DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Design and Implementation of a Shared Memory Backend for LAIK

Robert Hubinger

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Design and Implementation of a Shared Memory Backend for LAIK

Design und Implementierung eines Shared Memory Backends für LAIK

Author:	Robert Hubinger
Supervisor:	PD Dr. Josef Weidendorfer
Advisor:	M.Sc. Amir Raoofy
Submission Date:	15.09.2022

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

Munich, 15.09.2022 Robert Hubinger

Acknowledgments

I would like to thank my advisor, Amir Raoofy as well as my supervisor Josef Weidendorfer for their support during the whole process of creating this thesis. Thank you for the countless hours of discussions, guidance and advice.

Abstract

LAIK is a library for data management in the HPC environment. Its focus lies on providing lightweight fault tolerance and load balancing mechanisms. The LAIK library sits between the application and the communication library used to migrate the data. Multiple backends are supported and application programmers can implement new backends using their own communication libraries. Shared memory is a promising communication method for a backend as it is a highly efficient method for inter process communication. We present designs for a standalone shared memory backend as well as a secondary backend version which executes the data migration of other backends on shared memory. We implement our designs and compare their performance against other LAIK backends. Finally, we outline possible optimizations as a basis for future work.

LAIK ist eine Bibliothek zur Datenverwaltung im HPC Umfeld. Ihr Schwerpunkt liegt auf der Bereitstellung leichtgewichtiger Fehlertoleranz- und Lastausgleichsmechanismen. Die LAIK-Bibliothek liegt zwischen der Anwendung und der Kommunikationsbibliothek, die für die Migration der Daten verwendet wird. Es werden mehrere Backends unterstützt und Anwendungsprogrammierer können neue Backends für ihre eigenen Kommunikationsbibliotheken implementieren. Sared Memory ist eine vielversprechende Kommunikationsmethode für ein Backend, da es eine sehr effiziente Methode für die Kommunikation zwischen Prozessen ist. Wir präsentieren Entwürfe für ein eigenständiges Shared-Memory-Backend sowie eine Sekundärbackend-Version, die die Datenmigration anderer Backends auf Shared Memory ausführt. Wir implementieren unsere Designs und vergleichen deren Leistung mit anderen LAIK-Backends. Schließlich skizzieren wir mögliche Optimierungen als Grundlage für zukünftige Arbeiten.

Contents

1 Introduction

LAIK is a library for data management in the HPC environment. Its focus lies on providing lightweight fault tolerance and load balancing mechanisms. The LAIK library sits between the application and the communication library used to migrate the data. Multiple backends are supported and application programmers can implement new backends for their own communication libraries. Shared memory is a promising candidate for such a new backend as it is a very efficient method for inter process communication. The only disadvantage of shared memory is that it is limited to one node as different nodes can not share memory.

Our work presents the design and implementation of two shared memory based backends. We will implement both a standalone backend capable of running LAIK applications on a single node as well as a secondary backend which will provide shared memory based data migration for other LAIK backends. We will first introduce the core structure of the LAIK library and the used IPC mechanisms in chapter 2. The design of our library will be presented in chapter 3. There we will elaborate on the initialization of the backend, the data transport over shared memory and the differences between the two different backend versions. In chapter 4 we will explain how the previously described designs were implemented. We will explain the structure of our backend and how a backend can use the functionality of our secondary backend version. We will then go into detail about our implementation of the initialization process and the data transport. Our implemented design will then be tested against other LAIK backends in chapter 5. We conclude our work by discussing possible optimizations for future work in chapter 6

2 Background

Understanding the LAIK library as well as Inter Process Communication is fundamental for understanding the design of our shared memory backends. This chapter is divided into two parts. In the first, we will briefly introduce the fundamentals of LAIK, with particular emphasis on the topics of action sequences, the backend interface and data storage in LAIK. In the second part we will cover the necessary fundamentals of inter process communication, especially shared memory and semaphores.

2.1 LAIK

LAIK, which stands for „Leichtgewichtige AnwendungsIntegriete Datenhaltungskomponente“, is a library for data management in the HPC environment. Created out of the need for higher flexibility in regard to scheduling and fault tolerance strategies, LAIK provides support for distributing data across parallel applications by controlling the data and its partitioning. The goal of LAIK is to provide fault tolerance mechanisms and load balancing for HPC applications in the most lightweight and performant way possible [1]. As shown in Figure 2.1, LAIK sits between the application and the library used for communication.

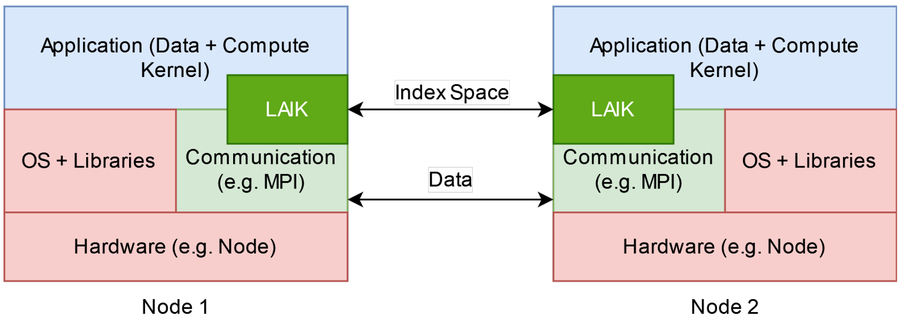

Figure 2.1: LAIK and communication backend[1]

2.1.1 Action Sequences

Action sequences are lists of actions which provide the information a node needs to execute a specific chain of actions. Actions are the atomic unit of execution in LAIK, they are predefined procedures which contain the necessary information for their own execution. LAIK creates action sequences based on the applications needs. After a sequence is created it can be optimised with the method `prepare()` before it is executed. When an action sequence is executed, the corresponding backend performs every action of the sequence in the set order. Action sequences can be executed multiple times. When an action sequence is not needed any more, it can be deleted by calling `cleanup()`.

2.1.2 Backend Interface

In LAIK backends are used to execute the data migration. As per its specification, LAIK supports different communication libraries to execute the data migration via a backend interface[1]. When executing a LAIK application, the backend can be chosen by setting the environment variable `LAIK_BACKEND` to the desired backend. Application programmers are able to implement new backends to use their desired communication library for LAIKs data migration [1].

```
struct _Laik_Backend {
        char* name;
        void (*finalize)(Laik_Instance*);
        void (*prepare)(Laik_ActionSeq*);
        void (*cleanup)(Laik_ActionSeq*);
        void (*exec)(Laik_ActionSeq*);
        void (*updateGroup)(Laik_Group*);
        void (*sync)(Laik_KVStore* kvs);
        bool (*log_action)(Laik_Action* a);
        void (*make_progress)();
        Laik_Group* (*resize)(Laik_ResizeRequests*);
        void (*finish_resize)();
};
```

Figure 2.2: A shortened version of the Laik_Backend struct.

To implement a new backend, a developer has to write his own versions of the methods defined in `struct _LAIK_Backend` (Figure 2.2). Those functions are then given as pointers to the backend struct. After that LAIK uses those functions to perform its backend actions if the corresponding backend was chosen. A developer does not need to implement all functions defined in the backend interface to implement his backend, he can also leave out functions if he does not need them for his backend. Apart from the functions defined in the backend struct, an initialization function with the signature `Laik_Instance *laik_init_shmem(int *argc, char ***argv)` must also be provided.

For our purposes, the most important methods are `init()`, `prepare()`, `exec()`, `cleanup()` and `finalize()`. The initialization method is used to create a `Laik_Instance`. For that, the different nodes need to get aware of their peers and establish means of communication. The method `prepare()` optimizes the passed action sequence for the given backend. The optimization happens in multiple passes which replace, add or delete actions. This can also include the addition of backend specific actions, which are custom actions used in only one backend. The `exec()` function executes a given action sequence which means that exec iterates over the given action sequence and runs every action in the order set by the sequence. To delete an action sequence, `cleanup()` is used. When a LAIK application terminates, `finalize()` is called to free any space allocated by the backend of the given laik instance.

2.1.3 Data Storage

To store and manage an applications data, LAIK uses the struct `Laik_Data`. When an application needs to store data it creates a `Laik_Data` struct by calling for example `Laik_Data* laik_new_data(Laik_Space* space, Laik_Type* type)`.

```
struct _Laik_Allocator {
       Laik_MemoryPolicy policy;

       Laik_malloc_t malloc;
       Laik_free_t free;
       Laik_realloc_t realloc;

       void (*unmap)(Laik_Data* d, void* ptr, size_t length);
};
```

Figure 2.3: A shortened version of the Laik_Allocator struct.

For allocating space to store data, the `Laik_Data` struct has a pointer to a `Laik_Allocator` struct (Figure 2.3). A `Laik_Allocator` contains pointers to a malloc and a free function. By default, those two function pointers point to wrappers for malloc and free from the standard library. A developer can also implement custom allocation and free functions to create a different allocator.

2.2 Inter Process Communication

When processes have to communicate with each other, wait for each other or exchange data and resources, Inter Process Communication (IPC) gets used [2]. IPC is a collective term for various mechanisms which enable two or more processes to work together. The most popular IPC mechanisms are (named) pipes, message queues, streams, sockets, shared memory and semaphores [2].

To implement our shared memory backend we used sysV shared memory and sysV semaphores. The shared memory is mostly used for data transport and the initialization of the backend. The semaphores are used to synchronize the access to the shared memory since shared memory is not synchronized by itself.

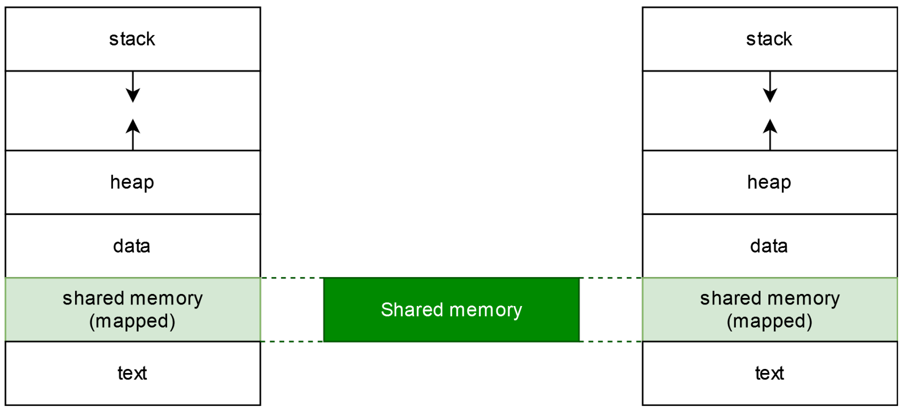

Figure 2.4: Schematic address spaces of two processes attached to the same shared memory segment[3]

2.2.1 SysV Shared memory

To perform the data transport and initialization of our backend, we used sysV shared memory. Shared Memory is memory shared between two or more processes which can be read and written by different processes. The memory segment is created by the kernel and mapped to the data segments of each attached process as can be seen in Figure 2.4. In comparison to other types of IPC like pipes, shared memory can be accessed directly without needing to copy the data to the processes data segment [2]. The attached processes can access shared memory just like normal memory. SysV shared memory is not synchronized, therefore developers need to implement their own synchronization mechanisms.[2].

```
int shmget (key_t key, size_t size, int shmflg);
void *shmat (int shmid, const void *shmaddr, int shmflg);
int shmdt (const void *shmaddr);
int shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

Figure 2.5: The signatures of the four sysV shared memory calls[2]

SysV shared memory can be controlled with the four shared memory calls seen in Figure 2.5. The function `shmget()` creates or opens a shared memory segment and returns the corresponding shared memory id on success. On failure it returns -1. The first argument `key` is a unique identifier for the shared memory segment, two calls with the same key will allays try to open the same segment. The parameter `size` specifies the size of the shared memory segment in bytes. 0 can also be passed if the segment already exists. The flag `shmflg` manages the access privileges: `IPC_CREAT` allows the call to create a new segment, `IPC_EXCL` assures in combination with `IPC_CREAT` that the shared memory segment was created by the process and did not exist previously. Should the segment allready exist, `shmget()` will return -1[2].

The second call `shmat()` returns a pointer to the shared memory segment with the id `shmid`. On Failure it returns -1. The second argument `shmaddr` tells the kernel where to attach the segment, due to portability reasons it is recommended to pass `NULL`. The last argument `shmflg` can be set to `SHM_RDONLY` so the attached process is only able to read, otherwise 0 is passed[2].

To detach an attached shared memory segment the call `shmdt()` is used. The argument `shmaddr` is the pointer to the shared memory segment the application wants to detach from. On success it returns 0, on failure -1[2].

The last call `shmctl` is used to query or change the status of shared memory segments. For our purposes the delete functionality is most important. To delete a shared memory segment with id `shmid`, the second argument `cmd` must be `IPC_RMID` and `buf` must be 0. The function returns 0 on success and -1 on failure[2].

2.2.2 SysV Semaphores

To synchronize our backends shared memory segments we decided to use sysV semaphores. A semaphore is a data structure which can be used to control the access to a resource. If a process wants to access the resource he has to check that the semaphores value is positive. If the semaphores value is positive the process sets the value to 0 and is able to use the resource after that. After the process is done using the resource he increments the semaphore to allow other processes to access the resource.

```
int semget(key_t key, int n_sems, int flag);
int semctl(int semid, int sem_num, int command, union semun arg);
int semop(int semid, struct sembuf *sops, size_t nsops);

struct sembuf {
unsigned short sem_num;
short sem_op;
short sem_flg; };
```

Figure 2.6: The signatures of the three sysV semaphore calls and the definition of the sembuf struct [2]

SysV semaphores are managed in semaphore sets which can be controlled with the three calls seen in Figure 2.6. To create or open an existing semaphore set `semget()` is used. Parallel to sysV shared memory `key` is a unique identifier for the semaphore. The second argument `n_sems` specifies the number of semaphores in the semaphore set. The last parameter `flag` works like the one for `shmget` of sysV shared memory. The call returns the id of the semaphore set on success and -1 on failure[2].

The `semctl` call is used to control a set of semaphores. The first argument `semid` is used to identify the semaphore set and `sem_num` is used to specify the semaphore value. The argument `command` specifies the action. For our purposes the most important actions are: `SETVAL` which sets the value of a semaphore variable and `IPC_RMID` which deletes the semaphore set. The union `semun` is used as an additional parameter for the commandos. When using `SETVAL`, it specifies the value we want to set the semaphores to. When using `IPC_RMID`, 0 can be given as an argument. In our usecases the function returns 0 on success and -1 on failure[2].

To perform operations on the semaphores the `semop()` call is used. The first argument `semctl` identifies the semaphore set. The second argument `sops` points to an array of semaphore operations of type `struct sembuf` which specifies the operations to be performed on the semaphore set. Each `sembuf` struct contains `sem_num` to choose the semaphore from the set, `sem_op` to specify the operation and `sem_flg` to pass extra options. If `sem_op` is greater than 0, the semaphore gets freed, if its is less than zero it gets locked and if it is equal to zero the call immediately returns. The call waits until he can access the semaphore or if the flag `IPC_NOWAIT` is set it returns an error if it can not immediately access the semaphore. The flag `SEM_UNDO` ensures that an accessed semaphore gets reset if the process should terminate prematurely. The last parameter `nsops` is the length of the `sops` array[2].

3 Design

The previous chapter introduced the core design of LAIK and the IPC mechanisms used in our implementation. The following chapter explains the top level architecture of our shared memory backend. We design two versions of our shared memory backend: a standalone version able to run on a single node and an integragted version which supports another backend with efficient shared memory based inter node communication. We first explain how the backend initializes itself and transports data over shared memory. We then move on to explain how the standalone version was changed to be able to function as a secondary backend assisting another Backend with a different communication library.

3.1 Initialization

To be able to transport data across our processes, the processes first have to get aware of their peers and establish adresses to be able to adress another. During initialization, the processes establish the required means of communication as well as a master process. After initialization, every process needs to be aware of its own adress and the size of the group.

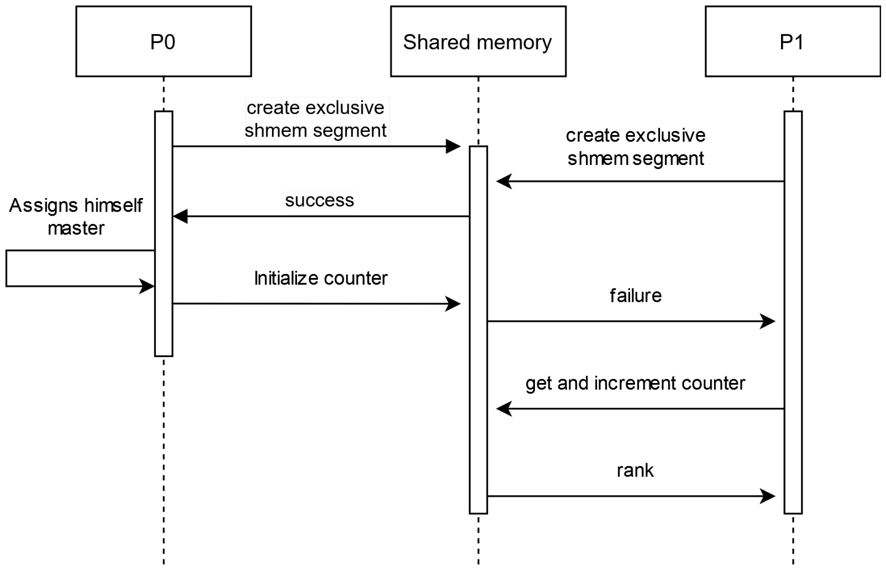
P0
Shared memory
P1
create exclusive shmem segment
create exclusive shmem segment
Assigns himself master
success
Initialize counter
failure
get and increment counter
rank

Figure 3.1: Exemplary process of an initialization with two processes

To start the initialization, the processes first need to establish a preliminary means of communication on which they can assign their addresses. Those addresses will be called ranks in the following. To start initialization, all processes will try to create the same exclusive shared memory segment. That way only one process can succeed in creating this segment. This process assigns himself the rank 0 which means he will be the master of the group. He will also initialize a counter in the created shared memory segment which will be used to assign the processes their ranks. All other processes are going to fail at creating the exclusive shared memory segment, which means they will be client processes. Those processes will then open and access the segment to increment the counter and assign themselves the new counter value as their rank. For example in Figure 3.1 the first process P0 succeeds in creating the segment, assigns himself master and initializes the counter. The second process P1 then fails to create the segment because the first one already did. It then increments the counters value and assigns himself the current value as rank.

3.2 Data Transport

The data transport is the core of our backend, without it no data migration would be possible. To perform the data transport over shared memory we implemented two versions: a 2 copy and a 1 copy version. A 2 copy version is needed because LAIK allows applications to directly provide memory. Directly provided memory is memory that is allocated by the application and not by LAIKs allocator. If such a memory space sends data, the receiving party can not directly copy out of the segment because it is not a shared one. In that case a 2 copy version the only viable option. However, the 2 copy version only serves as a fallback when a 1 copy transport is not possible.

3.2.1 2 Copy Transport

In order to accommodate the transport of data from directly provided memory we implemented a 2 copy data transport. This means that any data sent with this method is copied two times during transport as can be seen in Figure 3.2. The sending process creates a new shared memory segment and attaches to it. The process then copies the relevant segment to the shared segment. The receiving process opens the shared segment and copies the data to his own segment. After that the shared memory segment is destroyed.

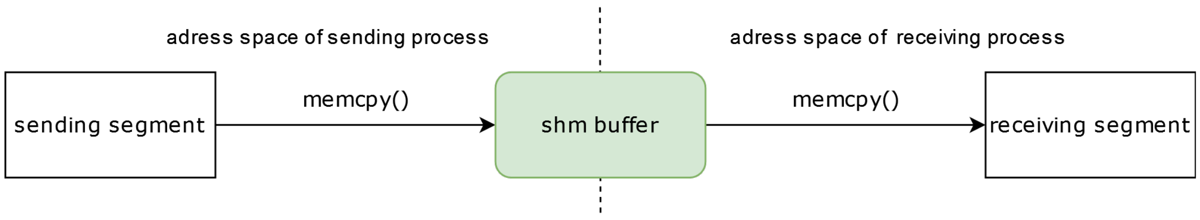
adress space of sending process
adress space of receiving process
sending segment
memcpy()
shm buffer
memcpy()
receiving segment

Figure 3.2: Schematic of the 2 copy transport

3.2.2 1 Copy Transport

As the copying of the segment is the most resource intense operation of a data transmission operation, it makes sense to copy only once. We therefore design a 1 copy version of our data transport over shared memory. In contrast to the 2 copy version, both segments are shared segments which allows the sending and receiving processes to copy directly from one to the other. For this the sending process marks the segment as ready so that the receiving process can copy it directly into his own memory. The receiving process waits until the segment he wants to copy from is marked as ready and then copies it into his own segment. As we can be seen in Figure 3.3 we can save one `memcpy()` call this way, which should increase the throughput of our backend.

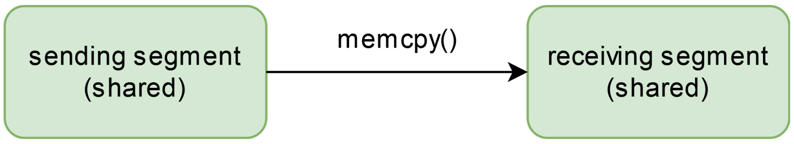

Figure 3.3: Schematic of the 1 copy transport

3.3 Standalone Version

The standalone version we developed only uses shared memory to fulfil its tasks and can therefore only run on a single node. This version is not able to run across nodes as the shared memory of one node is not accessible from another node. The standalone version implements all necessary backend functions and is therefore able to run on its own. The methods the standalone backend implements from the backend interface are similar to those of the MPI backend. Only the data transport was replaced by our shared memory based transport. This backend works like a normal backend, it prepares, executes and cleans up his action sequences by itself. The backend can therefore be used like any other backend as long as the application runs on a single node.

3.4 Secondary Backend Version

The secondary backend version was created to be able to run on multiple nodes. In the example in Figure 3.4 we have 2 clusters with 2 processes each. The 2 pairs of processes on the same node would be able to communicate with each other over shared memory but not with the other pair, as it is not possible to use shared memory for inter node communication. The shared memory backend therefore needs to be combined with another communication backend to be able to deploy an application on multiple nodes. The shared memory backend should therefore only support the other communication backend with providing efficient intra node communication. The backend that uses the shared memory backend will be called primary backend and the shared memory backend will be called secondary backend. The secondary backends tasks get reduced to replacing normal actions with shared memory actions where possible and executing those shared memory actions. The primary backend will function as a normal backend which delegates its actions to shared memory where possible. The secondary backend should be easy to integrate into new backends as most backends should profit from its support.

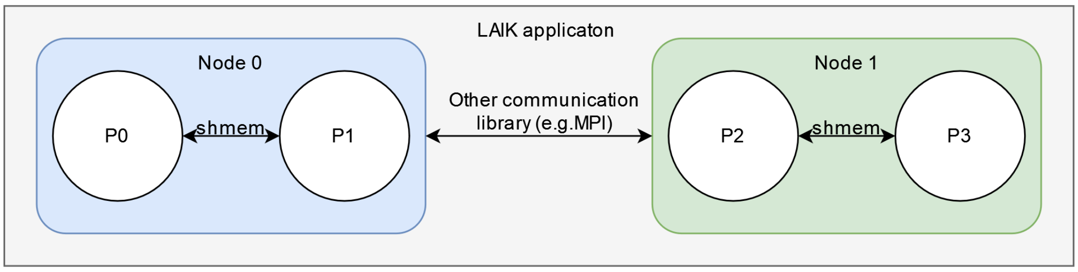

Figure 3.4: Exemplary deployment of an application with 4 processes on 2 nodes

3.4.1 Integration into Other Backends

The secondary backend version is not able to run standalone, it therefore needs to be integrated into another backend to run. As most backends should profit from the support of the shared memory backend, the backend should be able to be easily integrated into any other LAIK backend. A backend should therefore need as few modifications as possible to integrate the secondary version as its secondary backend. The primary backend should only have to call one initialization method to initialize the shared memory backend. After that the primary backend should be able to optimize and execute all its action sequences with the help of the secondary backend. After an action sequence is optimised with shared memory all actions between processes on the same node should be replaced with shared memory based actions. During execution of an action sequence the primary backend will then execute all actions which involve processes on different nodes. Those actions will then be executed with the primary's communication library. All other actions will be delegated to the secondary backend to be executed there.

3.4.2 Initialization of multiple Shared Memory Clusters

As the secondary backend versions purpose is to use shared memory where possible when deploying an application across nodes, we have to manage different shared memory clusters. A shared memory cluster are all processes of an application which can communicate with each other via shared memory. In the example seen in Figure 3.1 we would have 2 clusters with 2 processes each ({0, 1}, {2, 3}). If we would now initialize all four processes with the method presented section 3.1 we would get 2 clusters, but the two clusters would not be aware of each other. The secondary backend version will therefore use the primary backends communication library to get aware of other clusters. After initialization each process should be aware of all other processes and their cluster affiliation.

3.4.3 Action Substitution

To use the secondary backend, the primary backend must substitute its generic actions with shared memory based actions during preparation. The secondary backend must provide an optimization pass for that purpose which the primary backend can use during preparation of an action sequence. This optimization pass should iterate over every action and check whether the involved processes are on the same node. If so, the action can and should be replaced by its corresponding shared memory action. The returned action sequence can then be further optimized by the primary backend to change the remaining generic actions.

4 Implementation

The following chapter describes how the design discussed in chapter 3 was implemented to function as a LAIK backend. At first we will discuss the overall structure of the implemented backend. We will then explain how the secondary backend version can be integrated into other backends. The rest of the chapter will explain in detail how the previously discussed core functionality of the backend was implemented.

4.1 Structure

As we implement both a LAIK backend as well as the corresponding shared memory based communication library we split our backends functionality up into 2 files. The first one implements the backend functionality and the second one the communication library for the backend. The backend file (`backend-shmem.c`) will contain both the standalone version and the secondary backend version. The standalone version has a similar structure to the MPI backend but it replaces all MPI library calls with equivalent ones to the shared memory library. The secondary version part of this file provides a customized initialization routine for the secondary backend we will further discuss in subsection 4.2.2, an optimization pass to replace normal methods with shared memory based methods, an execution method to execute the shared memory actions and a finalize method which closes all open shared memory segments. Those methods provide an interface which a developer can use to provide shared memory support for his primary backend. The file providing the shared memory based communication library (`shmem.c`) implements the initialization routines and the data transport.

4.1.1 Integration of the Secondary Backend Version in a Primary Backend

As mentioned, the secondary backend version should be easily integratable into other backends. To integrate the shared memory backend into a new backend a developer needs to amend the initialization, prepare and finalize methods of his backend.

The initialization method of the primary backend must call the modified secondary backend initialization routine to initialize the secondary backend. As we discussed in subsection 3.4.2, the secondary initialization method needs to communicate with the primary backends communication library to discover processes on other nodes. The primary backend therefore has to provide a send and receive function using its communication library. Those two functions get passed as function pointers to the secondary backends initialization method.

The primary's prepare method must be extended to also include the optimization pass of the shared memory backend. The added optimization pass will replace generic actions with shared memory actions where possible. As the added optimization pass introduces new shared memory backend specific methods, the primary's exec function needs be able to execute those.

The exec function must be changed, so that it calls the secondary exec function if it does not know how to execute a function. The secondary exec function will then execute the action and return a success or, if it can not execute it return a failure.

To close all open shared memory segments, the primary's finalize must also calls the secondary backends finalize method.

4.2 Initialization

This section will first discuss how the general initialization mechanism was implemented. After that we will explain the initializations differences between the standalone and the secondary backend version.

As we discussed in section 3.1 we need to let every process try to create an exclusive shared memory segment to assign one process the rank of master. The processes will try to create the segment by calling `shmget()` with a predefined value as `key` and the flags `IPC_CREAT` and `IPC_EXCL` set. That way only the first process will succeed and all others will receive -1 from `shmget()`. The master will then initialize a counter in the shared memory segment by setting an atomic integer to 0. The other processes which were not successful in creating the exclusive segment will then call `shmget()` again but without the `IPC_EXCL` flag set. They will then increment the atomic counter and assign themselves the counters value as rank.

After the ranks are assigned, each process will create a shared memory segment which will be needed for the data transport. We will discuss those segments in more detail in section 4.3.

4.2.1 Standalone Initialization

As the standalone version can only be deployed on one node, the environment variable `LAIK_SIZE` can be used to determine the size of the shared memory cluster. `LAIK_SIZE` represents the number of processes of a running LAIK application. The master will keep the exclusive shared memory segment open as long as the counters value is below `LAIK_SIZE`. After that every process will have gotten his rank and the initialization process can be ended.

4.2.2 Secondary Initialization

As the secondary backend version can be deployed across multiple nodes, there can be multiple clusters. After initialization each process needs to know which processes are in his cluster and what other processes exist. As `LAIK_SIZE` only depicts the number of all processes of a LAIK application it can not be used to determine the cluster size.

To compute the clusters we therefore introduce the concepts of primary and secondary ranks and colours. A processes primary rank is the rank he has in the primary backends communication library. The secondary rank is the rank the process got assigned himself during the secondary backend initialization. Unlike the rank of the secondary backend, the rank of the primary backend is unique. The colour of a process is the primary rank of the clusters master. The colour therefore serves as an identifier for a cluster. According to our definition, the processes seen in the exemplary LAIK application from Figure 4.1 result in the ranks and colours seen in Table 4.1.

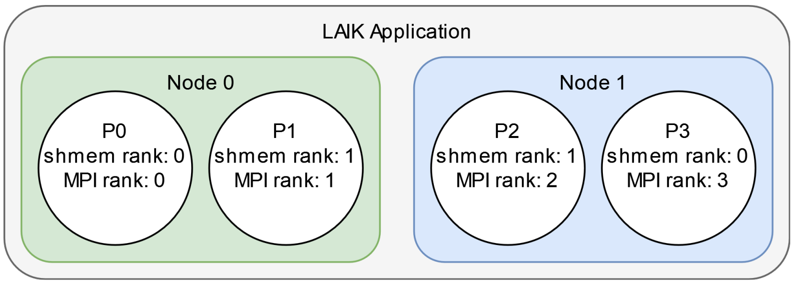

Figure 4.1: Ranks of a LAIK application with 4 processes deployed on 2 nodes

	P0	P1	P2	P3
primary rank	0	1	2	3
secondary rank	0	1	1	0
color	0	0	3	3

Table 4.1: The ranks and colours of the LAIK application from Figure 4.1.

After each process got its rank from the atomic counter, they will then use the provided send and receive methods from the primary backend to send their secondary rank to the process with the primary rank 0. This process will then accumulate all the nodes secondary ranks. It will then compute the colour of every process and the sizes of all clusters and distribute that information to all the other nodes. After that is done, every process knows which processes are in his group and what other processes exist.

4.3 Data Transport

This section explains how the data transport was implemented in `shmem.c`. We will first explain how we decide which variant executes a given data migration. After that, we will elaborate on the implementation of our 1 and 2 copy data transport versions.

To provide information about the shared memory segment which is about to be sent, each process has a semaphore to synchronize its sending operations and its own shared memory segment which hosts a `metaInfos` struct. The `metaInfos` struct contains the receiving processes secondary rank, the number of elements to be sent, the shmid of the shared memory segment which is to be copied from as well as the offset the pointer has from the start of the shared memory segment. The semaphore will be locked by the process it belongs to until he has completed a sending operation and waits for the receiving process to receive the data.

After the receiving process received the segment, the sending process will lock the semaphore again.

As discussed in section 3.2 the 2 copy version should only be used if an application directly provided memory. All other `Laik_Data` was allocated via a `Laik_Allocator` and is therefore shared memory. Every time a shared memory segment is allocated with via an allocator, the shared memory library saves the pointer to the segment, its shmid and its size. This allows us to determine whether a given pointer points to a shared memory segment or not. When a shared memory action gets executed, the one copy methods get called. The 1 copy send method will then try to determine the shmid of the segment the given pointer points to. If the given pointer to does not lie in any of the previously allocated shared memory segments, the sending process will set the shmid in the `metaInfos` struct to -1 and call the 2 copy version. After the receive call reads the shmid out of the meta info segment it will check whether a valid shmid was given. If the given shmid is -1 it will call the 2 copy version.

4.3.1 2 Copy Transport

As the 2 copy transport needs to create its own shared memory segment to copy the data to, it needs to generate a unique key for each transaction. To generate this key we hash the secondary rank of the sender add the recipients rank and hash the sum again (`key = hash(rank_recipient + hash(rank_sender))`). Those keys are unique for our purposes, since we can only have one send and receive operation between two processes at a time. With the generated key the two processes create and open a shared memory segment which equals the buffers size. The sending process will then copy his data into the segment and update his meta information segment. After that he will unlock the semaphore for his segment. The receiving process will then lock the semaphore and copy the data from the segment to its buffer. After copying the data, the receiving process will free the semaphore which the sending process will immediately lock again.

4.3.2 1 Copy Transport

The 1 copy transport is used if the given buffer pointer points to a shared memory segment. The 1 copy send method will update the meta info segments information with the data of the new segment and free the semaphore.

As mentioned in subsection 2.2.1 shared memory is mapped to each attached processes address space. This means a pointers adress can differ between processes despite them pointing to the same element of a shared memory segment. We therefore need to compute the pointer from the shared segments id and the offset which are accessible over the meta Information segment. The offset is the difference between the pointers address and the shared segments address. To compute the actual pointer we first attach the shared memory segment by its id. The offset must then be added to the returned pointer to compute the correct pointer.

After the sending operation unlocked the semaphore, the receiver is able to lock the semaphore and access the sending processes meta information segment. The receiving process will then read shmid, offset and the sending operations number of elements. The receiver will then check that his receiving buffer is big enough for all the data the sending process wants to send. It will then copy the data directly from the sending processes buffer to its buffer. The semaphore will then be unlocked by the receiving process and the sending process will lock the semaphore again.

4.4 Action Substitution

In this section we will elaborate on how the generic actions get replaced by shared memory actions.

Every action has a type which is an unique integer to identify the action. We defined our own action types four our shared memory actions. To replace the generic actions we need to change their type to the corresponding shared memory action type.

As mentioned in subsection 4.1.1 `backend-shmem.c` provides an optimization pass with the signature `bool laik_aseq_replaceWithShmemCalls(Laik_ActionSeq *as)`. This method replaces the generic actions with shared memory actions where possible by iterating over the actions and checking whether they can be replaced.

A generic call can be replaced when all involved processes are in the same cluster. This can be determined by looking at the processes colour which identifies the cluster they are in. If all processes have the same colour, the action type will be changed to the corresponding shared memory action type and the primary ranks stored in the action will be replaced by the corresponding secondary rank. This ensures that the shared memory library can process the actions correctly.

4.5 Action Execution

We discussed in subsection 4.1.1 that the primary backend calls the secondary backend if it can not execute an action because its type is unknown to the primary backend. In that case the method `bool laik_shmem_secondary_exec(Laik_ActionSeq *as, Laik_Action *a)` is called to execute the action. This method checks whether the given actions type is known to the secondary backend and executes it if it knows the type. In that case the method returns true, if it does not know the action it returns false. This is done to let the primary backend know whether the action was executed by the secondary or if it should alert LAIK due to an unknown action.

5 Performance Analysis

In this chapter we will test our backends performance and compare it to other backends. First we will explain the setup used for testing. We then go on to test and analyse the throughput of our backend with LAIKs ping pong example application.

5.1 Setup

To test our backend the Leibnitz Rechenzetrum (LRZ) provided us with access to its testing environment BEAST. We tested our setup on the ice cluster of the LRZ BEAST. This cluster consists of two Intel Xeon Platinum 8360Y CPUs running on 2.40GHz. Every CPU has 2 sockets with 36 cores each.

We tested our secondary backend version integrated into an MPI backend against two versions of the LAIK MPI backend. Th first one used a shared memory fabric and the second one a tcp fabric. LAIK was compiled with gcc 11.2.0 and intel mpi version 2021.4.0.

The MPI backend was run with the environment variable `I_MPI_ASYNC_PROGRESS` set to 1. This enables asynchronous progress threads which achieve better communication/computation overlapping[4]. When using the shared memory fabric we set `I_MPI_SHM` to `icx` to choose the shared memory transport solution for Intel Xeon processors with Ice Lake microarchitecture [4]. We also set the `I_MPI_OFI_PROVIDER` to shm to choose shared memory for transportation. The `I_MPI_SHM_CMA_THRESHOLD` variable was set to 2000000000 to ensure that shared memory was used despite of the large sizes involved. When we used the tcp fabric we set `I_MPI_OFI_PROVIDER` to tcp to use tcp for transportation. We also set `FI_LOG_LEVEL` to debug to be able to confirm through the debug information, that the correct fabric was used[4].

5.2 Ping Pong

The ping pong example is one of the examples provided in the LAIK library. It moves an array of doubles repeatedly between processes. We can set the arrays size as well as the repetitions over command line arguments. Ping pong performs the data migrations between pairs of processes with even and odd rank ((1,2), (2,3), ...).

We used the Ping Pong example to measure and compare the throughput of the tested backends. The throughput in GB/s is a good indicator of a backends performance as it measures how fast a backend can transport data between processes. We tested the ping pong example with an array size of 100 million doubles and 10 iterations.

Throughput

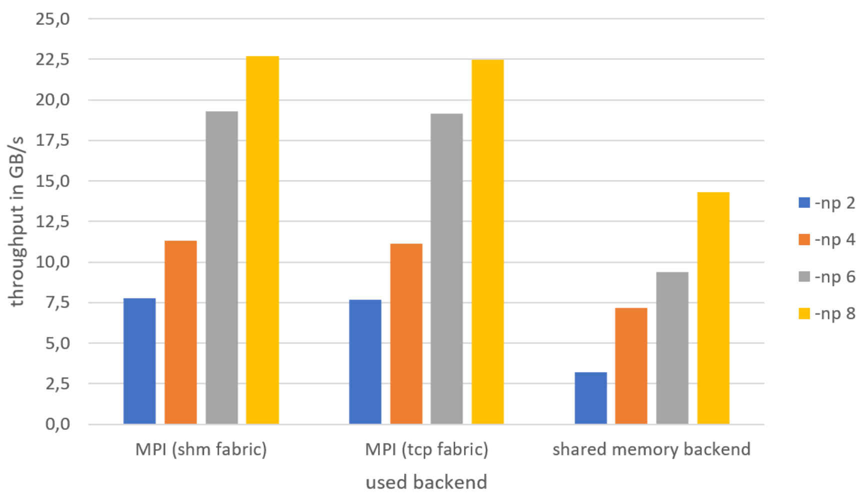

Figure 5.1: Throughput of the different backends with different group sizes.

In our first test we measured the throughput of the backends at different process numbers. As we can see in Figure 5.1, the backends throughput increases with the number of processes. This was to be expected as more processes are able to send more data. The two versions of the MPI backend nearly achieve similar performance regardless of the used fabric. Only our shared memory backend is lacking in performance which is unexpected. Through profiling we could find out, that the linked `memcpy()` call used in the data migration was slow. The call alone took longer than a `MPI_Send()` call for an array of equal size. This is due to a slow implementation of `memcpy()` being linked to our backend during compilation. The performance of the backend would have likely been better if a more performant `memcpy()` call was linked to it. Unfortunately, we could not complete this within the scope of our work.

Influence of the Deployment on Performance

We also tested how the throughput of our backend changes depending on the deployment of our processes. We deployed the ping pong example with two processes on different processors to see the influence the deployment had on the performance. The processes were pinned to the processors by using the environment variable `I_MPI_PIN_PROCESSOR_LIST`.

First we tested how the throughput differs between two processes deployed on the same socket and two processes being deployed on different sockets. One would expect the 2 processes on the same socket to be faster. Our tests confirmed that two processes on the same socket achieve around 0.5GB/s more throughput than the processes on different sockets (see Figure 5.2).

The influence of NUMA domains on the throughput was also examined. Processes in the same NUMA domain are expected to achieve a higher throughput. As we can see in

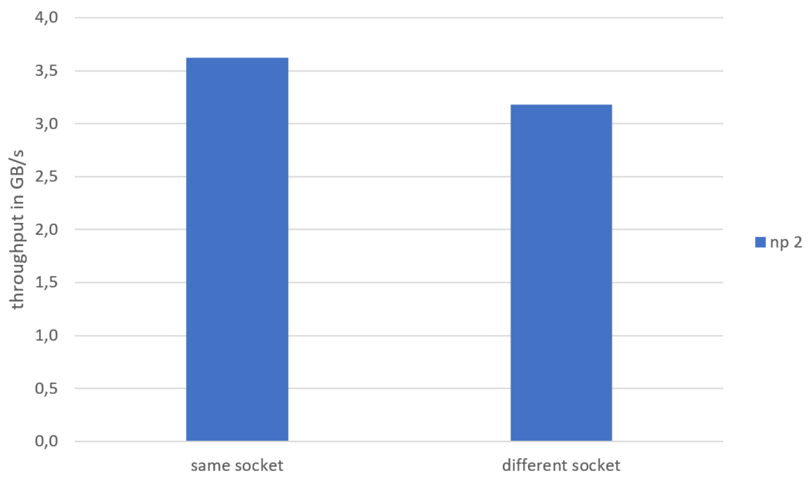

Figure 5.2: Throughput of ping pong with two processes on the same and on different sockets

Figure 5.3 the processes in the same NUMA domain achieved a higher throughput than those in different NUMA domains.

Comparison between 2 and 1 Copy Data Transport

Finally, we compared the 2 copy data transport against the 1 copy data transport to see how big the performance difference between the 2 versions is. It is expected, that the 1 copy version achieves a higher throughput since it uses one `memcpy()` operation less than the 2 copy version. Figure 5.4 shows that the 1 copy versions throughput is more than twice as high as the throughput of the 2 copy version. Besides the additional `memcpy()` call, the 2 copy version also has to create and destroy a shared memory segment for each data migration which adds aditional overhead. It is therefore no surprise that the 1 copy versions performance is far better.

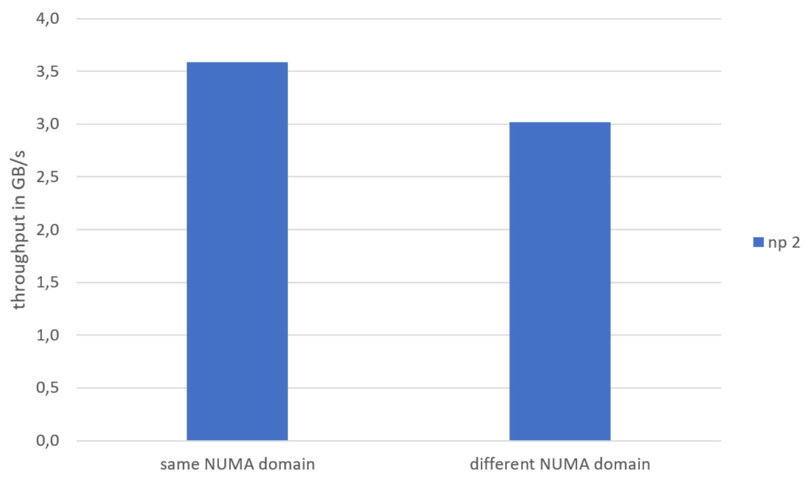

Figure 5.3: Throughput of ping pong with two processes in the same and in different NUMA domains

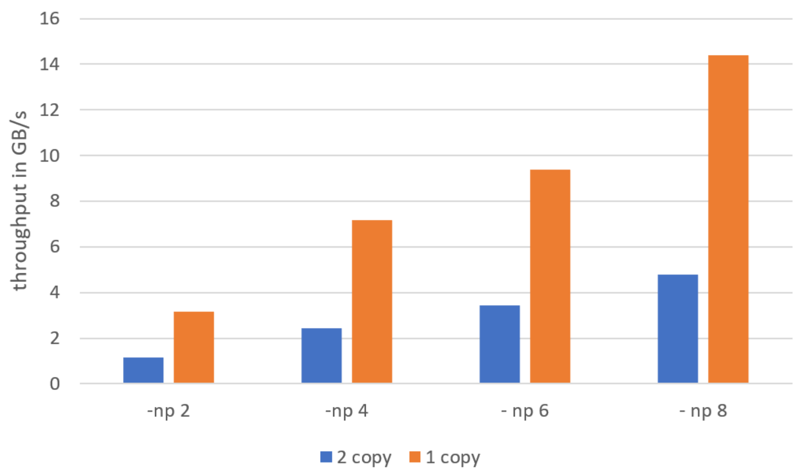

Figure 5.4: Comparison between the throughput of the 2 copy and the 1 copy data transport

6 Summary and Outlook

In this work we presented the designs of two shared memory based backends for the LAIK library. We introduced the reader to the core functionality of LAIK and the IPC mechanisms used in the backend. After introducing the necessary background we explained the design of our shared memory backends. We then explained how we implemented both backends and how a backend can use the functionality of the secondary backend version to execute its data migration on shared memory. After that we tested our backends performance and compared it against other LAIK backends.

We hope that this work can serve as the basis for future work. It would be interesting to see the performance of the backend with an efficient implementation of memcopy linked to it. Besides that, the approach of a 0 copy data transport where all processes work on one synchronized shared memory segment seems to be a promising approach to further increase LAIKs performance.

List of Figures

List of Tables

Bibliography

[1] C. T. Josef Weidendorfer Dai Yang. “LAIK: A Library for Fault Tolerant Distribution of Global Data for Parallel Applications”. en. In: *Konferenzband des PARS’17 Workshops*. Gesellschaft der Informatik. Hagen, 2017, p. 10.

[2] J. Wolf. *Linux-UNIX-Programmierung, Kapitel 9: Interprozesskommunitkation*. Rheinwerk Verlag, 2006.

[3] K. Johri. *System V Shared Memory in Linux*. https://www.softprayog.in/programming/interprocess-communication-using-system-v-shared-memory-in-linux (accessed: 08.09.2022). June 2022.

[4] *Intel® MPI Library Developer Reference for Linux* OS*. https://www.intel.com/content/dam/develop/exter devref-linux.pdf (accessed: 10.09.2022).