SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY

INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Design and Implementation of an RDMA backend supporting malleability for LAIK

Alisa Parashchenko

SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY

INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Design and Implementation of an RDMA backend supporting malleability for LAIK

Entwurf und Implementierung eines RDMA-Backends mit Malleability für LAIK

Author:	Alisa Parashchenko
Supervisor:	Dr. rer. nat. Josef Weidendorfer
Examiner:	Dr. rer. nat. Josef Weidendorfer
Submission Date:	15th September 2024

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

Munich, 15th September 2024 Alisa Parashchenko

Abstract

Currently, the LAIK partitioning library has two main backends: MPI and TCP2. The MPI backend translates LAIK transitions into MPI function calls and has high performance, but does not support resizes. The TCP2 backend directly executes LAIK transitions using TCP sockets and is slow, but it supports resizes.

We describe the implementation of a new backend that is intended to offer both high performance and resize functionality, and evaluate its actual performance. Due to time constraints, only part of the resize functionality has been implemented: new processes can join the world after the computation has already started, but processes cannot be deleted from the world yet.

Our new backend is based on libfabric, a library offering an API on top of different providers such as InfiniBand or TCP that supports asynchronous data transmissions and RDMA.

Contents

1 Introduction

High-Performance Computing applications use parallelism in order to make best use of the available hardware, which is increasingly parallel. Writing parallel code requires handling communication between the parallel processes.

LAIK is used to simplify such parallel code and make it independent of a specific communication library. It is a library that manages the partitioning of the data and handles the communication that is needed for switching partitionings. Instead of directly calling into MPI or another communication library, the programmer calls LAIK to specify the partitioning of the data.

A partitioning describes which parts of the data should be available in the local memory of each process. Multiple processes may request the same part of the data for read-only access. They also can request to write to the same part of the data, but then a reduction operation (such as sum or minimum) must be provided for resolving the conflict. [1]

When the application switches to another partitioning, usually there is data that needs to be transferred between the processes. LAIK calculates the action sequence needed to do so, then uses one of its backends to execute that action sequence using some communication library.

Currently, the available backends are MPI, TCP2, TCP, and "Single". MPI uses MPI function calls. TCP2 and TCP both use TCP sockets, but TCP2 is newer. "Single" expects there to only be one process (i.e. a world size of 1), and has no support for communication across different nodes. [2]

LAIK also supports *malleability*. An application that uses LAIK may call `laik_allow_world_resize()` to specify a point where where the world size may change, i.e. additional processes may join or existing ones may be removed. This is useful, for example, to integrate LAIK with failure prediction: if an external tool that monitors the hardware predicts that a particular node will fail soon, it could ask LAIK to remove that node from the world during the next resize. [1]

However, currently only the TCP2 backend actually implements malleability. The MPI backend does not implement it because MPI itself does not support malleability. This is a problem because, as described in the comment on top of `src/backend-tcp2.c`, the TCP2 backend is intended for easy debugging, rather than for performance. Our measurements in chapter 4 show that it is indeed much slower than MPI. So there is no

backend that has both good performance and malleability.

In this thesis, we address that problem by implementing a new LAIK backend with malleability support, based on the *libfabric* library.

Libfabric is a low-level communications library for HPC, developed by the OpenFabrics Interfaces Working Group, that offers an abstraction on top of different providers, such as TCP, InfiniBand ("verbs"), or EFA (Amazon's Elastic Fabric Adapter). [3]

If multiple providers are available on the system, it is possible to specify which one to use during the initialization of libfabric. Our backend code is exactly the same regardless of whether libfabric sends the data through InfiniBand, through TCP, or through anything else. In section 4.1, we evaluate whether there is a performance difference between different libfabric providers for our backend.

Libfabric was chosen due to its support for asynchronous operations and for Remote Direct Memory Access (RDMA). Another appealing feature is its support for reliable unconnected data transfer, called *Reliable Datagram Messages* (RDM). For our backend, this means that we do not need to maintain a connection to every node within the world, like would have been necessary with connected data transfer, while also being guaranteed by libfabric that the data either arrives at the destination or or an error is reported. Reliable connected (similar to TCP) or unreliable unconnected (similar to UDP) data transfers are also supported, but we do not use them.

Development of our LAIK backend took longer than expected, and the lack of time prevented us from implementing shrinking and from fully debugging our implementation of the joining of new processes. It still works well enough to evaluate it (see subsection 4.1.3), and we believe that fixing the remaining test cases and implementing shrinking just needs more development time to get the details right, rather than new concepts or large amounts of new code.

2 Background

2.1 LAIK

A LAIK backend can implement the following functions (see: `include/laik/backend.h`):

- `exec()` executes an action sequence. Every backend needs to implement this.
- `prepare()` is called before `exec()`, and can be used to apply transformations to the action sequence or make other preparations for a more optimal execution in subsequent `exec()` calls. Both MPI and our Fabric backend use this, while TCP2 does not.
- `cleanup()` is called when the action sequence is no longer needed. If a backend allocated resources in `prepare()`, it needs to clean them up in `cleanup()`.
- `resize()` is called in every already-running process when the application allows a resize to happen. Here, processes check for join or delete requests, exchange addressing information with newly-joining processes, and update internal data structures. `resize()` returns a new group that contains all processes of the resized world, both the already-running (if there was no remove request for them) and the new ones that joined during the resize. Used by TCP2 and by our Fabric backend.
- `finishResize()` is used to free the resources associated with processes that were removed during the last resize. Used by TCP2.
- `sync()` synchronises the key-value store. Implemented by MPI and by TCP2, but not by our Fabric backend. So applications that use LAIK's key-value store will not work with the Fabric backend. None of the examples that come with LAIK use the key-value store, but there are some tests that test for it.
- Other functions: `updateGroup()` updates backend-specific group data. `log_action()` logs a backend-specific action. `make_progress()` asks the backend to progress, and is only used by TCP2. `finalize()` performs the final cleanup before the application exits.

Each LAIK backend must provide a `Laik_Backend` struct that contains the its name and the addresses of the functions that it implements. This struct gets passed to to `laik_new_instance()` during initialization, and LAIK uses it to invoke backend functions.

Note that there is no `init()` function in this struct. While most of the code for a new backend can be contained in one file and requires no changes to other LAIK code, `laik_init()` in `src/core.c` has to be modified to call the `init()` function for the new backend if the `LAIK_BACKEND` environment variable is set to that backend.

An action sequence as calculated by LAIK may contain a `TExec` action, which describes a transition that needs to be executed. `laik_aseq_splitTransitionExecs()` splits that `TExec` into send, receive, and reduce actions. Except for "Single", all the backends invoke this function. The resulting send/receive/reduce actions are easier both to execute and to perform further transformations with. These further transformations, which are used both by MPI and our Fabric backend, are discussed in more detail in section 3.2.

Each action stores its action type and additional information on how to execute it (e.g. for a send, the ID of the node that it needs to be sent to).

The simplest send/receive action types are `BufSend`, which sends data from a buffer in local memory to a receiver node, and `BufRecv`, which receives data from some node into a local buffer. However, the data that needs to be sent is not always already in one single local buffer. For these cases, there is `MapPackAndSend`, which packs data from multiple addresses into one buffer according to a specified mapping, then sends it. On the receiver side, its counterpart is `MapRecvAndUnpack`.

While there are many other action types (a full list is in `src/action.c`), the main ones that a backend needs to implement on its own are are `BufSend` and `BufRecv`. Action types like `MapPackAndSend` can be transformed into `MapPackToBuf` and then `BufSend`. Local actions such as `MapPackToBuf` either already have a default implementation provided by LAIK or can be implemented with a few lines of code that do not depend on backend-specific features.

A backend can also define its own backend-specific actions for internal use. Since for both MPI and for Fabric the underlying libraries have asynchronous sends/receives, they use backend-specific actions to split the `BufSend` into two halves, one that initiates the send and another that awaits its completion. They then move the wait to the end of the action sequence.

2.2 libfabric

As mentioned in the introduction, libfabric is a communication library offering an abstraction on top of different providers. It supports both regular send/receive operations, where the receiver must copy the data into the target buffer, and Remote Direct Memory Access (RDMA), where the networking hardware directly places the data into the target buffer and no action is required on the receiver side.

Both regular and RDMA data transfers complete asynchronously. Completions and errors are reported in a completion queue (CQ). Separate completion queues can be created for for send completions and receive completions.

2.2.1 Initialization

To initialize libfabric, first it is necessary to obtain information about available providers using the `fi_getinfo()` function. If specific properties are needed, such as support for RDMA or a provider with a particular name (*verbs*, *tcp*, etc.), these can be set in a `fi_info` struct that is then passed as an argument to `fi_getinfo()`, which writes out all providers that support these properties (or NULL if there are no such providers) into a linked list of `fi_info` structs.

One of these structs (our backend simply uses the head of the list) is then passed to `fi_fabric()` and `fi_domain()`, which open the fabric and domain described by the struct. In libfabric terminology, a domain usually represents an NIC, while a fabric refers to "a collection of hardware and software resources that access a single physical or virtual network. All network ports on a system that can communicate with each other through their attached networks belong to the same fabric." (`man fi_fabric` [3])

The command-line utility `fi_info`, which is installed as part of libfabric, can be used to list the providers, fabrics, and domains that are available on a system.

After the fabric and domain are open, other libfabric resources can be opened: endpoints, address vectors, completion queues, and event queues (though event queues are not used by our backend). Address vectors and completion queues must be associated with an endpoint using the `fi_ep_bind()` function. An endpoint must be bound to a completion queue, and if it is connectionless, also to an address vector, before it can be enabled.

After the endpoint has been enabled, its address can be obtained with `fi_getname()`. The address format depends on the provider: with *verbs*, the InfiniBand address is obtained, with *tcp*, an IPv4 or IPv6 address, and so on.

An address vector associates the address of an endpoint with a 64-bit unsigned integer (as described in the `fi_av` man page, "a simpler, more efficient value that can be used by the libfabric API in a fabric agnostic way" [3]). This integer is then used

instead of the endpoint address whenever an endpoint address needs to be specified, such as for `fi_send` or `fi_write`.

There are two different types of address vectors: *FI_AV_MAP*, which stores the addresses using a provider-specific mechanism and maps them to arbitrary numbers that must be saved by the application, and *FI_AV_TABLE*, which simply maps them to consecutive indices. The first inserted address gets index 0, the second gets index 1, and so on, and there is no need for the application to save these numbers.

We use *FI_AV_TABLE* for our backend, since this means that just by inserting the addresses into the AV in the correct order, we can make these indices match the node IDs used by LAIK. For example, if we are executing a LAIK `BufSend` action to node 10, we can simply specify 10 as the address:

```
fi_writedata(ep, buf, ..., 10, ...);
```

But before we can insert the addresses of the other nodes into the AV, we need to know these addresses. Libfabric appears not to have any mechanisms for address exchange between nodes. Our backend implements an address exchange using sockets; see section 3.1.

After the addresses have been inserted into the associated AV, the endpoint is ready to perform sends, receives, and RDMA.

2.2.2 Usage

Non-RDMA sends and receives are initiated using `fi_send()` and `fi_recv()`. These functions take the endpoint, the buffer, its length, and the target node address as arguments. However, while `fi_send()` always sends to the specified target node, `fi_recv()` will normally ignore the source address passed to it and receive from any node that sends data to it. To make it consider the source address, the endpoint must be configured with the `FI_DIRECTED_RECV` capability. The `verbs` (InfiniBand) provider does not support this capability, but `tcp` does.

The completion of a non-RDMA send or receive is reported through the completion queue associated with the endpoint. The completion queue can be read using `fi_cq_read()`, which is non-blocking, or using `fi_cq_sread()`, which blocks until either a completion report is available or a specified timeout is reached. Timeout can be disabled by setting it to a negative value.

Before an RDMA can be performed, the node whose memory will be accessed must register that memory for RDMA using `fi_mr_reg()`. We could not find information in the manual about what happens if an RDMA is attempted before the memory is registered, but a quick test using our backend shows that the data is never received, but also no error is reported to the sender.

Memory registration requires specifying a key. The sender will need to know this key when performing the RDMA, but it will not need to know the virtual address on the receiver's side, as the memory region to be accessed is specified using the key and an offset (in bytes) from the beginning of the registered memory buffer. Section 3.3 describes a way of calculating the same key independently in the sender and the receiver, so that no key exchange is required.

Alternatively, it would be possible to set the memory registration mode to `FI_MR_VIRT_ADDR`, which requires the sender to explicitly specify the virtual address that it wants to access. This would probably mean (we did not test it) that key calculation would no longer be needed, but an exchange of virtual addresses would be required instead.

For the sender, RDMA write completions are reported through the completion queue, just like regular send completions. But the receiver never initiated an operation that would be reported through the CQ. Indeed, it is possible to perform an RDMA that the receiver does not get notified of. But our backend requires a way to get notified about received RDMA writes so that the receiver can proceed from `BufRecv` to the next action.

The solution is `fi_writedata()`. It takes one additional parameter over the regular `fi_write()`: an unsigned 64-bit integer, which is the data. If the completion queue was configured with `FI_CQ_FORMAT_DATA`, once the RDMA completes, a completion queue report that contains the specified data is generated on the receiver side.

Note that there is no `fi_readdata()`, so a node cannot be notified about the completion of a read of its memory by a remote node.

At least for `fi_writedata`, a completion of an RDMA can be distinguished from the completion of a non-RDMA operation by the presence of the `FI_REMOTE_CQ_DATA` flag in the completion queue report, both on the sender and on the receiver side.

There are also vectored functions (such as `fi_writev`) for send/receive operations and RDMAs. These take an *iovec* as argument and can read or write multiple buffers on the same node with just one function call. Memory registrations can also be performed with a vectored function.

3 Design and Implementation

3.1 Initialization: address exchange

`laik_fabric_init()` initializes libfabric, performs the address exchange, and initializes LAIK. The initialization of libfabric has already been described in subsection 2.2.1.

Since in this part of the program, libfabric has not yet been fully initialized, the address exchange uses TCP sockets.

The address exchange requires exactly one master process that is listening on the address specified by the `LAIK_FABRIC_HOST` (default: localhost) and `LAIK_FABRIC_PORT` (default: 7777) environment variables. We copied the approach of the TCP2 backend for ensuring this:

First, every process checks whether it is running on the master host `LAIK_FABRIC_HOST`. This is done using the `check_local()` function. That function is part of the TCP2 backend, but since it does not use anything TCP2-specific, we can also invoke it from our backend. It works by attempting to bind a socket to an arbitrary port on the master host. The bind will only succeed if the process is running on that host. All processes where `check_local()` succeeded then attempt to bind to the port specified by `LAIK_FABRIC_PORT`. Since it is not possible for multiple processes to bind to the same port, only one process will succeed and the others will fail. The process that succeeded becomes the master.

Every non-master process connects to the master host and port and sends its own libfabric endpoint address (obtained using `fi_getname()`) to the master over this connection.

The master allocates an array for `LAIK_SIZE` many libfabric endpoint addresses, inserts its own endpoint address as the first entry, then receives the endpoint addresses from the other processes over the socket into the array. Once it has received `LAIK_SIZE-1` addresses, it sends out information about the world to the non-master processes. To each process, it sends the assigned ID of that process (which is just its index in the array of addresses), the world size, and the array of addresses. It also sends the phase and epoch, but when the master is in `laik_fabric_init()`, these are always zero. Non-zero phase only occurs after a resize.

After these steps, the master and all non-master processes have the same array of libfabric endpoint addresses. Every process calls `fi_av_insert()` to insert them into

the address vector. This completes the libfabric initialization.

Afterwards, LAIK data structures are set up. A LAIK instance and a world group are created and initialized. Memory for the acknowledgement array that our Fabric backend uses (see section 3.3) is also allocated.

Non-master processes check whether the phase is non-zero, i.e. whether the process joined during a resize. What it does if the phase is non-zero is described in section 3.4.

Finally, the LAIK instance is returned.

The initialization function also checks an environment variable `LAIK_FABRIC_MODE` and sends a variable `emode` from it. This was intended for evaluation of different implementation choices in this backend (see section 3.5), but never was actually used due to time constraints.

3.2 Preparation

3.2.1 Transformations

Our backend uses the action sequence transformations as the MPI backend, except for `laik_aseq_replaceWithAllReduce()`. First, any `TExec` are split into send, receive, and reduce operations. Then, `MapPackAndSend` / `MapRecvAndUnpack` are "flattened", i. e. replaced with with simpler actions: If the mapping is known and one-dimensional, no packing step is necessary, and the `MapPackAndSend` action can be replaced with a `BufSend` or a `MapSend`. Otherwise, a packing step is necessary. In the case, the flattening splits it up into separate actions: a `BufReserve` to reserve space for a temporary buffer, a `PackToRBuf` or `MapPackToRBuf` that packs the data into that buffer, and a `RBufSend` that sends the packed data.

Then, similar `BufSend` / `BufRecv` / `Reduce` actions are merged. Two `BufSend` are considered similar if they have the same target node and are in the same round. All the similar `BufSend` are replaced by a `CopyToRBuf` action that copies their buffers into a temporary buffer and an `RBufSend` that sends from that buffer. Merging `BufRecv` works the same way, but in the other direction.

After that, buffers are allocated and actions like `RBufSend`, `RCopyToBuf`, and so on are replaced with `BufSend`, `CopyToBuf`, and so on. The difference is that the actions without the `R` store a pointer to the buffer that is allocated in this step.

Reduce actions are split into multiple basic actions, and buffers are allocated again. Actions are then sorted by rounds; two actions being in the same round means that they have no dependence on each other. Similar actions are merged for a second time and buffers are allocated. Finally, sends and receives are sorted for deadlock avoidance.

All this is implemented by the LAIK library (our backend, like MPI, just invokes the functions that do this) in `src/action.c` [2], which also has comments that describe

some of these transformations in more detail.

These transformations result in an action sequence where the only actions that actually need to transfer data from or to other nodes are `BufSend` and `BufRecv`. There are no longer any `MapPackAndSend` or similar compound actions. This makes the implementation of the `exec()` function much simpler.

The reason why we do not use `laik_aseq_replaceWithAllReduce()` even though the MPI backup does is that it requires our backend to implement a `Reduce` action. Currently, we do not have a performant libfabric-based reduce implementation (though this is a consideration for the future; see section 3.5), so we do not use that transformation in order to avoid generating `Reduce` actions.

Finally, our `prepare()` performs a backend-specific transformation that splits `BufSend` into `FabAsyncSend`, which initiates the send, and `FabSendWait`, which awaits its completion. Since it does not matter which order the completion of the sends is reported in, only a single `FabSendWait` is placed at the end of the action sequence and awaits the completion of all the sends. `BufRecv` is renamed to `FabRecv` for consistency, but is not split. Receives are still awaited in place.

The idea with the split backend-specific actions was also copied from the MPI backend. Since both MPI and libfabric have sends that complete asynchronously, it also makes sense for our Fabric backend. However, there is one difference: MPI needs to make the buffer available for a receive using `MPI_IRecv` before it can await the completion of the receive. Our Fabric backend also needs to make the buffer available by registering its memory, but it is sufficient to do this once during the `prepare()`, it is not necessary to do this every time the sequence is executed. So instead of making it part of the action sequence, we register the memory in `prepare()`.

3.2.2 Memory registration

Our `fabric_aseq_RegisterMemory()` iterates over the action sequence, and for each `FabRecv` action, it registers the memory of the associated buffer for RDMA.

To avoid performing a key exchange, we need a key that both sender and receiver can calculate from the action sequence independently of each other. In our implementation, keys are derived from 3 numbers: the action sequence ID, the ID of the sender (`aa->from_rank`), and how many receives from this sender this action sequence already had (i.e. we initialize a `rcvcount` array with zeroes at the beginning of the prepare, then do `rcvcount[aa->from_rank]++` after each memory registration).

For example, if action sequence 123 looks like this:

```
FabRecv: from_rank = 3, buf = 0x12345
FabRecv: from_rank = 1, buf = 0x12345
```

```
FabRecv: from_rank = 3, buf = 0x54321
FabRecv: from_rank = 3, buf = 0x12345
```

Then the buffer at `0x12345` will be registered with the keys $(123, 3, 0)$, $(123, 1, 0)$, the buffer at `0x54321` with the key $(123, 3, 1)$, and finally the buffer at `0x12345` again with the key $(123, 3, 2)$. It is permitted to register the same memory multiple times with different keys, so the fact that some buffers have multiple keys is not a problem. What would not be permitted would be to register multiple buffers with the same key; that would fail with a "Requested key is already in use" error.

This works because for each receive in the action sequence, the sender node must have a matching send in its own action seqeuence. In `exec()`, every node maintains an array `sndcount` that counts how many sends were already sent to each node (i.e. after each send, it performs `sndcount[aa->to_rank]++`). This `sndcount` matches the `rcvcount` that was calculated on the other side during `prepare()`. The other two numbers are the ASeq ID and the sender's own ID.

Currently, the three numbers are just packed into a 64-bit integer to make the key (see `make_key()` in `src/backend-fabric.c`). The problem with this is that it imposes an artificial, arbitrary limit on the amount of action sequences, the amount of nodes within a world, and the number of receives from the same node within one action sequence. Specifically, 24 bits are used for the action sequence ID, 32 bits for the sender ID, and 8 bits for the `rcvcount`.

We considered whether this artificial limit could be avoided by making the key a hash of these three numbers instead of the numbers themselves. The problem is that our `exec()` uses the sender ID that it extracts from the key; since the completion queue report for a completed RDMA has no field that indicates the sender, this is the only way for it to obtain that information. But it should be possible to rewrite `exec()` so that it does not need the to know the sender ID. `exec()` determines whether a completed receive is the expected one by comparing the expected key with the key reported by the CQ. This would not be affected by making the key a hash of the numbers. `exec()` only needs the sender ID for the acknowledgements data structure (see section 3.3). If that structure is rewritten to not need the sender ID, it should be possible to use hashes and avoid this artificial limit.

To ensure that no RDMAs are attempted before all buffers are registered for RDMA, we put a barrier at the end of the `prepare()` function. All processes must reach the barrier before being allowed to proceed and return from the `prepare()`. This barrier is implemented by having all non-master processes report to the master (by sending 1 byte with `fi_send()`) and await 1 byte in response. The master waits until it has received `world_size - 1` sends, then sends a byte to every non-master node to allow them to proceed.

3.3 Execution: RDMAs with libfabric

Conceptually, our `exec()` is simple: `FabAsyncSend` uses `fi_writedata()` to initiate an RDMA write from the specified buffer (`aa->buf`) to the specified node (`aa->to_rank`). The RDMA key is calculated as described in section 3.2. `FabSendWait` reads the transmit completion queue until it has received the specified number (`aa->count`) of completion reports. `FabRecv` reads the receive completion queue until it receives the completion report for an RDMA write to the expected local buffer by the expected remote node.

Much development time was spent on figuring out the libfabric API and its peculiarities (see also subsection 2.2.2). The libfabric documentation provides excellent manpages for the individual functions of its API, but no usage examples for these functions, nor any examples of what a simple complete application looks like. There are some utilities and tests that come with libfabric, but these have a very complex structure. We are not aware of any other software that uses libfabric, either.

Also, `FabRecv`'s "read the CQ until the right completion report is received" is somewhat more complex than it sounds. Since there is no way to put back a report into the completion queue after reading it, if we received a completion report for a different RDMA than the one that we were awaiting, we need to store that information somewhere, so that a subsequent `FabRecv` does not read the CQ forever waiting for a message that we removed from the CQ. We will call that information an "acknowledgement" that the message with a particular key has been received.

Before `FabRecv` starts reading the CQ, it needs to check the stored acknowledgements for whether the message that it awaits has already been received. Without debug statements, the code looks like this:

```
case LAIK_AT_FabRecv: {
  Laik_A_FabRecv *aa = (Laik_A_FabRecv*) a;
  uint64_t key = make_key(as->id, aa->from_rank, rcvcount[aa->from_rank]);
  if (pop_ack(aa->from_rank, key))
    goto recv_done;
  while (1) { /* Data not received yet, await new completion reports */
    RETRYCQ(fi_cq_sread(cqr, (char*) &cq_buf, 1, NULL, -1), cqr);
    if (cq_buf.data == key) goto recv_done;
    else ack_rma(cq_buf.data);
  }
recv_done:
  rcvcount[aa->from_rank]++;
  break;
}
```

Currently, the implementation of `acks` is based on an array of size `world_size` of key lists, where key list *i* stores the keys of acknowledged messages sent by node *i*. The entire key list of node *i* is searched in every lookup. `ack_rma()` stores the key that it acknowledges in the first free spot of the list, or reallocates the list to be larger if there are no free spots left.

This is a terrible solution that was only chosen because it seemed like the most obvious way to do it at the time of development and there was no time to rewrite it into something better later.

If further development decides to replace this implementation, ideally with something that does not need information about the sender of the message (see the discussion of hashes for key calculation in subsection 3.2.2), they will just need to replace the functions `pop_ack()`, which searches for an acknowledgement of a given key and removes it if found, and `ack_rma()`, which stores an acknowledgement of a key, and adjust the allocation of memory for `acks` during `init()` and `resize()`.

Note that RDMA completion reports can also be read from the CQ outside of `exec()`: The `prepare()` function calls `barrier()`, which invokes `await_completions()`, which reads from the receive CQ. So when `await_completion()` reads a completion report from the CQ, it checks whether the flag `FI_REMOTE_CQ_DATA` is set in the report. If so, it is the completion of an RDMA and gets acknowledged using `ack_rma()`. Otherwise, it is the completion of a regular receive and is counted towards the number of completions that are being awaited.

The only information in a CQ report that we can actually use is the flags and the data. Neither the key, nor the sender, nor the buffer into which the message was placed, are reported by the CQ. There is a field `buf`, but it is only applicable to `fi_recv()` with the `FI_MULTI_RECV` flag set, not to our RDMAs. We work around the fact that the key is not reported by sending the key as the data in `fi_writedata()`.

To perform error handling for libfabric function calls while still keeping the code readable, we defined some macros:

- Many libfabric functions return zero if there was no error, and a non-zero value otherwise. There is a function `fi_strerror()` to translate the non-zero return value into an error message. Our `PANIC_NZ(a)` macro checks the return value of *a*, and if it is non-zero, logs the error message and exits.
- Libfabric sends and receives may return `-FI_EAGAIN`, in which case they should be tried again. Our `RETRY(a)` macro retries *a* as long as the return value is `-FI_EAGAIN`, and if the return value is non-zero after that, exits with the `fi_strerror()` error message.
- Reading from a completion queue may return `-FI_EAGAIN` like the sends and

receives, but it also returns `-FI_EAVAIL` if the top-most CQ entry is for a failed transfer. In that case, `fi_cq_readerr` can be used to retrieve information about the error. This is what `RETRYCQ` does.

Error handling only consists of printing an error message and exiting if any error is encountered; no attempt is made at recovery.

3.4 Resize

Our implementation of resizes is based on the resizes in the TCP2 backend. A resize is initiated when the application invokes LAIK's `laik_allow_world_resize()` function. For all existing nodes, this results in LAIK invoking the `resize()` function of our backends. Nodes that have not joined during the initial creation of the world, however, are still stuck in `init()` and waiting for the master node to accept their connection.

The master node, which is in `resize()`, polls whether there are any new connection requests on its socket (which is still bound to `LAIK_FABRIC_PORT` on `LAIK_FABRIC_HOST`). If so, it reads the addresses of the new nodes that are joining.

The amount of new nodes (or zero, if none) and the array of their addresses is sent to all existing nodes.

It would be possible to send the complete array of addresses, both old and new ones, to the newly-joining nodes right away. The problem is: the application that invokes LAIK may expect that the world group of a newly-joined node has a parent group that only contains the nodes that were there before the resize. For example, `examples/jac1d.c` expects this.

Therefore, at first the newly-joining nodes only get sent the old world size and address list, as well as the assigned ID for the new node and the current phase (which is non-zero when `resize()` is invoked and increases with every resize) and epoch, and only after that the amount and the addresses of the new nodes.

A newly-joining node receives the old world data, initializes the LAIK world group, and finally checks whether *phase* is non-zero. Since the node joined during a resize, *phase* is indeed non-zero. In this case, it first receives the current action sequence ID from the master and then does exactly the same as the `resize()` of non-master nodes, which is to call the `recv_new_addrs()` function to receive the addresses of new nodes.

After that, regardless of whether this is a master node, a non-master node, or a newly-joining node, `handle_new_addrs()` is called. It inserts the new addresses into the address vector, re-allocates buffers that depend on the world size (such as `acks`), then creates and initializes a new world group that contains both the old and the new nodes. The old world group is set as its parent.

This new world group is returned by `resize()`. The newly-joining nodes also have the old world group as a parent of the new world group because they also first create the old world group and then resize it, it just happens in a different function.

The reason why new nodes receive the action sequence number (`int aseq_id`, which was supposed to be a value internal to `src/action.c`, rather than get set by any backend) is because the key calculation described in subsection 3.2.2 only works if sender and receiver have the same action sequence ID for the same action sequence.

This could be solved by rewriting key calculation to not require the action sequence ID. If `rcvcount` would be modified to be a global count instead of being reset for each new action sequence, then the pair between sender ID and `rcvcount` would already be enough for a unique key for each memory registration. This would also require changing the memory registration cleanup code in `cleanup()`, since currently it takes advantage of the fact that the action sequence ID is encoded in the memory registration key.

3.5 Potential further work

Besides the functions described in section 2.2, libfabric offers some more advanced features that we did not have the time to evaluate, but which might improve the performance of our backend.

- Instead of having the action sequence transformations described in subsection 3.2.1 split `MapPackAndSend` into a sequence of actions that first copies the inputs into one temporary buffer and then sends that buffer, the vectored operation `fi_writev()` could be used to send the data from multiple buffers with just one function call, without the need for LAIK to perform copies of the data.

- Libfabric supports *collective operations*, such as reduces, broadcasts, all-to-all, and gather/scatter operations. These are described in the `fi_collective` man page. These could be used to implement a `Reduce` action like the MPI backend has.

  However, it is unclear whether they are supported by the Verbs provider. The TCP provider definitely does not support them because it does not support the libfabric atomics that the collective operations depend on. Verbs supports atomics, but the man page `fi_verbs` does not say anything about support for collective operations.

- According to the `fi_intro` man page (section "Address Vectors", subsection "Sharing AVs Between Processes"), it is possible for processes that run on the same node to share a single copy of an address vector, rather than one copy per process.

This would improve memory usage when there are many processes running on the same node.

- It is possible to make memory registration calls complete asynchronously rather than synchronously. Completions would be reported through the event queue. This might make `fabric_aseq_RegisterMemory()` run slightly faster, since it could start the registrations asynchronously while iterating through the action sequence, then await their completion at the end.

4 Evaluation

We evaluated our backend on two nodes of the BEAST hardware cluster at the Leibniz Supercomputing Centre. The nodes, *daos1* and *daos2*, have identical configurations:

- Memory: 374 GiB
- CPU: 2x 24-Core Intel Xeon Gold 6240R
- Linux kernel version: 5.14.21
- InfiniBand: Mellanox MT28908 Family (100000 Mbps, full-duplex)
- Network adapter: Intel Ethernet X722 (1000 Mbps, full-duplex)

All nodes in the BEAST cluster, including these two, are connected to an InfiniBand switch. MPI 4.1.2 is installed on both nodes. libfabric 1.17 is also installed, but it has been compiled without support for InfiniBand, so we had to compile and install libfabric 1.22 locally within our user directory. The CFLAGS for the compilation were just `-O2`. LAIK and the examples were compiled with `-O2 -g`. Unless otherwise specified, all measurements used commit `b32fba8bbba0acc92c24de88df70c629757ad6af` of the `fabric` branch of LAIK.

4.1 Measurements

4.1.1 Throughput: ping-pong

There are several usage examples that are part of LAIK and can also be used as benchmarks. We used ping-pong (`ping_pong.c`) to measure the largest throughput that a backend can achieve.

Ping-pong requires an even amount of processes. Two processes with adjacent process IDs (e.g. 0 and 1, 2 and 3, etc.) are considered a process pair. Ping-pong sets up a one-dimensional array of doubles and splits it evenly across the process pairs. Memory is reserved and action sequences are pre-calculated. Then, each process pair sends its part of the array back and forth between the first and the second process of

the pair, for a given number of iterations (10 by default). No computation is performed, the data is just sent back and forth.

Ping-pong measures how much real time (as provided by `gettimeofday()`) was spent executing the iterations, and calculates the throughput (in GB/s, where 1 GB is 10^9 bytes) as follows:

$$\text{throughput} = \frac{8.0\,\text{byte} \cdot \mathit{size} \cdot 2 \cdot \mathit{iters}}{\Delta t\,\text{sec} \cdot 10^9}$$

where 8.0 is the size of a double, *size* is the amount of doubles, $2 \cdot \mathit{iters}$ is how often it was transmitted (each iteration transmits it two times, first in one direction and then in the other), and Δt the total execution time in seconds.

There are options to turn off pre-calculation of action sequences and memory reservation, but these were not considered, as the goal was to measure the maximum throughput the backend can achieve when just sending data without additional calculations. The performance difference between pre-calculating action sequences or not is evaluated in later tests with Jac3D, which uses more action sequences than ping-pong.

A ping-pong with the default size of 100000000 doubles and 10 iterations between between *daos1* and *daos2* measures the following bandwidths for the different LAIK backends, and in the case of Fabric, different *libfabric* providers:

Backend	1	2	3	Average
Fabric (Verbs)	2.600500	2.592921	2.577000	2.590140
Fabric (TCP)	2.974513	2.936928	2.993294	2.968245
Fabric (UDP)	0.156010	0.147222	0.187039	0.163424
MPI	3.738841	3.764352	3.728997	3.744063
TCP2	0.117632	0.117633	0.117633	0.117633

Table 4.1: Throughput in GB/s for 3 invocations of `ping_pong` with default parameters across two nodes

`inxi -n` reports that the InfiniBand controller has a maximum throughput of 100000 Mbps (= 12.5 GB/s), while the Ethernet controller has 10000 Mbps (= 1.25 GB/s). This means that MPI uses approximately 30% of the maximum InfiniBand throughput, Fabric 23% of InfiniBand, and TCP2 approximately 9% of Ethernet throughput.

As expected, TCP2 (which was not designed for performance, and also uses the slower Ethernet connection) is the slowest. Both Fabric (except with the UDP provider) and MPI are significantly faster.

For the Fabric backend, the UDP provider is much slower than the other providers. Enabling LAIK logging (`LAIK_LOG=1`) confirms that it also selects `ib0`, not the slower Ethernet. The `fi_udp` man page states that it is a "basic provider that can be used on

any system that supports UDP sockets", and is intended for easier development, rather than performance [3]. So this is likely a result of the provider not being optimised, rather than any specific properties of UDP.

Surprisingly, the TCP provider, which uses TCP over the InfiniBand `ib0` interface, is measurably faster than the Verbs provider, which directly accesses InfiniBand. It should be noted that, as described in the `fi_verbs` man page, Verbs does not support RDM endpoints (which are used by our backend) directly and instead uses the RxD ("RDM over DGRAM") or RxM ("RDM over MSG") provider as an additional layer to emulate RDM. The TCP provider supports RDM directly since version 1.18, according to its man page. [3]

It is not clear whether that relates to the performance difference. In section 4.2, we profile the ping-pong with `perf` to look for other possible reasons.

For a ping-pong between two processes on *daos1*, the measurements are as follows:

Backend	1	2	3	Average
Fabric (Verbs)	4.456717	4.453464	4.256173	4.388785
Fabric (TCP)	5.642627	5.658217	5.694886	5.665243
Fabric (SHM)	4.702158	4.701286	4.705579	4.703008
Fabric (UDP)	0.531493	0.529914	0.533470	0.531626
MPI	3.083289	3.047862	3.030666	3.053939
TCP2	0.740936	0.737006	0.743194	0.740379

Table 4.2: Throughput in GB/s for 3 invocations of `ping_pong` with default parameters with two processes on the same node

In this case, MPI is slower than fabric. We thought that this might be caused by *xpmem* being broken on *daos1* and MPI having to fall back onto something slower. However, setting up a new OpenMPI 4.1.6 module where *xpmem* works does not improve performance:

Backend	1	2	3	Average
MPI 4.1.2 (no xpmem)	3.083289	3.047862	3.030666	3.053939
MPI 4.1.6 (xpmem)	2.966649	2.977013	3.026859	2.990174

Table 4.3: Comparison of the previous MPI measurement to MPI 4.1.6 with *xpmem* support. Throughput in GB/s for 3 invocations of `ping_pong` with default parameters.

Therefore, all further measurements were conducted with the old MPI module.

Here, we also evaluated the *libfabric* SHM provider, which uses shared memory and

the `process_vm_readv` / `process_vm_writev` Linux system calls for communication between processes on the same node (see `fi_shm` man page [3]). It is faster than the `verbs` backend, but surprisingly still slower than the TCP backend.

Since `process_vm_readv` and `process_vm_writev` use vectored reads and writes, it would have been interesting to evaluate whether the SHM provider performs better if the LAIK libfabric backend is extended to support vectored reads/writes with `fi_writev`, rather than having the `prepare()` function split them into individual reads/writes. This idea is described more detailedly in chapter 3, but we did not have the time to actually implement it.

4.1.2 Runtime: jac3d

As an example of an actual workload that involves computation, not just sending data across, we also evaluated Jac3D (`jac3d.c`). It computes a three-dimensional Jacobi matrix and every 10 iterations also the residuum. We chose this particular example because it has an option to prepare the action sequences one time and then execute them many times. Both the MPI and the Fabric backend spend more time in `prepare()` in order to optimize the action sequence for subsequent executions, so we expect them to perform better when the action sequence they prepared is actually executed multiple times instead of being immediately cleaned up after the first `exec()`.

Peformance on large world sizes

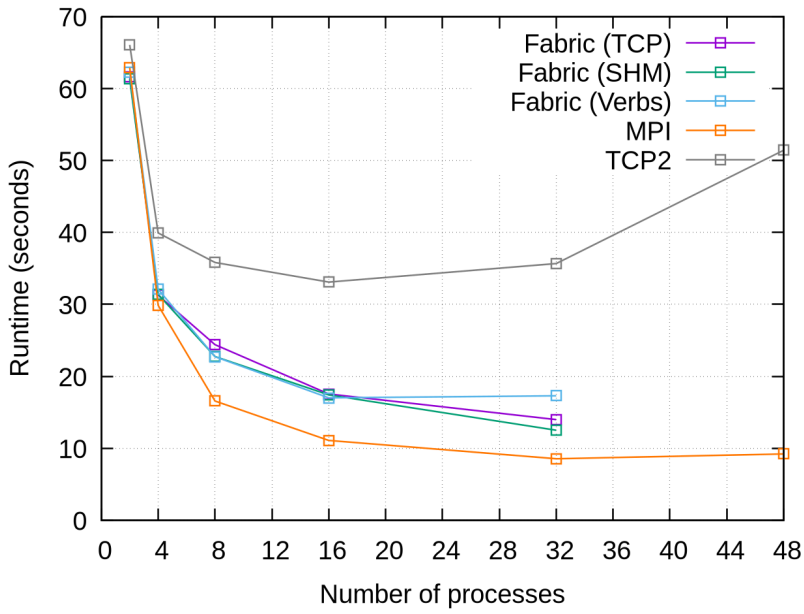
`jac3d -a 1000` computes a 1000 x 1000 x 1000 matrix. This size was chosen arbitratily, but it makes the program run both long enough to get meaningful time measurements even with 48 processes and short enough that measurements do not take hours. The `-a` flag enables the pre-calculation of action sequences.

To see how well the different backends scale when the LAIK world size is large, we measured the runtimes for `jac3d -a 1000` from 1 to 48 processes on *daos1*. The results are in Figure 4.1.

Measurements were performed using the `time` utility. We only measured the real time, not the CPU time or system call time. 48 was chosen as the maximum world size because it is the amount of cores on *daos1*. All numbers given are an average between 3 measurements. The differences between measurements are small enough that an average is sufficient for comparison between different backends (see Table 4.4).

For more than 32 processes, the Fabric backend did not run. This assertion failed in line 2117 of `src/action.c`, within `laik_aseq_addReduce3Rounds()`:

```
// collect values from tasks in input group
unsigned int bufOff[32], off = 0;
```



Backend	2	4	8	16	32	48
Fabric (TCP)	61.63	31.27	24.4	17.5467	13.9867	Fail
Fabric (SHM)	61.3167	31.3867	22.75	17.42	12.51	Fail
Fabric (Verbs)	62.23	32.09	22.7267	17.0133	17.2967	Fail
MPI	62.9033	29.87	16.57	11.0933	8.55	9.21667
TCP2	66.0533	39.9233	35.82	33.1	35.6733	51.42

Figure 4.1: Average runtime of `jac3d -a 1000` (in seconds), depending on world size

```
assert(inCount <= 32); // TODO: support more than 32 partitial inputs
```

This is a limitation of LAIK, not of the Fabric backend. While the Fabric backend uses almost the same transformations on the action sequence as the MPI backend, there is one difference: the MPI backend replaces certain group reduces with all-reduce, which is then executed using the `MPI_Reduce()` function. If that transformation is turned off by setting the environment variable `LAIK_MPI_REDUCE` to 0, the MPI backend also fails on that assertion.

These measurements show that our backend performs worse than MPI when there are more than 4 processes. For 16 processes, the Fabric backend is two times slower than the MPI backend.

Also worth noticing is the difference between the different Fabric providers for 32 processes. To determine whether it is a coincidence, we repeated Fabric measurements

Backend	2	4	8	16	32	48
Fabric (TCP)	0.16	0.2	0.3	1.44	1.51	Fail
Fabric (SHM)	0.76	0.22	0.38	3.75	1.48	Fail
Fabric (Verbs)	0.74	0.07	4.14	3.16	0.61	Fail
MPI	0.09	0.05	0.34	0.06	0.05	0.01
TCP2	0.33	1	1.54	1.36	2.3	1.7

Table 4.4: Difference in seconds between the shortest and the longest run of `jac3d -a 1000`, depending on world size

for the different providers 16 more times. As Figure 4.2 shows, while runtimes can vary by multiple seconds across measurements even with the same provider (verbs has the biggest difference with 5.06 seconds), the difference between providers is reproducible.

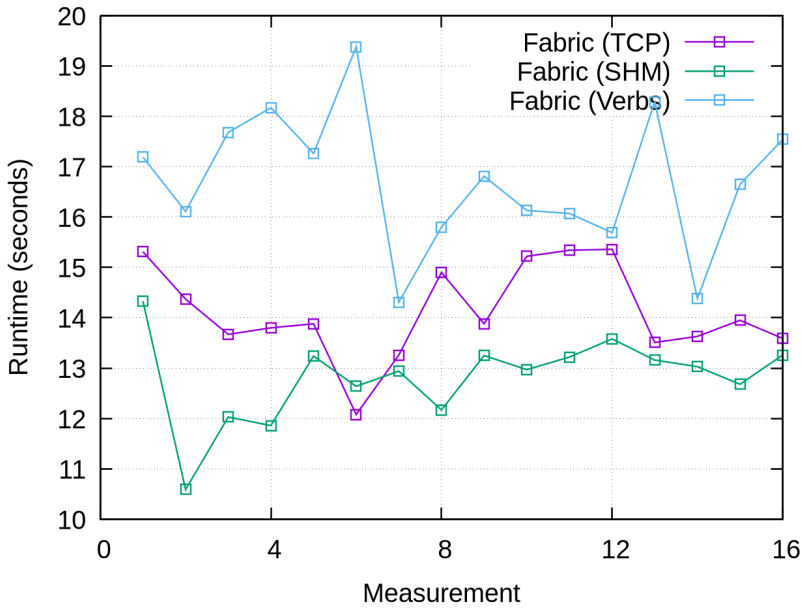


Figure 4.2: Runtimes (in seconds) of `jac3d -a 1000` with world size 32 in 16 consecutive measurements

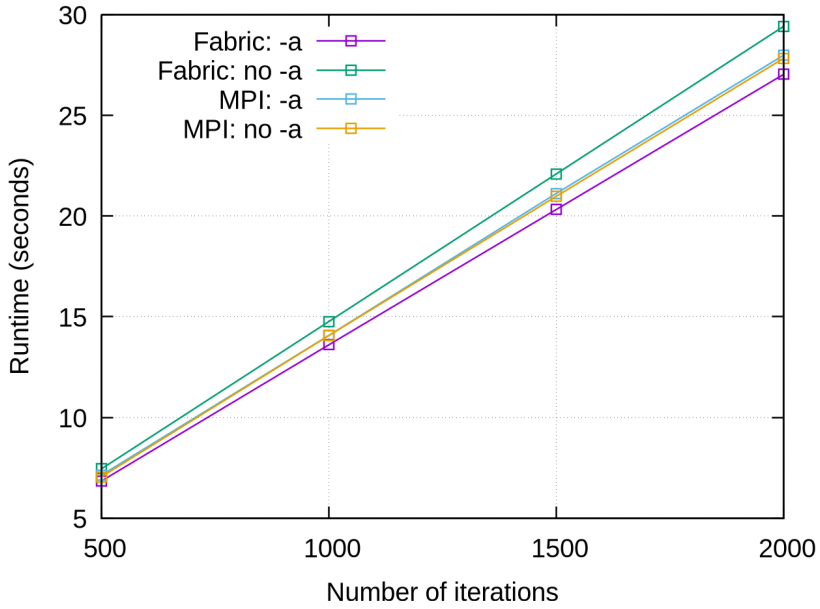
It would have been good to check whether the same difference occurs when increasing world size even further, but that was unfortunately not possible because the Fabric backend did not run with more than 32 processes.

Effect of pre-calculation of action sequences

We also measured how much pre-calculation of action sequences affects the performance of the Fabric backend.

If the `-a` flag is not set, Jac3D invokes LAIK's `laik_switchto_partitioning()` in every iteration and it calculates the necessary action sequences. If the `-a` flag is set, the action sequence is calculated only once at the start, and each iteration executes that pre-calculated action sequence. Therefore, the performance difference between `-a` and no `-a` should get larger with an increasing amount of iterations.

This is indeed what we measured, but not to the extent that we expected. Even for 2000 iterations, the difference between `-a` and no `-a` for Fabric is only 2.38 seconds. For MPI, it is even smaller.



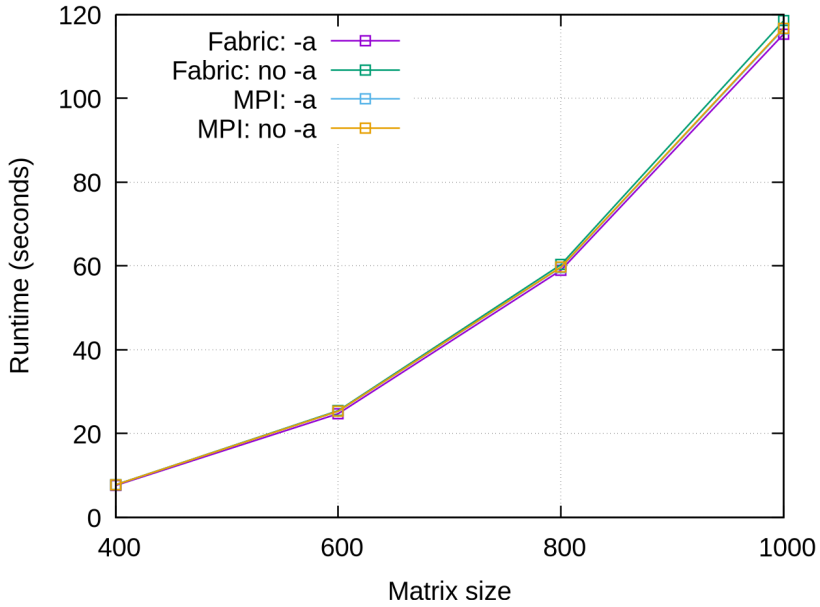
Backend	50	500	1000	1500	2000
Fabric: -a	0.76	6.844	13.606	20.322	27.032
Fabric: no -a	0.834	7.448	14.76	22.088	29.414
MPI: -a	0.862	7.138	14.072	21.118	27.982
MPI: no -a	0.856	7.052	14.066	20.974	27.828

Figure 4.3: Average runtime (in seconds) of `jac3d -a 200` and `jac3d 200`, for different numbers of iterations

These measurements were conducted between `daos1` and `daos2`. The Fabric backend

was invoked with the TCP provider. Averages were taken between 5 measurements to get these numbers; the biggest difference between measurements with the same parameters was 0.72 seconds.

Increasing the size of the matrix also appears to have an effect: for Jac3D with matrix size 1000 and 50 iterations, we measured 3.18 seconds difference between `-a` and no `-a` for the Fabric backend.



Backend	400	600	800	1000
Fabric: -a	7.62333	24.76	58.9867	115.323
Fabric: no -a	7.79333	25.4867	60.3133	118.507
MPI: -a	7.80333	25.3367	59.6467	116.527
MPI: no -a	7.79	25.36	59.7333	116.737

Figure 4.4: Average runtime (in seconds) of `jac3d -a` and `jac3d` with 50 iterations, for different matrix sizes

Note how for all these measurements, with `-a`, Fabric is slightly faster than MPI.

4.1.3 Resize

For evaluating the performance of resizes, we modified the `resize.c` example so that it measures how much time `laik_allow_world_resize()` uses. Since we again are interested in how it scales when there are many processes involved in the resize, we

measured on the *daos1* node, instead of sending data between *daos1* and *daos2*. Fabric used the TCP provider, and `resize.c` was invoked as `resize -s 2 -t 1`: `-t` is the option that we added for reporting resize times, and we only do one round of resizes.

When measuring with a large amount of processes, sometimes the TCP2 backend gets stuck waiting for some processes and never completes. Sleeping for two seconds instead of one (`-s 2`) appears to make it happen less often, so we used this option for our measurements. The Fabric backend has a different problem: occasionally, one of the processes fails with the error `Failed to listen on socket: Address already in use`. During development, we were unable to find the reason for the socket errors. For the resize measurements, this just means that the resize completes with one less process. Since this happenes infrequently and the resulting time measurements are indistinguishable from those of normal resizes, nothing was done to exclude them from the measurement.

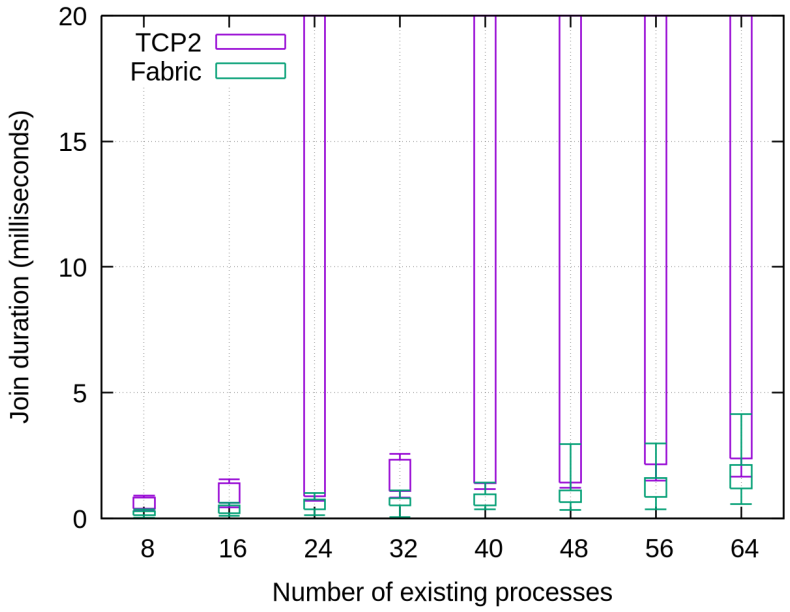


Figure 4.5: Resize time (in milliseconds) after one additional process joins a larger world

Figure 4.5 shows resize times when one additional process joins a larger number of existing processes. For each world size, the measurement was repeated 16 times, each time taking the average of the resize times that all existing processes reported. The lower and upper end of the box are the minimum and maximum of these averages. The minima and maxima of resize times were also recorded, and the smallest minimum

and largest maximum are the lower and upper ends of the whiskers.

These measurements show that the resize implemented in the Fabric backend has somewhat better performance than the resize in the TCP2 backend. They also show a strange behaviour of the TCP2 resize: while most of the time, the average resize time is less than 3 ms, occasionally (approx. 1 time per 16 measurements) an average of over 40 ms is measured. Figure 4.6 shows a version of the figure without these "outliers".

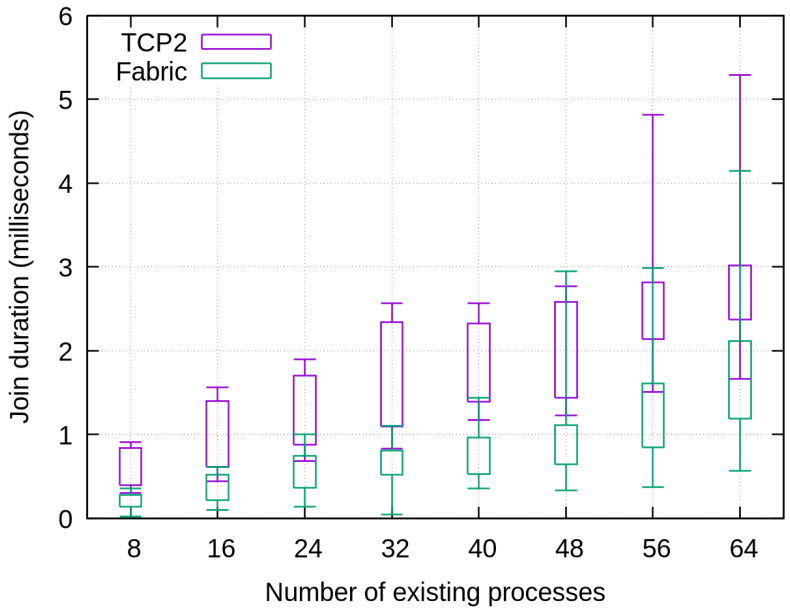


Figure 4.6: Resize time (in milliseconds) after one additional process joins a larger world — without outliers

We also measured the resize times when more than one process joins. In Figure 4.7, for a given world size n, n more processes join during the resize. For $n > 6$, TCP2 fails to receive all joins during just one resize round, so we could not measure above that. Due to the very short runtimes, we repeated the measurement 32 times, but otherwise the method was the same as for Figure 4.5. The results again show that the Fabric resize is faster.

However, our resize is not fully working yet. New processes joining works in `jac1d.c` and in `resize.c`, but not in `vsum3.c`. In `vsum3.c`, though the resize appears to work, the program then gets stuck when executing the next action sequence. Shrinking is also not yet supported.

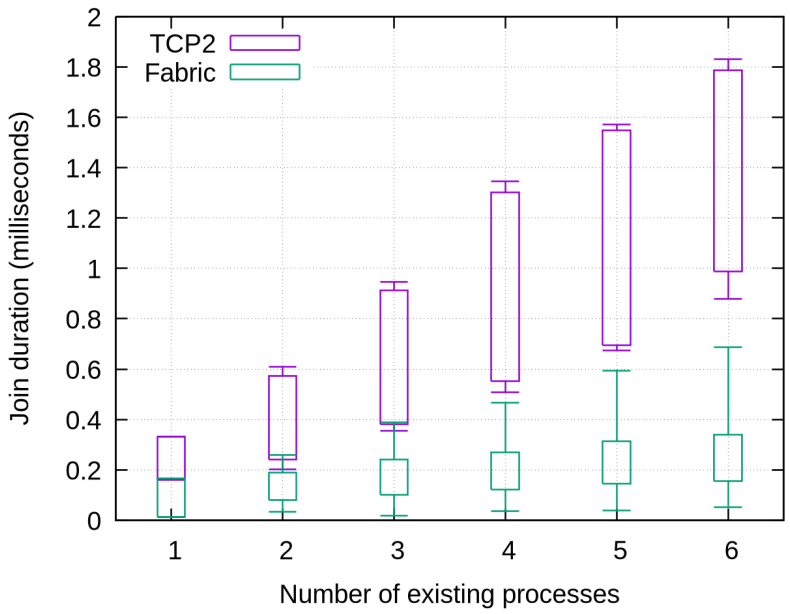


Figure 4.7: Resize time (in milliseconds) after n processes join a world of size n

4.2 Profiling

4.2.1 Ping-pong between daos1 and daos2

In order to determine why the *tcp* provider performs better than the *verbs* provider on a ping-pong test (subsection 4.1.1), we used `perf` to profile a ping-pong between *daos1* and *daos1* with the default size and amount of iterations.

It shows that the TCP provider spends most of the time in kernel functions:

```
74.35%  ping_pong  [unknown]              [k] 0xffffffffa617de1e
 3.70%  ping_pong  [unknown]              [k] 0xffffffffa617d7d7
 1.80%  ping_pong  ping_pong              [.] main
 1.40%  ping_pong  [unknown]              [k] 0xffffffffa6414d1a
 0.60%  ping_pong  [unknown]              [k] 0xffffffffa6414d0a
```

While Verbs spends much time on locking and unlocking mutexes:

```
36.86%  ping_pong  libc-2.31.so          [.] __memcpy_evex_unaligned_erms
14.74%  ping_pong  libpthread-2.31.so    [.] pthread_mutex_lock
 9.30%  ping_pong  libpthread-2.31.so    [.] __pthread_mutex_unlock_usercnt
 3.86%  ping_pong  libpthread-2.31.so    [.] pthread_spin_lock
```

```
 3.37%  ping_pong  [unknown]            [k] 0xffffffffa617d7d7
 2.02%  ping_pong  [unknown]            [k] 0xffffffffa65fd21c
 1.74%  ping_pong  libmlx5.so.1.24.40.0 [.] mlx5dv_get_vfio_device_list
```

For comparison, the MPI backend of LAIK spends much less time on locks:

```
48.75%  ping_pong  [unknown]            [k] 0xffffffffa617de1e
10.34%  ping_pong  [unknown]            [k] 0xffffffffa65fd21c
 2.29%  ping_pong  [unknown]            [k] 0xffffffffa617d7d7
 1.37%  ping_pong  [unknown]            [k] 0xffffffffa5f344d8
 1.07%  ping_pong  [unknown]            [k] 0xffffffffa5f55c14
 0.96%  ping_pong  libpthread-2.31.so   [.] pthread_mutex_lock
 0.95%  ping_pong  ping_pong            [.] main
 0.91%  ping_pong  libc-2.31.so         [.] poll
```

As LAIK is not a multi-threaded application, this locking appears unnecessary and may be the reason why Verbs is measurably slower than TCP. We tried explicitly setting the libfabric threading model is `FI_THREAD_DOMAIN`, which means that only one thread at a time will access any object belonging to the same domain, but that does not significantly affect the time that Verbs spends on locking:

```
37.55%  ping_pong  libc-2.31.so         [.] __memcpy_evex_unaligned_erms
13.79%  ping_pong  libpthread-2.31.so   [.] pthread_mutex_lock
 8.88%  ping_pong  libpthread-2.31.so   [.] __pthread_mutex_unlock_usercnt
 3.77%  ping_pong  libpthread-2.31.so   [.] pthread_spin_lock
```

It is also very strange to see Verbs spend so much time on a `memcpy()` that none of the other backends perform. While it is to be expected that some `memcpy()` will occur due to LAIK copying data between its internal buffers, these 37.55% of time spent on `memcpy()` cannot be caused by LAIK, because then they would have been reported in all providers and not just in Verbs.

4.2.2 Ping-pong locally on daos1

Measuring with two processes on the same node again shows the TCP provider spending all its time in kernel functions:

```
78.30%  ping_pong  [unknown]            [k] 0xffffffffa617de1e
 4.06%  ping_pong  [unknown]            [k] 0xffffffffa617d7d7
 0.62%  ping_pong  [unknown]            [k] 0xffffffffa64dec81
 0.47%  ping_pong  [unknown]            [k] 0xffffffffa5ea28ba
 0.42%  ping_pong  [unknown]            [k] 0xffffffffa6414c82
```

While Verbs again spends much of its time on locking and memcpy:

```
41.99%  ping_pong  libc-2.31.so          [.] __memcpy_evex_unaligned_erms
12.38%  ping_pong  libpthread-2.31.so    [.] pthread_mutex_lock
10.88%  ping_pong  libpthread-2.31.so    [.] __pthread_mutex_unlock_usercnt
 3.50%  ping_pong  [unknown]             [k] 0xffffffffa617d7d7
 3.09%  ping_pong  libpthread-2.31.so    [.] pthread_spin_lock
```

The SHM provider, which uses shared memory, spends much less time on locking and more time on kernel functions.

```
40.59%  ping_pong  [unknown]             [k] 0xffffffffa617de1e
21.30%  ping_pong  [unknown]             [k] 0xffffffffa65fd21c
 3.16%  ping_pong  [unknown]             [k] 0xffffffffa617d7d7
 2.15%  ping_pong  libpthread-2.31.so    [.] __pthread_mutex_unlock_usercnt
 1.97%  ping_pong  libc-2.31.so          [.] sched_yield
 1.91%  ping_pong  libpthread-2.31.so    [.] pthread_mutex_lock
```

The profile for MPI on *daos1* looks completely different to the MPI profile in the previous subsection:

```
44.80%  ping_pong  libc-2.31.so            [.] __memcpy_evex_unaligned_erms
22.68%  ping_pong  libopen-pal.so.40.30.2  [.] mca_btl_vader_component_progress
12.72%  ping_pong  libopen-pal.so.40.30.2  [.] opal_progress
 5.36%  ping_pong  libmpi.so.40.30.2       [.] ompi_coll_libnbc_progress
 4.76%  ping_pong  libopen-pal.so.40.30.2  [.] opal_timer_linux_get_cycles_sys_timer
 2.14%  ping_pong  [unknown]               [k] 0xffffffffa617d7d7
```

4.2.3 jac3d with a large world size on daos1

We also wanted to profile the Fabric and the TCP backends for a large world size to look for explanations for the performance difference observed in subsection 4.1.2. We ran `jac3d -a 1000` with 32 processes on *daos1*. However, the profiles do not show anything that would explain the performance difference.

This is the profile for the Fabric backend with the SHM provider:

```
3.44%  jac3d    jac3d              [.] main
5.20%  jac3d    [unknown]          [k] 0xffffffffa65fd21c
2.73%  jac3d    libc-2.31.so       [.] __memcpy_evex_unaligned_erms
2.29%  jac3d    libc-2.31.so       [.] sched_yield
```

```
2.28%  jac3d    libpthread-2.31.so    [.] pthread_mutex_lock
2.10%  jac3d    libpthread-2.31.so    [.] __pthread_mutex_unlock_usercnt
1.77%  jac3d    [unknown]             [k] 0xffffffffa6800158
1.72%  jac3d    [unknown]             [k] 0xffffffffa617d7d7
1.60%  jac3d    [unknown]             [k] 0xffffffffa660d2c3
```

And this is the profile for the MPI backend:

```
82.14%  jac3d    jac3d                  [.] main
 5.71%  jac3d    libc-2.31.so           [.] __memcpy_evex_unaligned_erms
 3.70%  jac3d    libopen-pal.so.40.30.2 [.] mca_btl_vader_component_progress
 2.34%  jac3d    [unknown]              [k] 0xffffffffa617d7d7
 1.32%  jac3d    liblaik.so             [.] unpack_lex
 1.11%  jac3d    jac3d                  [.] setBoundary
 0.93%  jac3d    libopen-pal.so.40.30.2 [.] opal_progress
 0.36%  jac3d    liblaik.so             [.] pack_lex
 0.35%  jac3d    libmpi.so.40.30.2      [.] ompi_coll_libnbc_progress
```

5 Conclusion

We implemented a new backend for LAIK based on *libfabric*. It passes all of the common tests (from the `tests/common/` directory) of the LAIK test suite, except for those that require KVS and `test-spmv2-shrink`. Our evaluations show that our Fabric backend achieves comparable performance to the MPI backend when there are only two nodes, but performs worse than MPI for larger world sizes.

We also implemented basic resize functionality in our backend, and measured that it has better performance than the TCP2 resize.

Our implementation shows that a malleable backend based on *libfabric* is feasible, and provides a base upon which further work may be done.

Many things still need to be fixed before the backend is ready for production use: Shrinking still needs to be implemented, and resize needs to be fixed so that it also works in the *vsum3* test case (`tests/fabric/test-vsum3-2-2.sh`). The reason why the initialization of our backend occasionally fails with `Failed to listen on socket: Address already in use` needs to be found and fixed. We attempted it, but we did not find the reason. Some general code cleanup also needs to be performed, ideally replacing some of the quick-and-dirty solutions described in chapter 3 (such as the key generation that imposes arbitrary limits on the maximum number of action sequences and nodes or the array for storing the acknowledgements) with more optimized and/or reliable solutions.

Measurements with `perf` show that the Verbs provider spends a surprisingly large amount of time on a `memcpy()` that none of the other providers or backends performs, and on `pthread_mutex_lock()`. Further work with this backend should investigate why that happens.

After solving these issues, more advanced features of libfabric (as described in section 3.5) could be evaluated to see if they can improve the performance of this backend.

An integration with the shared memory backend that is being developed for LAIK, so that shared memory is used for communication between processes on the same node and libfabric for communication between different nodes, could also be attempted and its performance compared to the performance of just using libfabric between all processes.

List of Figures

List of Tables

Bibliography

[1] Carsten Trinitis Josef Weidendorfer Dai Yang. "LAIK: A Library for Fault Tolerant Distribution of Global Data for Parallel Applications." In: *Konferenzband des PARS'17 Workshops*. Gesellschaft der Informatik. 2017, p. 10.

[2] Josef Weidendorfer, Alexander Kurtz, Dai Yang, Amir Raoofy et al. *LAIK source code*. `https://github.com/envelope-project/laik`. Sept. 15, 2024.

[3] *Libfabric Programmer's Manual*. Version 1.22. OpenFabrics Interfaces Working Group. 2024.