DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Transferring Hardware-related Information in sys-sage Across Processes and HPC Nodes

Finn Romaneessen

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Transferring Hardware-related Information in sys-sage Across Processes and HPC Nodes

Übertragung von Hardware-Informationen in sys-sage Bibliothek Zwischen Prozessen und HPC Knoten

Author:	Finn Romaneessen
Supervisor:	Prof. Dr. Martin Schulz
Advisor:	Stepan Vanecek
Submission Date:	May 15th, 2024

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

Munich, May 15th, 2024 Finn Romaneessen

Acknowledgments

Abstract

In this thesis, a file backed shared memory system was implemented, which can be used to share *sys-sage* topologies between processes and nodes in HPC systems. The shared memory regions used to transfer the topologies between the processes are implemented with memory mapped files created using the `mmap()` syscall.

To share a *sys-sage* topology, the exporting process recursively copies all components, including attribs and DataPaths, of the chosen topology into the created shared memory region. Since the virtual memory addresses inside the shared region, all pointers within the topology must be translated to offset based pointers.

All importing processes can then recreate the topology in their local memory, translating the offset based pointers back into regular pointers and recreating the component tree structure.

The performance of the implemented shared memory system is tested using multiple example topologies with different amounts of components, attribs and DataPaths.

Additionally, some limitations of the presented implementation are discussed and suggestions for future improvements are made.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**
1.1 Motivation 2
1.2 Related Work 2

2 Sys-Sage **4**

3 Approach **6**
3.1 Capabilities 7

4 Implementation **8**
4.1 Shared Memory 8
4.2 Components 10
4.2.1 Exporting Components 11
4.2.2 Copying Vectors 12
4.2.3 Importing Components 12
4.3 Attribs 13
4.3.1 Packing and Exporting Attribs 13
4.3.2 Unpacking and Importing Attribs 16
4.4 DataPaths 16
4.4.1 Exporting DataPaths 17
4.4.2 Importing DataPaths 18

5 Performance **19**
5.1 Execution 19
5.2 Basic Component Tree 20
5.3 Attribs 20
5.4 DataPaths 21

6 Conclusion and Future Work **23**

List of Figures **24**

Listings **25**

Bibliography **26**

1 Introduction

In high-performance computing (HPC), powerful, highly parallel computers are used to solve various technical challenges in all major scientific areas or any other applications processing large amounts of data or handling large-scale computations. [CHC12]

HPC clusters are complex, heterogeneous systems, consisting of many nodes, each with an ever increasing number of cores. The hierarchical composition of an HPC systems hardware architecture and its individual components is referred to as hardware *topology*. [Aa18]

The Message Passing Interface (MPI) is commonly used in HPC systems and other parallel machines to communicate between processes and nodes within the cluster. This allows each node in a HPC system to execute independent programs in its own local memory, while still cooperating and communicating with the entire cluster as needed. [Nie16]

Due to the heterogeneous composition of HPC systems and the high degrees of parallelization utilized, Non-Uniform Memory Access (NUMA) is often used in HPC clusters to achieve more efficient memory access. NUMA systems use a distributed memory hierarchy to allow cores faster access to local memory regions, whereas accessing non-local memory can cause severe performance decreases. [GM11]

As memory bandwidth and cache usage have tremendous impact on the overall performance of highly parallelized HPC systems, making full use of the hardware topology to schedule processes accordingly is crucial in HPC clusters. [Bro+10b]

Processes working closely together will often benefit from sharing a cache to utilize local memory as much as possible, whereas independent, memory intensive processes could be better scheduled onto separate processors so as not to limit their memory bandwidth and available cache storage. [Bro+10a]

Making use of the hardware topology to create an efficient process-mapping has the added advantage of reducing the overall communication costs as related processes will be physically close within the cluster. [Aa18]

Many tools, such as *Portable Hardware Locality (hwloc)* [Bro+b], already exist to gather information about the hardware topology and make it available for these purposes. Hwloc is a software library that represents hardware resources such as cores or caches in a hierarchical tree structure to store easily accessible and versatile information about the hardware topology of HPC systems. [Bro+10a]

Hwloc collects the entire topology information at startup and makes the static information available to the application. [Bro+10a]

While this approach offers better performance due to the low overhead at runtime, the topology tree can't be adapted to dynamically changing factors at runtime.

Sys-sage [Van] is a library that extends hwlocs functionality to include dynamically changing hardware information and allow representation of arbitrary custom data. Since sys-sage is fully compatible with hwloc, users can easily use hwloc to initialize their topology data and complement it with custom data as needed.

1.1 Motivation

As mentioned above, using accurate and current information on the hardware topology of a HPC system has many applications in optimizing the overall performance of individual programs within the cluster.

Creating a detailed topology in sys-sage can cost significant resources during setup, as many custom attributes, components and DataPaths might need to be created.

Since the hardware topologies of different processes within a node are usually very similar and different nodes within a cluster often have significant overlap in their topologies as well, creating new topologies in each process and node would cause unnecessary overhead.

Instead, sharing entire topologies or specific component subtrees across processes and nodes of HPC systems could be used to reduce redundant creations of sys-sage topologies.

The objective of this thesis is to add a file backed shared memory implementation to sys-sage, which allows user to share entire topologies or select subtrees with other processes or nodes.

1.2 Related Work

Sharing data between processes and nodes using memory mapped files is not a new concept. In fact, it is widely used across many different sectors many different applications and sectors such as database management systems and data mining. [CLP22] [Rao+08]

This section highlights different approaches to sharing large amounts of data between processes and nodes using memory mapped files as implemented by different libraries.

hwloc

As mentioned above, portable hardware locality (hwloc) is the software library on which sys-sage bases its functionality and use cases. Much like sys-sage, hwloc is used to manage information about a systems hardware topology to improve the performance of HPC clusters.

Hwloc offers users the option to share topologies between processes. To do this, the exporting process must use the `hwloc_shmem_topology_write()` function, which copies the topology into a previously created memory mapped file. The duplicated topology can then be *adopted* by importing processes using `hwloc_shmem_topology_adopt()`. The user can then use the adopted topology as usual, however, it can not be modified anymore. [Bro+a]

The hwloc shared memory implementation uses file backed memory mappings with identical virtual memory addresses in each process. To achieve this, hwloc requires the user to find a memory region of sufficient size that is available in all processes.

This approach has the advantage that pointers inside the memory mapped region will work as usual, which means the shared topology can be used directly from within the shared memory section without having to be copied into the local memory of importing processes.

On the other hand, this approach puts more responsibility and effort on the users side, as finding a sufficiently large memory region might be difficult to achieve. Additionally, at the time of exporting the topology, it might not be fully known which processes will be adopting the shared topology during its lifetime, so the chosen memory section might not be available in all importing processes, which can cause expensive reallocations.

While this approach works well for hwloc, as topologies are mostly static in nature, the more dynamic needs of sys-sage topologies call for a shared memory solution that allows importing processes to change the topology as needed.

sharedstructures

Sharedstructures [Mic] is a small software library available in C++ and Python that can be used to store generic data structures such as hash tables or queues in file backed shared memory regions.

Contrary to hwlocs implementation, sharedstructures uses offset based pointers within the shared memory region, as the memory sections are resizable and thus might move during their lifetimes.

Sharedstructures provides custom allocators to manage the data structures in the shared memory region. These allocators are used to reserve memory space within the shared region and can be chosen based on the memory usage profile of the program.

2 Sys-Sage

Sys-sage is a software library created to collect, represent and provide data on the hardware topology of HPC systems. It is designed to extend on the existing hwloc library and provide a more versatile and dynamic use-case.

Since a lot of the hardware related data necessary for complex tasks such as thread scheduling or memory management dynamically changes during the execution of HPC computations, the hwloc approach of providing static topology information collected on startup is often not sufficient. Gathering the entire dataset at startup makes it difficult to incorporate user-defined data points into the topology and react to the current state of the system such as measured bandwidth or memory usage.

Sys-sage is designed specifically to enable users to create highly customized and dynamically changeable hardware topologies and build on top of existing hwloc topologies.

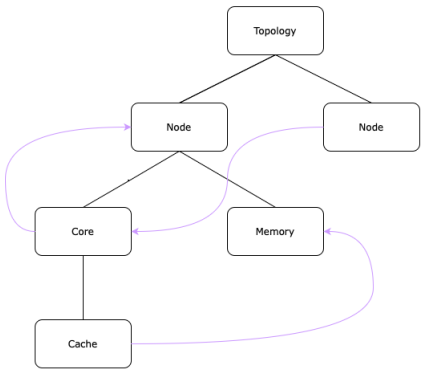

Figure 2.1: Basic Component Tree with DataPaths

The hierarchical structure of the hardware topology is represented in sys-sage as a tree of *components*. Each component corresponds to a specific part of the hardware, such as a *cache*, *core* or *node*. Depending on the type of component, further information on the underlying hardware can be added to the component, for example the size of a cache. Additionally, arbitrary attributes can be attached to components to provide further context or add dynamically updated values to the topology.

Beside the component tree, the *DataPath* graph adds additional information to the topology. DataPaths are connect two arbitrary components and are used to represent non-hierarchical relationships between components. Much like components, DataPaths can have custom attributes to add additional data to component relationships.

Figure 2.1 shows a simple example of a topology consisting of two nodes including a few components and DataPaths.

3 Approach

The goal of this thesis is to integrate a file backed shared memory system into the existing sys-sage functionality, allowing the transfer of topologies across processes and nodes within the HPC cluster.

This can be achieved, by creating memory mapped files using the `mmap()` syscall and copying the topology or component subtree into the shared memory region.

As described in section 1.2, there are different approaches to implementing shared memory regions using `mmap()`. For the purposes of this thesis, an approach similar to the *sharedstructures* implementation was chosen, meaning the implementation uses offset based pointers rather than enforcing identical virtual memory addresses across all processes.

This design was chosen, as it cannot necessarily be guaranteed that a sufficiently large memory section can be found for the shared topology, which could negatively affect the reliability of the system. Additionally, at the time of exporting the topology into the shared memory region, it is not always known, which processes will need to import the topology.

However, using offset based pointers also means that the topology will have to be deconstructed before being copied into the shared memory region, as some internal components, such as vectors or maps, rely on regular pointers that will not work within the shared region.

Due to the topology being deconstructed before being copied to the shared region, it cannot be used directly in the shared memory block and needs to be reconstructed in the local memory of the importing process before being usable again.

This unfortunately makes it impossible for multiple processes to share a topology, each process being able to update the component tree as needed. Instead, processes can only send snapshots of a topology to another process over the shared memory region. After the topology has been recreated in the importing processes local memory, each process can make updates to their own version of the topology, however, updates are not shared between processes. If a shared topology needs to be updated across all processes, the updated topology needs to be shared entirely anew.

3.1 Capabilities

In essence, the shared memory implementation of this thesis consists of two main functions.

With `export_topology()` the exporting process can create a new shared memory region in a provided file location and copy a chosen topology or component subtree including DataPaths and attribs into the newly created file backed memory region. The `export_topology()` function also allows the user to choose, which of the topologies attribs will be exported into the shared memory region.

Any process wishing to use the shared topology can then use `import_topology()` to recreate the topology in its local memory.

4 Implementation

The part of the sys-sage library implemented in this thesis enables users to share component subtrees or whole topologies between processes of a compute node by using shared memory regions.

To achieve this, all components of the given subtree, including its attributes and all DataPaths, are copied into a memory region shared between the involved processes. The component tree and DataPath graph are then recreated in the memory of the receiving process.

4.1 Shared Memory

The data sharing aspect of the implementation is realized using shared memory regions. These regions are created by opening files and mapping them using `mmap()`.

`mmap()` is a syscall that creates file backed memory mappings in the program's virtual address space that can be opened by multiple processes at once. The mapped file can then be used just like any regular memory location. [CLP22]

Using the *MAP_FIXED* flag when creating a `mmap()` backed memory location will guarantee the virtual memory addresses to be equal across processes. However, if the necessary memory location is not available in the current process, the mapping will fail, which could potentially have a major impact on the reliability of the library, depending on the total available memory and its current utilization. [Quote manpage mmap]

Consequently, the *MAP_FIXED* flag is not used for the purposes of this thesis to achieve higher reliability when sharing component trees between processes. As a result, the virtual memory addresses of the shared regions are not identical across processes.

Due to this, sharing pointers to addresses in the shared memory region between processes will not work, as the as the referenced location will have a different address in another process. Instead, offset based pointers have to be used to reference shared memory locations.

In the sys-sage shared memory implementation, all offsets for pointers are calculated relative to the top of the shared region. While this might not be possible for more general uses of offset pointers, as the start of the memory location might not always be known, it is practical for the particular use-case of this thesis, since memory regions

are always handled as a whole and importing only parts of a shared component tree is not supported.

Calculating the offsets based on a shared, fixed location has the advantage that the offset pointers can be used more similarly to regular pointers and don't need to be recalculated when shared. This means the location of components or DataPaths within the shared memory region can be compared or referenced without ambiguity or confusion about the base of the offset.

The lifetime of the shared memory region is handled by a *SharedMemory* object, as shown in Listing 4.1.

```
1  class SharedMemory {
2    public:
3      void* mem;
4      char* cur;
5      size_t size;
6      ...
7
8      SharedMemory(std::string path, size_t size);
9      SharedMemory(std::string path);
10     ~SharedMemory() { munmap(mem, size); }
11
12   private:
13     std::string path;
14 };
```

Listing 4.1: SharedMemory Class

Apart from the *path* and *size* size variables, which are used mainly in the creation and destruction of the shared memory region, the *SharedMemory* class consists of two pointers, *mem* and *cur*. The *mem* pointer always points to the top of the memory region, whereas the *cur* pointer marks the current location to write or read from while importing or exporting a topology.

When a shared memory region is first created by the process sharing the topology, the path to the file used in the mapping as well as the total size needed have to be known. The allocated size is then written to the start of the mapped file, to be read by other processes when importing the topology. The file-backed memory region can then be used to export the topology until the *SharedMemory* destructor is called and the file is unmapped using `munmap()`.

The process importing the topology then uses the constructor in line 9 of Listing 4.1, which opens and maps the previously created file, reads the total size of the data as written by the first process and then remaps the file to that size. This has the advantage

that the total size of the shared topology is always known to the importing process, without having to be provided separately. To share a topology, only the path to the memory mapped file needs to be provided, all other information can be read from the file, which simplifies the API and makes it easier for users to share topologies without much inter-process communication needed.

[Size Calculation]

4.2 Components

Topologies consist mainly of *components*, which are organized as a hierarchical tree structure. Each component represents a certain part of the hardware such as a CPU or cache. Depending on the type of hardware the component represents, there are different subclasses of Components that can store specific hardware information such as the size of a cache. Although there is no formal requirement, the top of the component tree is usually represented by a component of class *Topology*, a subclass of *Component*, which stores no additional values. Listing 4.2 shows part of the sys-sage component class implementation.

```
1  class Component {
2    public:
3      map<string, void*> attrib;
4
5    protected:
6      int id;
7      string name;
8
9      const int componentType;
10     vector<Component*> children;
11     Component* parent { nullptr };
12     vector<DataPath*> dp_incoming;
13     vector<DataPath*> dp_outgoing;
14 };
```

Listing 4.2: Component Class

As shown in Listing 4.2, the component tree structure is created by the vector of component pointers in line 10 for the children, as well as a component pointer for the parent. Apart from that, the *componentType* variable indicates, which component subclass and therefore which type of hardware is represented by the component. *Id* and *name* store additional information to identify the component and underlying hardware.

The *attrib map* enables users to attach arbitrary data to a component by associating it with a key string. This allows for a high degree of customization, as the user can attach any data and update it dynamically as needed.

The vectors *dp_incoming* and *dp_outgoing* in lines 12 and 13 are used to store pointers to DataPaths associated with the component.

4.2.1 Exporting Components

To export the topology into the shared memory region, the component tree structure including all *attribs* and DataPaths needs to be transformed into one contiguous memory block.

Copying the component tree into the shared memory region is performed as follows:

1. Determining the size of the component based on its *componentType*.
2. Copying the *Component* object into the shared region using `memcpy()` and the size determined previously.
3. Copying the *attrib* map.
4. Copying the *children* vector.
5. Recursively copying the children.

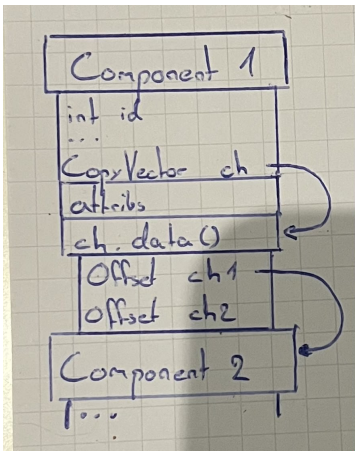

Figure 4.1: Component Tree in Shared Memory

Figure 4.1 illustrates how the component tree is transformed into a single contiguous memory segment. The *Component* object is copied first, followed by its *attribs* and the

data segment of the *children* vector. It is important to note that the component pointers of the *children* vector need to be replaced with the respective offsets of the children in the shared memory region, as the virtual memory addresses will be different in each process.

To achieve this, the children are recursively exported and their offsets written to the *children* vectors data array.

4.2.2 Copying Vectors

Since vectors use a pointer to the contiguous heap memory location storing the underlying data, simply copying the vector object into the shared memory region will not work, as the virtual memory address of the data will be different in other processes. [FIND QUOTE]

To circumvent this issue, the components *children* vector needs to be replaced with an offset based equivalent before being copied. This is done by overwriting the vector object with a *CopyVector* object as shown in Listing 4.3.

```
1 struct CopyVector {
2     size_t offset;
3     size_t size;
4 };
```

Listing 4.3: Component Class

The *CopyVector* struct consists of an *offset* and a *size* variable that are used to reference the underlying data of the original vector. *Offset* stores the offset of the vector's data relative to the start of the shared memory region, while *size* stores the number of elements in the vector.

While recreating the component in the importing process, the *CopyVector* can simply be replaced by a regular vector again, using the copied data by resolving the offset and size.

4.2.3 Importing Components

Since the component tree is transformed into a single contiguous memory block and all regular pointers are replaced by offset based pointers, simply using `memcpy()` to import the topology into the processes private memory will not work.

Importing the component tree into the receiving processes memory is done as follows:

1. Reading the *Components componentType* to determine its type.

2. Recreating the *Component* using the copy constructor of the correct component subclass.
3. Recreating the *attrib* map by inserting all copied key-value pairs.
4. Recursively copying the children and recreating the *children* vector.

To recreate the *children* vector, the offsets of the children stored in the *CopyVector*, as described in subsection 4.2.1, need to replaced with pointers to their respective memory addresses in the importing process.

4.3 Attribs

[Example and default attribs?] Apart from the predefined properties each component subclass uses to represent the underlying hardware, Components and DataPaths have an *attrib* map storing key-value pairs that can be used to add arbitrary data of any size. It is implemented as a `std::map<std::string, void*>`, mapping strings to void pointers.

This allows for a high degree of customization as there are no restrictions to the size or type of data. A string can just as easily be stored as a complex user-defined class. However, the highly versatile use of the *attrib* map also makes it difficult to copy the attrib map, since the size of the data is not necessarily known in advance.

4.3.1 Packing and Exporting Attribs

User-defined attributes can have any size and are not necessarily stored contiguously, which makes copying them into the shared memory region difficult.

A simple example for this problem is an *attrib* storing a linked list. Each item in the list is stored in a different memory location, the *attrib*'s pointer pointing to the first element. The memory utilization of the list also directly depends on its size, which may change during the runtime of the program.

If an *attrib* needs to be exported along with the component, the user needs to supply its size and provide the data as a single contiguous memory block. In the example of the linked list, the list could be transformed into an array containing the same data.

For this purpose the user passes a pointer to function that transforms the *attrib* into a *CopyAttrib* struct as shown in Listing 4.4. *CopyAttribs* consist of the attribs *key*, a *size* variable storing the datas total size in byte and a pointer to the linearized data block.

Figure 4.2 illustrates using the *pack* function supplied by the user to transform an *attrib* pointing to a linked list into a *CopyAttrib* containing the *attribs* key, its data as a contiguous memory block and the datas size in byte.

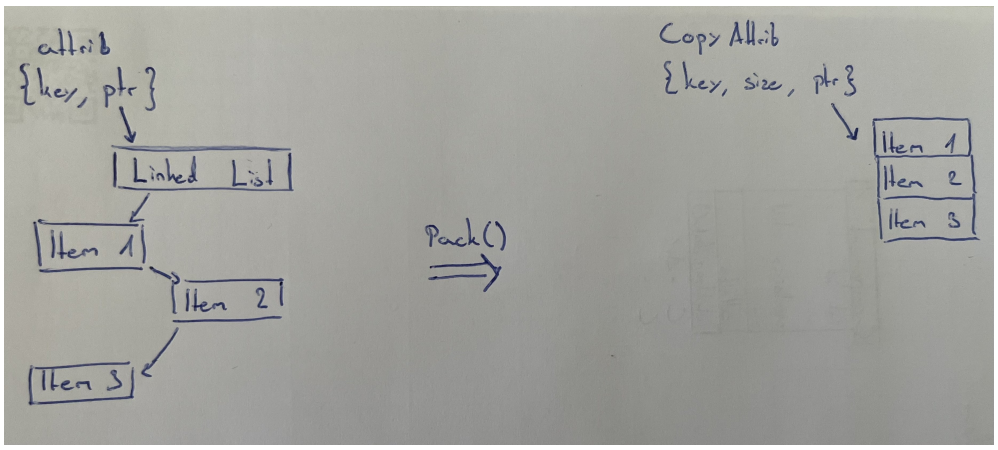

Figure 4.2: Packing a Linked List

```
1 struct CopyAttrib {
2     std::string key;
3     size_t size;
4     void* data;
5 };
```

Listing 4.4: CopyAttrib Struct

Listing 4.5 shows how the user-defined *pack* function for the linked list example could be implemented. It takes a `std::pair<std::string, void*>` containing an attribs key-value pair as a parameter and returns a *CopyAttrib* containing the packed attrib.

In line 2, the *pack* function compares the supplied key with all defined attrib keys to determine, what kind of data the attrib contains. In this case, the key `"EXAMPLE_KEY"` is associated with a linked list of *ints*. A new *int* array of the same size as the list is allocated in line 4 to store the lists elements in a contiguous memory block. In lines 7-9 the array is then filled by iterating over the list and placing it elements in the array one by one. Finally, a *CopyAttrib* consisting of the original key, the arrays size and a pointer to the array is returned in line 11.

Depending on the associated data, not every attrib is necessarily relevant to the receiving process. If an attrib doesn't need to be exported with the rest of the component, the *pack* function can simply return `{key, 0, nullptr}` as shown in line 13 to indicate that the data belonging to a specific key ode not need to be copied.

```
1 CopyAttrib pack(std::pair<std::string, void*> attrib) {
2   if (!attrib.first.compare("EXAMPLE_KEY")) {
```

```
3       std::list<int>* list = (std::list<int>*)attrib.second;
4       int* arr = new int[list->size()];
5       auto i = 0;
6
7       for (const auto& item : *list) {
8           arr[i] = item;
9       }
10
11      return {attrib.first, sizeof(int) * list->size(), arr};
12  }
13  return {key, 0, nullptr};
14 }
```

Listing 4.5: Example Packing Function

To export the components attribs, they need to be arranged into a contiguous memory block. This is done by writing the individual *CopyVectors* provided by the *pack* function sequentially into the shared memory segment.

Figure 4.3 shows how the linearized attrib map is arranged in the shared memory.

As the number and size of the attribs varies heavily, the linearized attrib map is preceded by a `size_t num_attribs` variable in the shared memory, as illustrated in Figure 4.3. The `num_attribs` value indicates the number of attribs exported from the components map to the importing process.

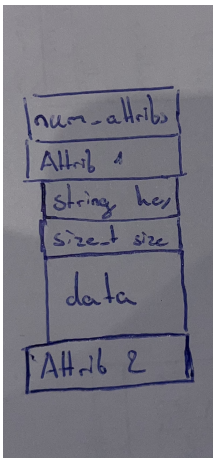

Figure 4.3: Attribs in Shared Memory

4.3.2 Unpacking and Importing Attribs

To import the components attribs, the original attrib map needs to be recreated in the receiving process. This is done as follows:

1. Reading the number of attribs from the top of the components attrib memory region as illustrated in Figure 4.3.
2. Iterating over the attribs, reading the key string and the size of the data.
3. Recreating the attrib key-value pair using the key and copying the data.

After the linearized attrib map has been read and the attrib key-value pairs have been recreated in the importing process, the original state of the data has to be recreated. For this purpose, the user has to supply an *unpack* function with the signature shown in Listing 4.6. The unpack function takes the `std::pair<std::string, void*>` created earlier, which consists of the attrib key and a pointer to a single contiguous memory block and returns the attrib in its original state, as it was before the *pack* function was called by the exporting process.

```
std::pair<std::string, void *> unpack(size_t size, std::pair<std::string,
    void *> attrib);
```

Listing 4.6: Unpack Function Declaration

In the example of the linked list used in subsection 4.3.1, the *unpack* function would receive a key value pair containing the attrib key and the *int* array created by the *pack* function earlier as well as the total size of the array in byte.

The *unpack* function would then recreate the original linked list by reading the individual elements of the array and return the original attrib key value pair containing the key string and the pointer to the linked list.

4.4 DataPaths

Apart from the component tree, sys-sage topologies, consist of *DataPaths*. DataPaths are used to connect two components of the component tree, to represent arbitrary relations between them.

Listing 4.7 shows part of the DataPath class implementation. DataPaths have *source* and *target* component pointers, as shown in line 6 and 7, that represent the components connected by the DataPath.

Additionally, DataPaths can either be oriented, meaning that the source and target components are differentiated, or bidirectional, which is indicated by the *oriented*

variable. The *dp_type* variable can be used to attach additional information about the nature of the relation between the components, such as a physical hardware connection or a shared cache.

Just like components, DataPaths have an *attrib* map, which can be used to attach arbitrary data to the DataPaths using key-value pairs.

```
1  class DataPath {
2    public:
3      map<string, void*> attrib;
4
5    private:
6      Component* source;
7      Component* target;
8
9      const int oriented;
10     const int dp_type;
11 };
```

Listing 4.7: DataPath Class

4.4.1 Exporting DataPaths

Since DataPaths are used to connect two components, they only need to be exported if both the source and target component are exported. While exporting the component tree, each components DataPaths will be put in a `std::map<DataPath*, std::pair<size_t, size_t>>`, mapping the DataPath to a `std::pair<size_t, size_t>`, storing the offsets of the DataPaths components in the shared memory region.

After the entire component tree is exported, it can easily be determined, which DataPaths need to be exported, by checking if both component offsets are set.

The actual export process for the DataPaths works as follows:

1. Writing the total number of DataPaths exported to the shared memory region.
2. Writing the offsets of the DataPaths source and target components, relative to the top of the memory mapped file, to the shared memory.
3. Exporting the actual DataPath object using `memcpy()`.
4. Exporting the DataPaths *attribs* using the provided *pack* function as described in section 4.3.

Figure 4.4 illustrates how the DataPaths are arranged in the shared memory file.

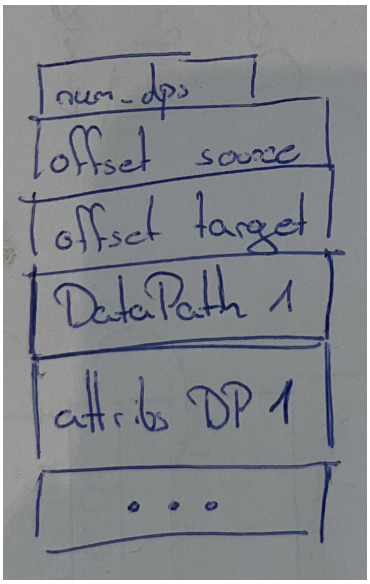

Figure 4.4: DataPaths in Shared Memory

4.4.2 Importing DataPaths

To import the DataPaths into the local memory of the receiving process, the steps described in subsection 4.4.1 need to be reversed:

1. Reading the total number of DataPaths from the top of the DataPaths memory region.
2. Iterating over the shared memory block reading the source and target offsets and recreating the DataPath object using its copy constructor.

Since the components have been imported into the local memory and thus have a new memory location, the read offsets need to be translated to regular pointers pointing to the components new memory location.

For this purpose, a `std::map<size_t, Component*>` is created while exporting the component tree, mapping the components offset in the shared memory region to its memory location in the importing process.

3. Recreating the DataPaths source and target pointers and adding the DataPath to the components DataPath vectors.
4. Importing the DataPaths *attribs* using the provided *unpack* function as described in section 4.3.

5 Performance

In the context of high-performance computing, where computational performance and efficiency are crucial, it is paramount to analyze every small piece of software used in the cluster. This is especially important for this thesis, as the *sys-sage* library is designed to be used in complex, heterogeneous high-performance compute clusters with complicated hierarchical hardware topologies.

As such, analyzing the shared memory implementation of this thesis on metrics such as execution time and scalability is unavoidable.

Since the memory footprint of a topology in shared memory is mostly dependent on the size of the topology itself, with little overhead created by the export process, the memory impact of sharing topologies using this implementation will not be evaluated as part of this thesis.

5.1 Execution

To perform the performance analysis of the *sys-sage* shared memory implementation, a series of sample topologies will be created, each with different compositions of components, DataPaths and attributes.

The different topologies will then be exported into shared memory regions and subsequently imported into the local memory of another process, measuring import and export times separately.

To investigate the scalability of the implementation, each topology will be evaluated in multiple sizes, adding more components, DataPaths or attribs as needed.

All performance tests will be executed on an Apple M1 processor with 16GB of RAM. To create a more reliable performance analysis, all topologies will be evaluated multiple times and the average execution times will be calculated.

Listing 5.1 shows how the execution time of exporting a topology is measured. In line 1 and 3, the current time is taken before and after the execution of the measured code section. This is done using `std::chrono::steady_clock`, an implementation of a *monotonic* clock, best suited for measuring time intervals [cpl]. In lines 5-6, a `std::chrono::duration` is created, representing the time elapsed between *time_start* and *time_end*, which is the execution time of the measured code region.

```
auto time_start = std::chrono::steady_clock::now();
SharedMemory* shmem = export_component(path, topo, pack);
auto time_end = std::chrono::steady_clock::now();

std::chrono::duration<double, std::milli> duration =
    time_end - time_start;
```

Listing 5.1: Measuring the Execution Time

5.2 Basic Component Tree

For the first performance test, a simple component tree without any DataPaths or attribs is created. The component tree consists of an increasing number of components in a straight line of children, similar to a linked list.

Components	Export	Import
100	0.172	0.138
1000	1.274	1.239
10000	14.736	16.442

Figure 5.1: Performance Component Tree

Figure 5.1 shows the execution time in milliseconds of exporting or importing this topology. As expected, the execution time grows linearly with the number of components in the topology.

5.3 Attribs

As complex sys-sage topologies can potentially have a lot of attribs that will often make up a major part of the total data amounts, evaluating the performance of exporting and importing attribs is paramount to analyzing the performance of the entire system.

Figure 5.2 shows the execution time of exporting or importing a topology consisting of a single component, with increasing amounts of attribs.

It is important to note that the performance of sharing complex data attached to an attrib will largely depend on the performance of the *pack* and *unpack* functions provided

Attribs	Export	Import
100	0.265	0.264
1000	2.426	2.785
10000	20.357	31.680
100000	196.529	361.431

Figure 5.2: Performance Attrib Topology

by the user. As this factor is impossible to quantify in a meaningful way, the attribs used for the purposes of this performance evaluation are simple integers that don't need to be packed or unpacked.

Figure 5.2 shows a linear increase of the execution time with growing number of attribs shared. The performance of export and import are roughly equal, with the import slowing down faster with larger amounts of attribs.

5.4 DataPaths

To test the performance of exporting and importing DataPaths, a small topology consisting of only two nodes is created. These components are then connected by a large number of DataPaths without attribs or any additional data, so as not to skew the results.

DataPaths	Export	Import
100	0.519	0.075
1000	3.584	0.456
10000	44.727	4.248

Figure 5.3: Performance DataPath Topology

Figure 5.3 shows the execution time of exporting or importing the topology described

above with different amounts of DataPaths. Apart from some minor fluctuations, the execution time increases roughly linear with the amount of DataPaths in the topology. The results are as expected, as the DataPaths are simply one after the other, with very little overhead or other performance limiting factors.

It is also notable that importing is consistently almost ten times faster than exporting. This is likely due to the exporting process needing to evaluate which DataPaths need to be exported, whereas the importing process can directly start importing.

6 Conclusion and Future Work

In this thesis, a file backed shared memory system was implemented, which expands the existing sys-sage library to allow users to share sys-sage topologies between processes and nodes in HPC systems.

The systems creates memory mapped files, shared between different processes, which can then be used to transfer topologies between the processes. To do this, all necessary components, DataPaths and attribs need to be copied into the shared memory region by the exporting process. Any process wishing to import the topology can then use this memory region to recreate the topology in its local memory.

One of the main drawbacks of the shared memory system implemented in this thesis is its inability to update shared topologies once they have been exported. This could be an issue for sys-sage users, as topologies are often prone to frequent changes and updates, such as regular measurements being saved in a components attribs. As it stands, each process would either have to provide its own version of the necessary measurements, or the the components in question would have to be exported after every update. Either solution could have significant impact on the systems performance, depending on the size of the components and the frequency of the updates.

In the future, it could be beneficial to implement additional functionality allowing users to update existing shared topologies without having to redo the entire export. Ideally, every process would be able to create updates to shared topologies, not just the process that originally exported the topology.

This would make it much easier for users to keep all their processes up to date on the topology of the HPC system.

Additionally, it would be convenient if shared component subtrees could be automatically integrated into the topology of receiving processes. This would make it much easier for users to share smaller parts of topologies, without having to add it to the topology manually.

Despite these drawbacks, the shared memory implementation works as intended and provides the necessary tools to share topologies within HPC systems.

List of Figures

2.1 Basic Component Tree with DataPaths 4

4.1 Component Tree in Shared Memory 11
4.2 Packing a Linked List 14
4.3 Attribs in Shared Memory 15
4.4 DataPaths in Shared Memory 18

5.1 Performance Component Tree 20
5.2 Performance Attrib Topology 21
5.3 Performance DataPath Topology 21

Listings

4.1 SharedMemory Class 9
4.2 Component Class 10
4.3 Component Class 12
4.4 CopyAttrib Struct 14
4.5 Example Packing Function 14
4.6 Unpack Function Declaration 16
4.7 DataPath Class 17

5.1 Measuring the Execution Time 20

Bibliography

[Aa18] S. B. Alotaibi and D. F. alboraei. "Topology-Aware Mapping Techniques for Heterogeneous HPC Systems: A Systematic Survey." In: *(IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 9, No. 10*. 2018.

[Bro+a] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. *Hardware Locality (hwloc) Documentation*. `https://www.open-mpi.org/projects/hwloc/doc/hwloc-v2.10.0-a4.pdf`. Accessed: 2024-05-12.

[Bro+b] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. *Portable Hardware Locality (hwloc)*. `https://www.open-mpi.org/projects/hwloc/`. Accessed: 2024-04-29.

[Bro+10a] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. "hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications." In: *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. 2010, pp. 180–186. DOI: 10.1109/PDP.2010.67.

[Bro+10b] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst. "ForestGOMP: an efficient OpenMP environment for NUMA architectures." In: *International Journal of Parallel Programming* (2010).

[CHC12] D. Chavarría-Miranda, Z. Huang, and Y. Chen. "High-performance computing (HPC): Application & use in the power grid." In: *2012 IEEE Power and Energy Society General Meeting*. 2012, pp. 1–7. DOI: 10.1109/PESGM.2012.6345493.

[CLP22] A. Crotty, V. Leis, and A. Pavlo. "Are You Sure You Want to Use MMAP in Your Database Management System?" In: *CIDR 2022, Conference on Innovative Data Systems Research*. 2022.

[cpl] cplusplus.com. *std::chrono::steady_clock*. `https://cplusplus.com/reference/chrono/steady_clock/`. Accessed: 2024-05-11.

[GM11] B. Goglin and S. Moreaud. "Dodging Non-Uniform I/O Access in Hierarchical Collective Operations for Multicore Clusters." In: *CASS 2011: The 1st Workshop on Communication Architecture for Scalable Systems, held in conjunction with IPDPS 2011*. 2011.

[Mic] M. Michelsen. *sharedstructures*. `https://github.com/fuzziqersoftware/sharedstructures/tree/master`. Accessed: 2024-05-12.

[Nie16] F. Nielsen. "Introduction to MPI: The Message Passing Interface." In: Feb. 2016, pp. 21–62. ISBN: 978-3-319-21902-8. DOI: 10.1007/978-3-319-21903-5_2.

[Rao+08] S. T. Rao, E. Prasad, N. Venkateswarlu, and B. Reddy. "Significant performance evaluation of memory mapped files with clustering algorithms." In: *IADIS International conference on applied computing, Portugal*. 2008, pp. 455–460.

[Van] S. Vanecek. *sys-sage*. `https://github.com/caps-tum/sys-sage/tree/master?tab=readme-ov-file`. Accessed: 2024-04-29.