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?]

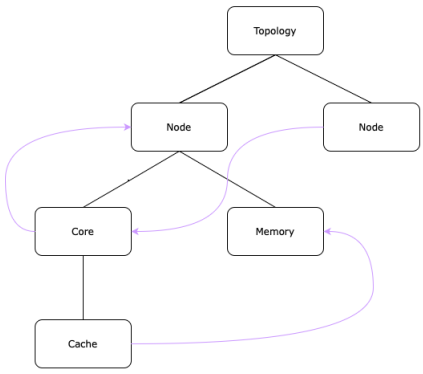

Figure 2.1: Basic Component Tree with DataPaths

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

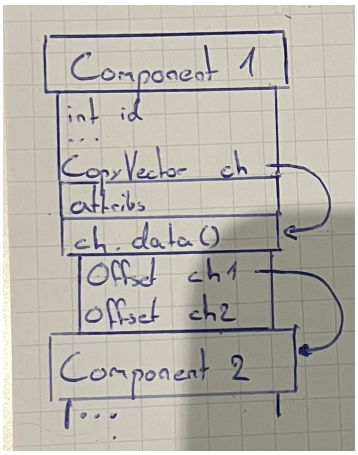

Figure 3.1: Component Tree in Shared Memory

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, 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

Components and DataPaths have an *attrib* map that stores key-value pairs that can be used to add arbitrary data of any size. It is implementated as a `std::map<std::string, void*>`, mapping strings to

3.5 DataPaths

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.