Project. 2018. LAIK Github Project Page. Retrieved May 12, 2018 from https://github.com/envelope-project/laik

[15] Linux man-pages Project. 2017. tcp - TCP protocol. Retrieved May 12, 2018 from http:

//man7.org/linux/man-pages/man7/tcp.7.html

[16] Linux man-pages Project. 2017. udp - User Datagram Protocol for IPv4. Retrieved May 12, 2018 from http://man7.org/linux/man-pages/man7/udp.7.html

[17] Linux man-pages Project. 2017. raw - Linux IPv4 raw sockets. Retrieved May 12, 2018 from http://man7.org/linux/man-pages/man7/raw7.7.html

[18] Linux man-pages Project. 2017. socket - Linux socket interface. Retrieved May 12, 2018 from http://man7.org/linux/man-pages/man7/socket.7.html

[19] Linux man-pages Project. 2017. socket - create an endpoint for communication. Retrieved May 12, 2018 from http://man7.org/linux/man-pages/man2/socket.2.html

[20] Linux man-pages Project. 2017. connect - initiate a connection on a socket. Retrieved May 12, 2018 from http://man7.org/linux/man-pages/man2/connect.2.html

[21] Linux man-pages Project. 2017. send, sendto, sendmsg - send a message on a socket. Retrieved May 12, 2018 from http://man7.org/linux/man-pages/man2/send.2.html

[22] Linux man-pages Project. 2017. bind - bind a name to a socket. Retrieved May 12, 2018 from http://man7.org/linux/man-pages/man2/bind.2.html

[23] Linux man-pages Project. 2017. listen - listen for connections on a socket. Retrieved May 12, 2018 from http://man7.org/linux/man-pages/man2/listen.2.html

[24] Linux man-pages Project. 2017. accept, accept4 - accept a connection on a socket. Retrieved May 12, 2018 from http://man7.org/linux/man-pages/man2/accept.2.html

[25] Linux man-pages Project. 2017. recv, recvfrom, recvmsg - receive a message from a socket. Retrieved May 12, 2018 from http://man7.org/linux/man-pages/man2/recv.2.html

[26] Linux man-pages Project. 2017. open, openat, creat - open and possibly create a file. Retrieved May 12, 2018 from http://man7.org/linux/man-pages/man2/open.2.html

[27] Linux man-pages Project. 2017. poll, ppoll - wait for some event on a file descriptor. Retrieved May 12, 2018 from http://man7.org/linux/man-pages/man2/poll.2.html

[28] SchedMD LLC. 2017. Slurm Workload Manager. Retrieved May 12, 2018 from https://slurm.schedmd.com

[29] MPICH Project. 2018. MPI_Comm_rank. Retrieved May 12, 2018 from https://www.mpich.org/static/docs/v3.1/www3/MPI_Comm_rank.html

[30] MPICH Project. 2018. MPICH | High-Performance Portable MPI. Retrieved May 12, 2018 from https://www.mpich.org

[31] MPICH Project. 2018. MPI_Init. Retrieved May 12, 2018 from https://www.mpich.org/static/docs/v3.1/www3/MPI_Init.html

[32] MPICH Project. 2018. MPI_Comm_size. Retrieved May 12, 2018 from https://www.mpich.org/static/docs/v3.1/www3/MPI_Comm_size.html

[33] MPICH Project. 2018. MPI_Send. Retrieved May 12, 2018 from https://www.mpich.org/static/docs/v3.1/www3/MPI_Send.html

[34] MPICH Project. 2018. MPI_Recv. Retrieved May 12, 2018 from https://www.mpich.org/static/docs/v3.1/www3/MPI_Recv.html

[35] MPICH Project. 2018. MPI_Finalize. Retrieved May 12, 2018 from https://www.mpich.org/

static/docs/v3.1/www3/MPI_Finalize.html

[36] MPICH Project. 2018. MPI_Comm_dup. Retrieved May 12, 2018 from https://www.mpich.org/static/docs/v3.1/www3/MPI_Comm_dup.html

[37] MPICH Project. 2018. MPI_Comm_split. Retrieved May 12, 2018 from https://www.mpich.org/static/docs/v3.1/www3/MPI_Comm_split.html

[38] MPICH Project. 2018. MPI_Reduce. Retrieved May 12, 2018 from https://www.mpich.org/static/docs/v3.1/www3/MPI_Reduce.html

[39] MPICH Project. 2018. MPI_Allreduce. Retrieved May 12, 2018 from https://www.mpich.org/static/docs/v3.1/www3/MPI_Allreduce.html

[40] MPICH Project. 2018. MPICH Overview. Retrieved May 12, 2018 from https://www.mpich.org/about/overview

[41] MPICH Project. 2018. MPI_Probe. Retrieved May 12, 2018 from https://www.mpich.org/static/docs/v3.1/www3/MPI_Probe.html

[42] MPICH Project. 2018. MPI_Get_processor_name. Retrieved May 12, 2018 from https://www.mpich.org/static/docs/v3.1/www3/MPI_Get_processor_name.html

[43] Open MPI Project. 2016. FAQ: Running jobs under SLURM. Retrieved May 12, 2018 from https://www.open-mpi.org/faq/?category=slurm

[44] Open MPI Project. 2018. Open MPI: Open Source High Performance Computing. Retrieved May 12, 2018 from https://www.open-mpi.org

[45] OpenMPI Project. 2016. FAQ: Fault tolerance for parallel MPI jobs. Retrieved May 12, 2018 from http://www.open-mpi.de/faq/?category=ft

[46] Postfix Project. 2018. Postfix Connection Cache. Retrieved May 12, 2018 from http://www.postfix.org/CONNECTION_CACHE_README.html

[47] Josef Weidendorfer, Dai Yang, and Carsten Trinitis. 2017. LAIK: A Library for Fault Tolerant Distribution of Global Data for Parallel Applications. Retrieved May 12, 2018 from https://mediatum.ub.tum.de/doc/1375185/1375185.pdf

[48] Dai Yang, Josef Weidendorfer, Carsten Trinitis, Tilman Küstner, and Sibylle Ziegler. 2017. Enabling Application-Integrated Proactive Fault Tolerance. In *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing, ParCo 2017, 12-15 September 2017, Bologna, Italy*, 475–484. Retrieved May 12, 2018 from https://www.lrr.in.tum.de/fileadmin/w00bph/www/Mitarbeiter/yang/laik_parco.pdf