Prototype for the Dynamic Topology Information System

Deliverable No:	D3.1
Deliverable Title:	Prototype for the Dynamic Topology Information System
Deliverable Publish Date:	30 September 2022
Project Title:	SparCity: An Optimization and Co-design Framework for Sparse Computation
Call ID:	H2020-JTI-EuroHPC-2019-1
Project No:	956213
Project Duration:	36 months
Project Start Date:	1 April 2021
Contact:	sparcity-project-group@ku.edu.tr

List of partners:

Participant no.	**Participant organisation name**	**Short name**	**Country**
1 (Coordinator)	Koç University	KU	Turkey
2	Sabancı University	SU	Turkey
3	Simula Research Laboratory AS	Simula	Norway
4	Instituto de Engenharia de Sistemas e Computadores, Investigação e Desenvolvimento em Lisboa	INESC-ID	Portugal
5	Ludwig-Maximilians-Universität München	LMU	Germany
6	Graphcore AS	Graphcore	Norway

CONTENTS

1 INTRODUCTION

The SparCity project is funded by EuroHPC JU (the European High Performance Computing Joint Undertaking) under the 2019 call of Extreme Scale Computing and Data Driven Technologies for research and innovation actions. SparCity aims to create a supercomputing framework that will provide efficient algorithms and coherent tools specifically designed for maximizing the performance and energy efficiency of sparse computations on emerging High Performance Computing (HPC) systems, while also opening up new usage areas for sparse computations in data analytics and deep learning.

Sparse computations are commonly found at the heart of many important applications, but at the same time it is extremely challenging to achieve high performance when performing the sparse computations. SparCity delivers a coherent collection of innovative algorithms and tools for enabling high efficiency of sparse computations on emerging hardware platforms. More specifically, the objectives of the project are:

- to develop a comprehensive application and data characterization mechanism for sparse computation based on the state-of-the-art analytical and machine-learning-based performance and energy models,
- to develop advanced node-level static and dynamic code optimizations designed for massive and heterogeneous parallel architectures with complex memory hierarchy for sparse computation,
- to devise topology-aware partitioning algorithms and communication optimizations to boost the efficiency of system-level parallelism,
- to create digital SuperTwins of supercomputers to evaluate and simulate what-if hardware scenarios,
- to demonstrate the effectiveness and usability of the SparCity framework by enhancing the computing scale and energy efficiency of challenging real-life applications.
- to deliver a robust, well-supported and documented SparCity framework into the hands of computational scientists, data analysts, and deep learning end-users from industry and academia.

1.1 OBJECTIVES OF THIS DELIVERABLE

The goal of this deliverable is to describe the prototypical realization of our dynamic topology system. We discuss the shortcomings of existing approaches to represent and manage hardware topology and describe the design of our approach, called **yloc**. We then discuss the prototypical implementation of our demonstrator, which is able to integrate data from multiple sources.

1.2 WORK PERFORMED

We have analyzed the shortcomings of exising tools in use to represent the toplogy information of modern HPC systems and devised a design that can flexibily represent toplogy information from multiple sources. A prototyplical implementation was created as a demonstrator that can integrate topology information from multiple sources, including Hwloc, multinode information from MPI and accelerator data from AMD or Nvidia GPUs.

1.3 DEVIATIONS AND COUNTER MEASURES

Due to hiring problems, work on work package WP3.1 could only be started with a delay of about 6 months. This delay was, however, fully compensated with an increased workforce starting at month 12 of the project.

1.4 RESOURCES

The source code for our dynamic topology information system is publicly available at the following location: https://github.com/sparcityeu/yloc

2 THE DYNAMIC TOPOLOGY SYSTEM YLOC

The hardware structure of current and future HPC systems is getting increasingly complex. The simple uniprocessors and symmetric multiprocessors of the past have been replaced by multicore chips with increasing core counts and on-chip networks that feature their own complex interconnect topology. Large-scale shared memory systems feature multi-level NUMA (non-uniform memory access) architectures that complicate programming for memory locality and chiplet-based manufacturing may expose another level of hardware complexity to the operating system and user. At a larger scale, HPC systems are composed of an increasing number of nodes and a variety of interconnect networks are employed that are not always representable as trees. The trend towards accelerator devices (e.g., GPUs) has led to yet another level of complexity for managing the available compute resources in these accelerator devices, which sometimes come equipped with dedicated accelerator-to-accelerator interconnects.

To manage the aforementioned complexity, a number of tools have been developed that support the discovery of hardware components available in a given system and their interconnection. The resulting information is usually referred to as the **hardware topology**. Some of these tools are maintained by the hardware manufacturers, some are developed as part of the operating system, some come from research organizations.

A representative list of tools available in this space is provided here for reference:

- lshw – Linux operating system tool to display the hardware configuration of a machine.
- cpuid – Operating system interface for querying information about CPUs.
- likwid-topology – Tool for displaying information about thread, core, cache, and memory topology.[1]
- pciutils – Offers portable access to PCI bus configuration.
- hwloc – Widely used tool to obtain the hierarchical map of computing elements such as CPUs, cores, memory, etc.[2]
- netloc – Portable network locality discovery tools and abstract representation of network topologies.[3]

[1] J. Treibig, G. Hager, and G. Wellein. "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments". *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*. 2010.

[2] François Broquedis et al. "hwloc: A generic framework for managing hardware affinities in HPC applications". *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE. 2010, pp. 180–186.

[3] Brice Goglin, Joshua Hursey, and Jeffrey M Squyres. "Netloc: Towards a comprehensive view of the HPC system topology". *2014 43rd International Conference on Parallel Processing Workshops*. IEEE. 2014, pp. 216–225.

- nvsmi and NVML – Command line utility and management library for NVIDIA graphics card devices.
- rocm-smi – Command line utility and management library for AMD graphics card devices.

Hwloc is an especially widely used tool that is also integrated as a portable data source in a number of other tools, such as likwid-topology.

Although many tools exist, which have the express goal to display the available hardware components and their connectivity, these systems suffer from a number of limitations:

- Often a tree representation is employed to organize the discovered components (e.g., a 'node' contains 'CPUs', which themselves contain 'physical cores', which may be composed of 'logical cores'). While such an organization is useful and often simple to work with, it no longer represents reality with fidelity in many cases. For example, a strict tree representation fails to model the physical or logical proximity of cores in a ring or 2D-torus core-to-core interconnect. It can also not be used to model locality of memory to compute elements (i.e., there may be more than one level of non-local memory, from the point of view of one compute core).
- Systems are often composed of many components by different vendors, and it is thus difficult to integrate a full topology picture of all components. Hwloc and operating system tools try to keep portable and up-to-date with current hardware developments, but it is difficult to integrate all data sources in one tool, if a tool follows a rigid data-model.
- There is generally no support for attaching dynamic (time-varying) information to the components of the topology. If such an augmentation is desired, a developer needs to import the topology information into a custom data structure and perform the augmentation there. Examples for such time-varying information could be temperature, power consumption, or hardware-derived or application-specific load data.
- There is little support to turn the available data into actionable insights. Common use-cases are often not directly supported by the tools, but require the user to navigate the available information manually to derive the necessary conclusion. Desirable would be tools that make it easier to work with topology information in a unified approach, such that common use-cases can be expressed more easily.

Hwloc is the most well known and widely used tool for dealing with the complex topology of modern machines in practice. This framework also suffers from some of the aforementioned problems, explicitly recognizing for example the inadequate nature of trees for representing the topology information in many cases.[4] The solution adopted by Hwloc to address this challenge is to organize compute objects and memory objects into separate hierarchies by using virtual negative depths for memory objects.

To offer a more general and widely applicable solution to the shortcomings and challenges identified above, we have developed **yloc**, a dynamic topology information system. The ideas and principles underlying yloc can be summarized as follows:

- **Flexible data schema.** Instead of relying on a tree data structure, data storage and representation in yloc is based on a flexible (multi-)graph representation. Nodes in this multi-graph

[4]Brice Goglin. "Exposing the locality of heterogeneous memory architectures to HPC applications". *Proceedings of the Second International Symposium on Memory Systems*. 2016, pp. 30–39.

represent components in the hardware setup of a machine, edges may represent any labelled relationship between these hardware components.

- **Support for multiple data sources.** Since our data representation format is very general, the tool has the ability to ingest and combine data from a variety of sources. Hwloc data can be easily represented in yloc, because our graph storage model is more general than the trees used by hwloc. Similarly, other tools can be integrated with little difficulty in the yloc model. In terms of our yloc prototype implementation, the ability to support multiple data sources is aided by a module system that makes it comparatively easy to add new data sources, such as vendor-specific tools.

- **Basic abstract machine model.** To attach semantic meaning to the components of the topology graph (nodes and edges), a simple extensible abstract machine model has been developed. This model covers the most important aspects of current-generation hardware found in high performance computing systems, but is also extensible in order to allow it to evolve to cover future hardware developments.

- **Unification and integration of data sources.** Since yloc allows data from multiple sources to be integrated into a single graph representation, the data domains of these sources may be overlapping. In order to provide a way to produce a coherently integrated data model, yloc identifies nodes that correspond to each other in different (sub-)graphs generated by different modules. For example, hwloc may provide the components of a single compute node including the location of the PCIe bridge, while a GPU manufacturer's tool may provide details about the GPU hardware while also including the PCIe bridge which can thus be used as the connection node for the two component graphs.

- **Support for dynamic data.** In addition to static information (hardware components and their relationship or connection), it may be often useful to augment this static structure with dynamic (time-varying) properties. Dynamic information may arise based on time-varying hardware properties, such as temperature or operating frequency of compute elements, or it may originate from software components, such as an application's dynamic load information.

- **Query support.** Having a flexible way to represent data in an information system is only a necessary, but not sufficient, condition for a useful and practical approach. What is additionally needed is a way to extract and find relevant data, i.e., a way to formulate and execute queries that can find and filter the data. This approach is supported in yloc by means of operations on the underlying graph such as predicate-based graph views and support for graph algorithms such as computing shortest paths or performing a breadth-first search.

2.1 DESIGN OF YLOC

A main goal underlying the design of yloc is extendability throughout all parts of the library. This is achieved by a modular and layered design.

Topology information is gathered from various data sources, provided by modules, and aggregated into a unified system topology graph. The graph structure is provided by a graph library. This graph structure is extended by our library in order to fulfill our requirements. On top of this unified topology graph, it is possible to query specific information, or to run graph

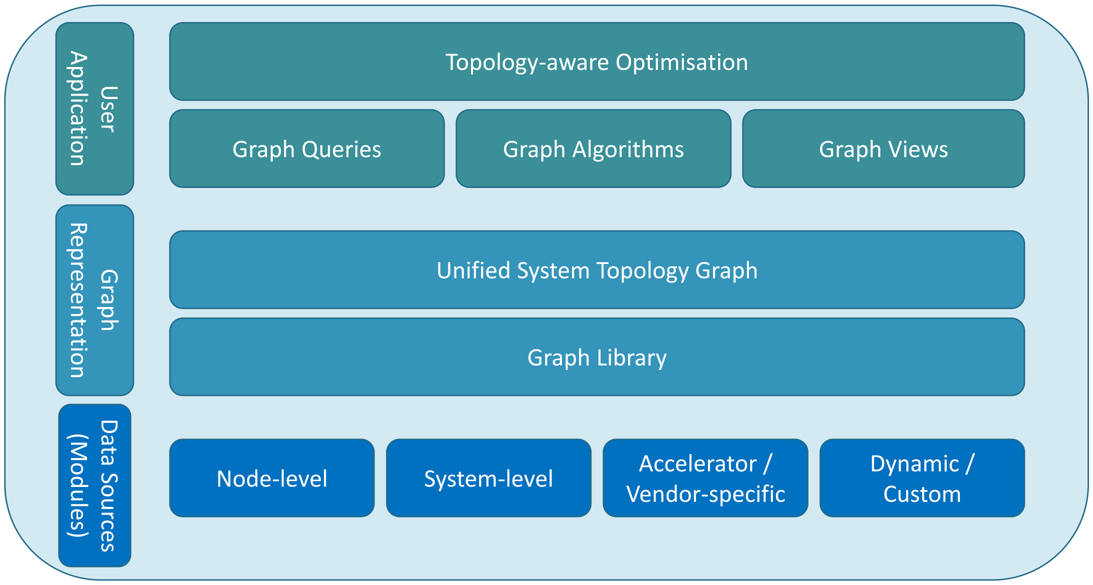

Figure 1 *Layered software architecture of yloc.*

algorithms. This enables use cases like, for example, topology-aware optimizations. See Figure 1 for an overview of this architecture.

The following subsections introduce the design parts of yloc in more detail. The first Subsection 2.1.1 describes the design of the modules. Afterwards, Subsection 2.1.2 introduces the requirements for the unified graph structure. Finally, Subsection 2.1.3-2.1.5 describe the abstract machine model and our methods to access topology information.

2.1.1 MODULES

Modules are the connection between data sources and yloc. A module is a wrapper for other libraries or tools that implements them as a yloc data source.

A possible design strategy for an extensible module system is to realize modules in the form of shared libraries that can be dynamically loaded by yloc. A downside of this approach is the increased client-side complexity for managing the dynamic loading, and the security risks that are associated with this approach. An alternative approach is to implement the modules as runtime polymorphic objects. All modules implement virtual interfaces that are defined by our library. These interfaces are introduced later in this section and in Section 2.1.4.

These virtual interfaces allow resolving the references to module code at run-time while keeping compile-time dependencies at a minimum. This ensures easy extendability and even allows linking against modules where no source code is available, which can be beneficial for third parties that do not want to provide source code.

The following list describes the methods of the module interface:

- **init_graph.** This function is responsible for creation of the module-specific part of the topology graph. It is invoked on library initialization for all enabled modules. The different parts are combined into a unified system topology graph.
- **export_graph.** This function can be implemented to provide interoperability with another library. The module can extract data from the topology graph into a data format that is compatible with another library. This would allow using yloc as drop-in replacement for

other libraries in existing use-cases. Another purpose of this function is a serializer module that serializes the topology information to import it on another machine.

- **update_graph.** This function is used to query and cache information from the modules and underlying structures. Not every piece of information is available instantly, and some sort of caching is necessary to provide a seamless user experience. `update_graph` can be used to trigger an update on these caches. This function can be invoked by the user application to update information that is expensive to retrieve when resources are free, for example during communication phases.

- **set_option.** Many of the tools and libraries used as data sources have a plethora of options. This function provides a uniform way to set their options.

2.1.2 Unified system topology graph

To minimize redundant work, we use an off the shelf graph library. In our case, we use the Boost Graph Library (BGL). This has the benefit that we don't need to implement the graph structures in our library, and the BGL also includes some graph algorithms out of the box.

During the initialization of our library, all modules operate on the same topology graph. This introduces conflicts when multiple modules describe the same hardware components. In order to have a unified graph, the modules need to edit the same part of the topology. Unfortunately, the BGL does not provide mechanisms that aid with this. All vertices are strictly identified by the order they were created in. By default, there is no efficient way to access a vertex by its properties.

One promising feature of the BGL are subgraphs. During the implementation, we experimented with them to see if they can solve this problem. However, this is not the case and the current approach does not use subgraphs. Although subgraphs are still interesting for other use cases. For example, it would be possible to use them to group related components in the graph. This could help for partitioning algorithms that try to distribute work across the available hardware.

Therefore, we extended the graph structure to also include identifiers that can be used by different modules to refer to the same hardware component. The idea behind this is that every component in the system can be identified unambiguously. However, the usage is still optional, for example when a module introduces logical elements to the graph that are only meaningful to this module. For more details on these identifiers, see Section 2.2.1.

2.1.3 Abstract machine model

An abstract machine model allows programmers to better reason about optimizing data placement and locations of computation by focusing on key aspects of a machine that are relevant.[5] The abstract model is agnostic to implementation- or vendor-specific hardware details and defines the organization of hardware components in a hierarchy. A node-level abstract machine model is composed of a processor, memory, accelerator, interconnect, I/O, and storage model. A system-level machine is represented by a system-scale model that integrates multiple node-level models of the computing system and the system-level interconnect model (see Figure 2). The goal of the abstraction is to facilitate porting applications from one system to another, or to enable automated methods of optimizing the mapping of data and computation to the system.

[5]James A Ang et al. "Abstract machine models and proxy architectures for exascale computing". *2014 Hardware-Software Co-Design for High Performance Computing*. IEEE. 2014, pp. 25–32.

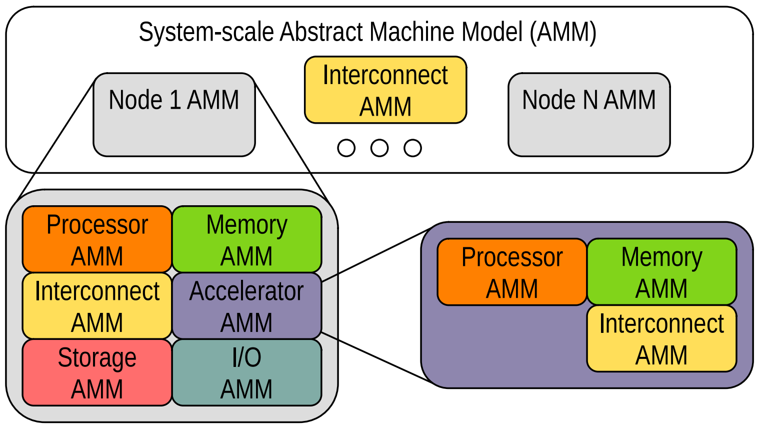

Figure 2 *Yloc abstract machine model.*

Yloc represents a machine's hardware components and their hierarchy in a topology graph. The topology graph of a machine, together with the *component types* (Section 2.1.5) and *properties* (Section 2.1.4) is an instantiation of an abstract machine model.

Not every aspect of the model will be relevant to a specific use case. Therefore, the topology hierarchy and properties are not strictly specified by yloc, but are defined by the modules. Nevertheless, yloc defines a set of extendible component classes that categorize components based on their type.

2.1.4 COMPONENT PROPERTIES

Hardware components within a system have a heterogeneous set of properties. A cache, for example, has a capacity property, while a CPU core does not have this property, but other properties such as SIMD length. Certain property values, for example power consumption, can dynamically change over time (dynamic properties), introducing another level of complexity.

Instead of statically storing property values within the graph, we chose to use *adapters*. The adapters serve as a mediator between a data source (module) and the yloc library, similar to the adapter software design pattern. They do not directly store property values, but instead specify how certain properties can be retrieved from a module. Additionally, an adapter stores the required information to identify components within the module's data source. This design choice makes it possible to query dynamic properties at runtime. However, an adapter can in principle also store a static property value once it is retrieved from a module.

The set of properties provided by a specific module is defined in the module's adapter. When a module is able to deliver any property for a component, it will add its (module-specific) adapter to the component's list of adapters. Together, the adapters of a component define the component's complete set of properties.

The adapter concept is easily extendible by additional properties and modules. However, it is important that all modules use the same physical unit for a property to enable meaningful processing of property values, such as summing the power consumption of multiple components. Therefore, the yloc library defines a set of common properties together with their physical units that modules have to adhere to.

2.1.5 HIERARCHICHAL COMPONENT TYPES

Graph elements in the topology graph represent a system's hardware components or logical components. Whether a component is represented by a vertex or an edge depends on its component type. Examples for hardware components represented by graph vertices are cores and caches in a multicore processor. Memory, network devices, accelerators, and whole compute nodes are also represented by vertices. Graph edges, on the other hand, connect the vertices and represent physical or logical links between hardware components. Examples of components represented by edges are multicore interconnects, bus links, direct accelerator-to-accelerator links, and system-level interconnects.

Many components have shared capabilities, or belong to a subcategory of the same upper category. For example, CPUs and a GPUs both have compute capabilities. GPUs and FPGAs, both belong to the accelerator category. The abstract model of yloc defines the following component capabilities.

- Compute (e.g. CPU core, GPU cores)
- Store (e.g. cache, DRAM, hard disk drive)
- Data transfer, (e.g. network card, multicore interconnect, GPU interconnect)

Part of our abstract machine model is a component type hierarchy with an is-a relationship between component types (see Figure 3). This allows to express queries for a whole category of components or capabilities. The component type hierarchy can be extended by introducing a new subclass of a component type.

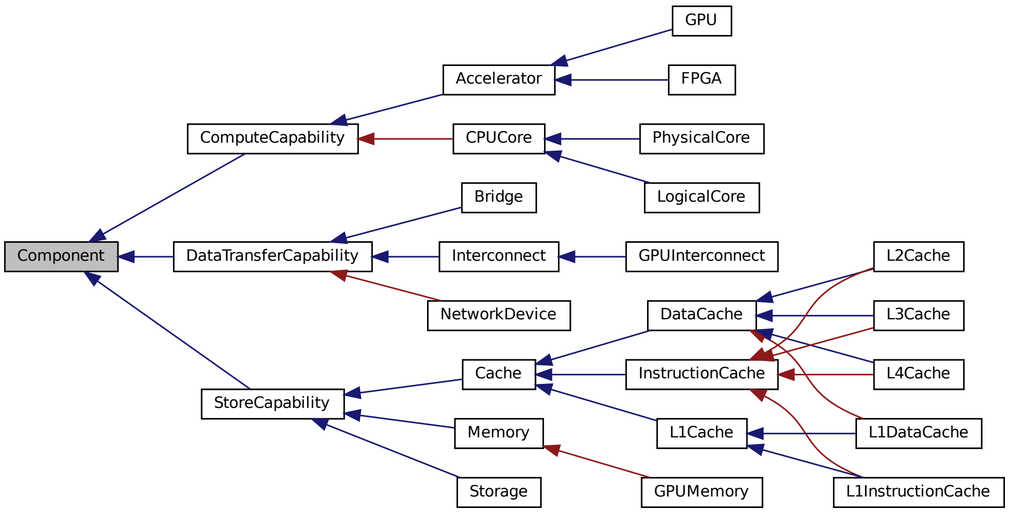

Figure 3 *Component type hierarchy of yloc.*

2.2 IMPLEMENTATION AND EVALUATION OF YLOC

We have implemented modules for the node level, accelerator level and system level topology. The node level uses the well known *hwloc*-library. On the accelerator level, we implemented modules for AMD's *ROCm SMI* and NVIDIA's *NVML* library. This covers two of the largest GPU manufacturers at this time. For the system level, we provide a demonstrator for the LRZ *SuperMUC-NG*-system using *MPI*. While this module is specific to this particular HPC system, it has the advantage that it does not require administrator privileges for its execution, such as tools like *ibnetdiscover* for InfiniBand-interconnects.

This Section describes the yloc implementation and is structured as follows. First, the graph structures from the BGL used in yloc, and our implementation of the component identifier are described in Subsection 2.2.1. The module implementation is discussed in Subsection 2.2.2, before Subsection 2.2.3 describes the properties implementation. The component type hierarchy implementation is described in Subsection 2.2.4. Subsection 2.2.5 presents two example use cases of yloc.

2.2.1 TOPOLOGY GRAPH AND VERTEX IDENTIFICATION

The topology graph is implemented using the Boost Graph Library (BGL). The underlying data structure is an adjacency list. The BGL allows choosing the underlying containers for the adjacency list. In our case, we have a static topology structure where no vertices are removed from the graph. For that reason, we chose a vector as storage container for the vertices. The indices of this vector are the *vertex descriptors*. These are used to access the graph vertices.

Since the BGL does not provide easy access to vertices based on their properties, we introduce an additional container that maps a component identifier to vertex descriptors. At the moment, we use a string as identifier. This is a good choice for simple use cases. It also allows easy extension for the prototype until the exact requirements of the identifier are known. However, it also would be possible to use a more complex type as key of this map. In the future, we could provide a specific type that uniquely identifies hardware components and improves compatibility between modules by providing an unambiguous frame.

This map is stored within a class (*yloc topology graph*) next to the BGL structure. This class provides additional methods, for example access to graph elements based on their identifier. The `add_vertex`-method can be used to add new vertices to the graph. If the optional identifier parameter is used, it returns the vertex with the same identifier if it already exists, otherwise a new vertex is created. Another useful method would be `add_edge` that should include extra checks that prevent the modules from adding multiple identical edges between the same vertices.

2.2.2 MODULE IMPLEMENTATION

The approach to create a new module is very easy. First, a copy of an existing module, which is used as a base, is created. It is only necessary to adjust very few class names, and afterwards the programmer can add their own code. Through the support of CMake the new module is automatically picked up and included in the next build process. A special example module for that purpose will be provided in the future.

At the moment, all modules only implement the `init_graph`-function. The current use cases do not require backwards compatibility with existing tools, and all information is queried live from the source library. Reasonable defaults are used at the moment for options, although, this is subject to change in future iterations of this software.

- **Hwloc.** The Portable Hardware Locality (hwloc) [6] software package provides an abstraction of the hierarchical hardware topology of modern architectures. It presents the different hardware components in a tree structure. We use this hwloc-tree to build the topology graph at the node-level. This is done by walking recursively through the tree visiting every node in the tree and adding it to the topology graph. Additionally, when available, identifiers are assigned to the entries in the graph. For example, PCIe devices get their

[6] https://www.open-mpi.org/projects/hwloc/

position in the system (bus, device, function and id) as identifier. This position is unique to the system and can be used by other modules to refer to the same PCIe device.

- **ROCm and NVML.** On the accelerator-level, we have implemented modules based on the ROCm System Management Interface (ROCm SMI) [7] library and the NVIDIA Management Library (NVML) [8] library. These C libraries provide a user space interface for applications to monitor and control states of GPU devices and applications (ROCm SMI for Linux only). The implemented modules capture details of the GPU topology of AMD and NVIDIA GPUs, such as PCIe links and accelerator-to-accelerator links. Additionally, they provide a vendor-agnostic method to query a set of properties of AMD and NVIDIA GPUs.

- **SuperMUC and MPI.** This module is a demonstrator for the LRZ SuperMUC-NG system [9]. It uses information derived from the hostnames of the nodes in order to create a topology graph from them. See Figure 4 for an example with 8 nodes. This example module shows how the internode topology of a machine can be derived and represented in yloc. In the case of SuperMUC-NG the machine is organized into islands (i), racks (r), chassis (c), and slots or servers (s), an organization that represents physical proximity in the computing system. Except for islands, the nominal interconnect speed between does not significantly depend on the physical location of a compute node in this system. However, the detailed physical location of a compute node may still be important to understand and optimize effects such as thermal load.

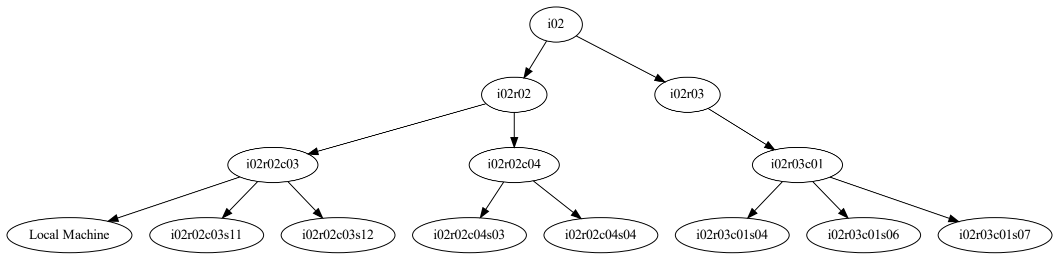

Figure 4 *Topology of 8 nodes from the SuperMUC-NG-system.*

2.2.3 PROPERTIES

Each module specifies its set of supported properties within the module's *property function map*. The property function map associates the name of properties with corresponding member functions (*property functions*) of the module's adapter class. Listing 1 shows a code snippet of the property function map implementation in the ROCm module adapter. Two properties are in the example's set of properties: numa affinity and temperature. The helper function `make_property` creates a property object with base class *AbstractProperty*, that wraps the member function pointer of a property.

[7] https://github.com/RadeonOpenCompute/rocm_smi_lib
[8] https://developer.nvidia.com/nvidia-system-management-interface
[9] https://doku.lrz.de/display/PUBLIC/SuperMUC-NG

Listing 1: ROCm module property function map code snippet.

```
// rocm_adapter.h:
static std::unordered_map<const char *, AbstractProperty *> &rocm_property_fn_map()
{
    static std::unordered_map<const char *, AbstractProperty *> pmap{
        {"numa_affinity", make_property(&RocmAdapter::numa_affinity)},
        {"temperature",   make_property(&RocmAdapter::temperature)}};
    return pmap;
}

virtual std::unordered_map<const char *, AbstractProperty *> &property_map()
{
    return rocm_property_fn_map();
}
```

Listing 2 shows an example of a dynamic- and static property function in the ROCm module. A static property (`numa_affinity`) can be stored in a module's adapter member variable, once it is retrieved for the first time from the data source. Dynamic properties (`temperature`) can vary over time and are queried at runtime, for example via a library function call. Besides the property function, an adapter also stores data to identify a graph component within the module's data source. This is the GPU device index provided by the ROCm-SMI library in case of the ROCm module.

Listing 2: Property function example.

```
// rocm_adapter.h:

// static property function:
std::optional<uint64_t> numa_affinity() const override
{
    uint64_t numa_affinity;
    // avoid successive library function calls after first query:
    if(!m_numa_affinity.has_value()) {
        rsmi_topo_numa_affinity_get(m_device, &numa_affinity);
        m_numa_affinity = numa_affinity;
    }
    return m_numa_affinity;
}

// dynamic property function:
std::optional<int64_t> temperature() const override
{
    int64_t temperature;
    // queries current temperature with a library function call:
    rsmi_dev_temp_metric_get(
        m_device, RSMI_TEMP_TYPE_EDGE, RSMI_TEMP_CURRENT, &temperature);
    return temperature;
}

private:
std::optional<uint64_t> m_numa_affinity{}; // adapter member for static property
uint32_t m_device;                         // ROCm device index
```

Yloc's set of common properties together with their physical units is defined in the adapter base class. Those common properties are documented, and a module must override the corresponding property function in its adapter to implement it. However, extending the set of properties by a custom property is simple: adding the property to a module's property function map is sufficient. Because graph components do not have every property, property functions return a `std::optional` object. This type allows for empty properties and enables, for example, a property value reduction over all graph vertices, even though they implement a heterogeneous set of properties.

2.2.4 COMPONENT TYPES

The component type hierarchy is implemented as a C++ class type hierarchy. All component types inherit from the virtual base class `Component`. Extending the type hierarchy becomes simply declaring a new type with the yloc-provided macro: `YLOC_DECLARE_TYPE(NewComponentType, BaseComponentType1, ...)`, where `(BaseComponentType1, ...)` is the list of base classes in the component type hierarchy. The component types can be used, for example, to filter the topology graph. However, the component type hierarchy, as seen in Figure 3, is still under development.

2.2.5 USE CASES

Listing 3 shows a usage example of the current state of the yloc library. The topology graph, returned by `yloc::root_graph`, is built when the library is initialized (`yloc::init`). The underlying boost graph is exposed to the user application via the member function `boost_graph()`. All available BGL features, for example filtering and the graph algorithms, can be used on the boost graph.

The example shows a filtering of all vertex components of type *GPU* (`g[vd].type->is_a<GPU>`), keeping all edges. Filtering is in general performed by applying an edge predicate and a vertex predicate on the graph. The property *temperature* is queried using `g[vd].get("temperature")`. `yloc::finalize()` shuts down the modules and deallocates internal data structures.

Listing 3: Full yloc usage example of filtering and querying a dynamic property.

```
// main.cpp:
yloc::init(YLOC_FULL); // initialize the library and modules

graph_t &graph = yloc::root_graph();        // yloc topology graph

// filter graph g for GPU components:
auto fgv = boost::make_filtered_graph(
    graph.boost_graph(),                    // exposing the boost graph
    boost::keep_all{},                      // edge predicate keep all
    [&](vertex_descriptor_t &vd) -> bool {  // vertex predicate
        return g[vd].type->is_a<GPU>();     // keep only GPU components
    });

// query a (dynamic) property (temperature) on each GPU component:
for (auto vd : boost::make_iterator_range(boost::vertices(fgv))) {
    auto property = g[vd].get("temperature");
    if (property.has_value()) {
        std::cout << p.value() << '\n';
    }
}

yloc::finalize();
```

Listing 4 shows an example query, determining the number of hops from one vertex to the other vertices in the graph. Such a query is useful, for example, in finding the closest CPU cores to a network interface card, or a GPU (leader pattern).

Listing 4: Yloc example of finding the number of hops to other vertices.

```
std::vector<uint> find_num_hops(graph_t &g, const vertex_descriptor_t &start)
{
    std::vector<uint> hops(boost::num_vertices(g.boost_graph()));

    auto vis = boost::make_bfs_visitor(boost::record_distances(
        boost::make_iterator_property_map(
            hops.begin(),
            boost::get(boost::vertex_index, g.boost_graph())),
        boost::on_tree_edge()));

    boost::breadth_first_search(graph, start, boost::visitor(vis));
    return hops;
}
```

2.3 STATUS AND FUTURE WORK

Using the yloc library in its current state in a meaningful way requires using BGL features in the user application. As a next step, we will implement a query class with generic implementations of the most useful recurring queries, providing application programmers with ways to formulate queries without requiring them to dive into the complexity of the BGL. However, the yloc library can only cover basic queries. The formulation of advanced, complex queries will still require the application programmer to use the BGL in the future.

The current demonstrator module for system-level topology requires domain knowledge of the SuperMUC-NG hostname convention. Without specific domain knowledge of the system-level topology, tools like ibnetdiscover (InfiniBand topology discovery) that require administrator privileges must be used to discover the topology.

An alternative approach would be to use micro benchmarks that measure the bandwidth and latency of the system. However, rerunning micro benchmarks every time is inefficient. A solution would be to execute the benchmarks in advance for the whole system. Afterwards, it would be sufficient to perform lookups instead of rerunning expensive benchmarks. If planned and scheduled accordingly, such a system benchmark could be done as part of maintenance to avoid interference with other applications.

Furthermore, we will integrate yloc with the digital SuperTwin of the SparCity project, and the methodologies for hardware topology aware partitioning in the future.

REFERENCES

Ang, James A et al. “Abstract machine models and proxy architectures for exascale computing”. *2014 Hardware-Software Co-Design for High Performance Computing*. IEEE. 2014, pp. 25–32.

Broquedis, François et al. “hwloc: A generic framework for managing hardware affinities in HPC applications”. *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE. 2010, pp. 180–186.

Goglin, Brice. “Exposing the locality of heterogeneous memory architectures to HPC applications”. *Proceedings of the Second International Symposium on Memory Systems*. 2016, pp. 30–39.

Goglin, Brice, Joshua Hursey, and Jeffrey M Squyres. “Netloc: Towards a comprehensive view of the HPC system topology”. *2014 43rd International Conference on Parallel Processing Workshops*. IEEE. 2014, pp. 216–225.

Treibig, J., G. Hager, and G. Wellein. "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments". *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*. 2010.