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

Contents

1 Introduction

1.1 Motivation

Due to the large number of cores used in high-performance computing (HPC), 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]

Due to the high degree of parallelization in HPC systems, memory bandwith and cache usage have tremendous impact on the overall performance. [Bro+10b]

Thus, making full use of the hardware topology to schedule threads accordingly is crucial in HPC clusters. Threads working closely together will often benefit from sharing a cache to utilize local memory as much as possible, whereas independant, memory intensive jobs could be better scheduled onto seperate proccessors so as not to limit their memory bandwith and available cache storage. [Bro+10a]

Many tools, such as *Portable Hardware Locality (hwloc)* [Bro+] 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.

1.2 Related Work Or Objectives

2 Sys-Sage

2.1 Functionality

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.

[Something about data sources / input parsers?]

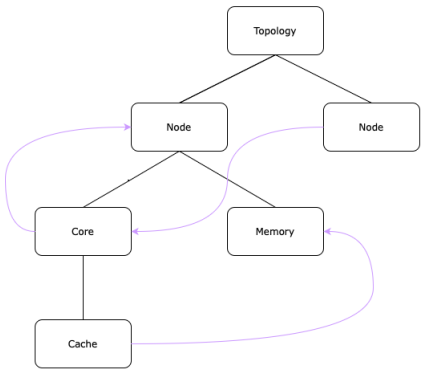

Figure 2.1: Basic Component Tree with DataPaths

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.

[Something about use-cases? User-flow? API? My Contribution?]

3 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 attruibutes 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.

3.1 Capabilities and Usage

If at any point during the lifetime of a sys-sage topology,

3.2 Shared Memory

The data sharing aspect of the implementation is realised 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 amiguity 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 3.1.

```
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 3.1: 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 3.1, 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 seperately. 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 comunication needed.

[Size Calculation]

3.3 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 3.2 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 3.2: Component Class

As shown in Listing 3.2, 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.

3.3.1 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 3.1 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.

3.3.2 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

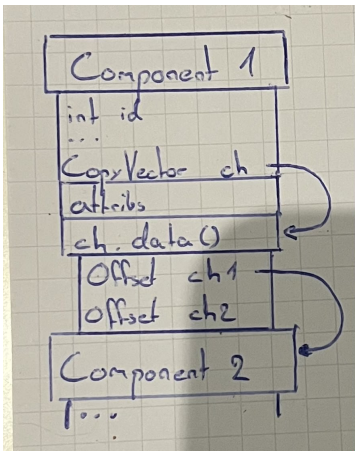

Figure 3.1: Component Tree in Shared Memory

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 3.3.

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

Listing 3.3: 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.

3.3.3 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 3.3.1, need to replaced with pointers to their respective memory addresses in the importing process.

3.4 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.

3.4.1 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 3.4. *CopyAttribs* consist of the attribs *key*, a *size* variable storing the datas total size in byte and a pointer to the linearized data block.

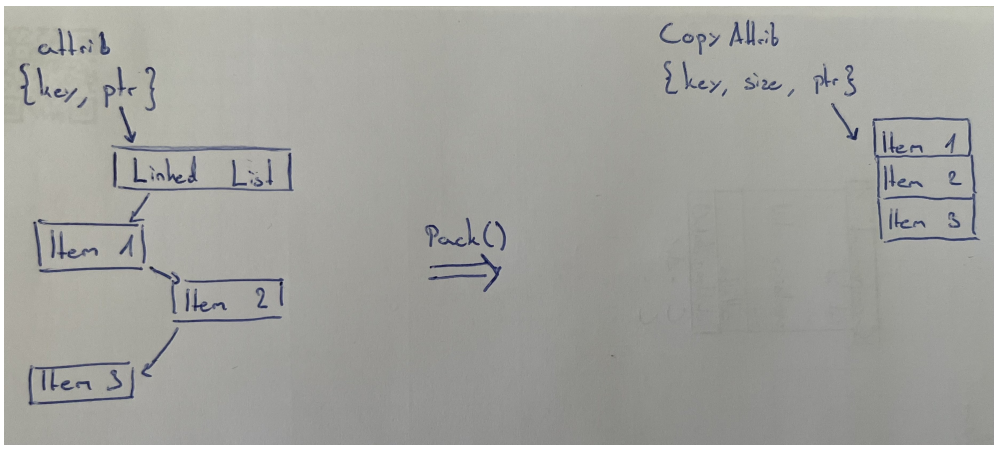

Figure 3.2: Packing a Linked List

Figure 3.2 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.

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

Listing 3.4: CopyAttrib Struct

Listing 3.5 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 3.5: 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 3.3 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 preceeded by a `size_t num_attribs` variable in the shared memory, as illustrated in Figure 3.3. The `num_attribs` value indicates the number of attribs exported from the components map to the importing process.

3.4.2 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 3.3.
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.

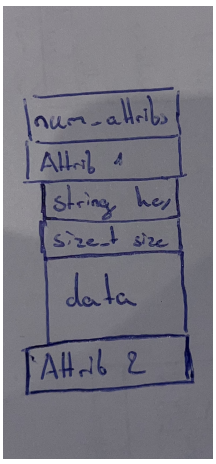

Figure 3.3: Attribs in Shared Memory

For this purpose, the user has to supply an *unpack* function with the signature shown in Listing 3.6. 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 3.6: Unpack Function Declaration

In the example of the linked list used in subsection 3.4.1, 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.

3.5 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 3.7 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 inforamtion 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 3.7: DataPath Class

3.5.1 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 eporvided *pack* function as descibed in section 3.4.

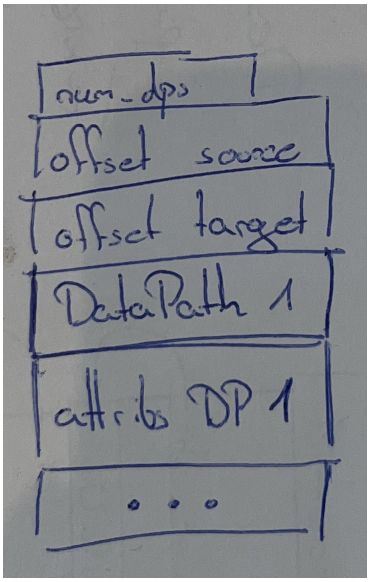

Figure 3.4: DataPaths in Shared Memory

Figure 3.4 illustrates how the DataPaths are arrangend in the shared memory file.

3.5.2 Importing DataPaths

To import the DataPaths into the local memory of the receiving process, the steps described in subsection 3.5.1 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 descibed in section 3.4.

List of Figures

List of Tables

Bibliography

[Bro+] 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).

[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.

[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.

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