P1795r2: System topology discovery for heterogeneous & distributed computing

Date: 2020-01-10

Audience: SG1, SG14

Authors: Gordon Brown, Ruyman Reyes, Michael Wong, Mark Hoemmen, Jeff Hammond, Tom Scogland, Domagoj Šarić

Emails: gordon@codeplay.com, ruyman@codeplay.com, michael@codeplay.com, mhoemme@sandia.gov, jeff.science@gmail.com, tscogland@llnl.gov, domagoj.saric@microblink.com

Reply to: gordon@codeplay.com

Acknowledgements

This paper is the result of discussions from many contributors within SG1, SG14 and the heterogeneous C++ group, including H. Carter Edwards, Thomas Rodgers, Patrice Roy, Carl Cook, Jeff Hammond, Hartmut Kaiser, Christian Trott, Paul Blinzer, Alex Voicu, Nat Goodspeed and Tony Tye.

Changelog

P1437r2 (PRA 2020)

- Update the proposed direction to be more specific regarding the approach.
- Introduce a pseudo list of executor properties that are expected to be useful to applications utilizing topology discovery.

P1437r1 (BEL 2019)

- Introduce terms of art for *system topology*, *system resource* and *topology traversal policy*.
- Introduce minimal design for `system_topology` class.
- Introduce minimal design for `system_resource` class.
- Introduce free function `this_system::discover_topology` for performing runtime system topology discovery.
- Introduce free function `traverse_topology` for traversing a `system_topology` using a *topology traversal policy* to return a collection of `execution_resource`s,

P1437r0 (COL 2019)

- Split off from [17], focussing on a mechanism for discovering the topology and affinity properties of a given system.
- Temporarily remove the proposed wording.
- Update the front matter to re-focus the motivation and goals of the paper.

Changelog from P0796

For the earlier changelogs from prior to the split from P0796 see Appendix A.

Preface

This paper is the result of a request from SG1 at the 2018 San Diego meeting to split [17] into two separate papers, one for the high-level interface and one for the low-level interface. This paper focusses on the low-level interface; a mechanism for discovering the topology and affinity properties of a given system. [18] focusses on the high-level interface, a series of properties for querying affinity relationships and requesting affinity on work being executed.

1. Background

Computer systems are no longer homogeneous platforms. From desktop workstations to high-performance supercomputers, and from mobile devices to purpose-built embedded SoCs, every system has some form of co-processor along side the traditional multi-core CPU, and often more than one. Furthermore, the architectures of these co-processors range from many-core CPUs, GPUs, FPGAs and DSPs to specifically designed vision and machine learning processors. In larger supercomputer systems there are thousands of these processors in some configuration of nodes, connected physically or via network adapters.

The way these processors access memory is also far from homogeneous. For example, the system may present a single shared virtual address space [21] [22], or it may have different address spaces mutually inaccessible other than through special functions [4]. Different memory regions may have different levels of consistency, cache coherency, and support for atomic operations. Different parts of the system may have different access latencies or bandwidths to different memory regions (so-called "NUMA affinity regions") [2]. Some parts of memory may be persistent. Different systems may configure the same types of memory in different ways around the processors.

In order to program these new systems and the architectures that inhabit them, it's vital that applications are capable of understating both what architectures are available and the properties of those architectures, namely their observable behaviors, capabilities and limitations. However, the current C++ standard provides no way to achieve this, so developers have to rely entirely on third party and operating system libraries.

2. Goals: what this paper is, and what it is not

This paper seeks to define, within C++, a facility for discovering execution resources available to a system that are capable of executing work, and for querying their properties.

However, it is not the goal of this proposal to introduce support in the C++ language or the standard library for all of the various heterogeneous architectures available today. The authors of this paper recognize that this is unrealistic as it would require significant changes to the C++ machine model and would be extremely volatile to future developments in architecture and system design.

Instead, it seeks to define a single, unified, and stable layer in the C++ Standard Library. Applications, libraries, and programming models (such as SYCL [3], Kokkos [19], HPX [13] or TBB [12]) can build on this layer;

hardware vendors can support it via standards such as OpenCL [4], CUDA [20], OpenMP [6], MPI [16], Hwloc [2], HSA [5] and HMM [21]; and it can be extended when necessary.

This layer will not be characterized in terms of specific categories of hardware such as CPUs, GPUs and FPGAs as these are broad concepts that are subject to change over time and have no foundation in the C++ machine model. It will instead define a number of abstract properties of system architectures that are not tied to any specific hardware.

The initial set of properties that this paper would propose be defined in the C++ standard library would reflect a generalization of the observable behaviors, capabilities and limitations of common architectures available in heterogeneous and distributed systems today. However the intention is that the interface be extensible so that that vendors can provide their own extensions to provide visibility into the more niche characteristics of certain architectures.

It is intended that this layer be defined as a natural extension of the Executors proposal, a unified interface for execution. The current executors proposal [14] already provides a route to supporting heterogeneous and distributed systems, however it is missing a way to identify what architectures a system has.

3. Motivation

There are many reasons why such a feature within C++ would benefit developers and the C++ ecosystem as a whole, and those can differ from one domain to another. We've attempted to outline some of these benefits here.

Improve performance

The clearest benefit is performance. Exposing, even at an abstract level, the properties of the underlying architecture that a program is running on, allows application and libraries to be fine tuned. This may result in significant performance improvements that would only otherwise be possible via third party or operating system libraries [1] [7] [8] [9] [10] [11] [15].

This includes but is not limited to how to structure data to ensure access patterns along with execution on the architecture to achieve coalesced memory access and optimal cache utilization and where to initialize data to make efficient use of hardware locality and process affinity.

There is a general trend to move towards a unified address space in heterogeneous and distributed systems via standards like HMM. However, there are still many architectures that still require distinct address spaces, are not yet in a position to move to a single address space, and may never be. Even if you were to consider a single unified address the ultimate goal for heterogeneous and distributed systems, this actually makes the case for affinity in C++ stronger. As long as different address spaces exist, the distinction between different hardware memory regions and their capabilities is clear, but with a single unified address space, potentially with cache coherency, distinguishing different memory regions becomes much more subtle. Therefore, it becomes much more important to understand the various memory regions and their affinity relationships in order to achieve good performance on various architectures.

Provide a unified interface

C++ is a major language when it comes to heterogeneous and distributed computing, and while it is a rapidly growing domain, it is still very challenging to develop in. There are a large number of C++ based third party and OS libraries. However, developing for heterogeneous and distributed systems often involves a combination of these libraries, which introduces a number of challenges

Firstly it's common that different architectures are discovered via different libraries, You may want to use CUDA for NVidia GPUs, OpenMP for Intel CPUs, SYCL for Intel GPUs, Hwloc for the higher-level nodes and so on. This means that you have to collect together resources discovered from a different libraries, which very often do not provide a consistent representation or any form of interoperability, and find some way for them to represent them in a coherent view.

Secondly, many of these libraries report the same underlying hardware. For example OpenMP, SYCL and Hwloc will all report the same Intel CPU. This means you have to collate the resources together such that resources from different libraries representing the same hardware are joined together, to avoid resource contention.

Categorize limitations

There are many architectures available within heterogeneous and distributed systems which cannot support the full range of C++ features. This includes, but is not limited to dynamic allocation, recursion, dynamic polymorphism, RTTI, double precision floating point and some forms of atomic operations.

It's crucial to allow developers to identify these limitations and which apply to the architecture they are running on, because in many cases if a C++ feature that is not supported on the architecture is used, the application would fail to execute or potentially crash.

Facilitate generic code

Developing algorithms for heterogeneous and distributed systems requires at least an abstract understanding of the underlying architecture(s) being targeted in order to achieve optimal performance. In some cases, it may call for a much more in-depth understanding. This means that each algorithm may require a different implementation on each architecture.

Another factor here is that in many heterogeneous and distributed programming models, the architectures available on a particular system are not known until runtime, where the topology of the system is discovered.

Having a unified interface for performing this topology discovery and querying the properties of the architectures available on a system would dramatically improve developers' ability to write generic algorithms.

Increase accessibility

Providing support for heterogenous and distributed computing as a first-class citizen of C++ will improve its accessibility and increase its utilization in libraries and applications, ultimately making the ecosystem stronger. This will become increasingly more important as heterogeneous and distributed computing becomes crucial to gaining the necessary performance in applications in more domains of C++.

Provide a broader standardization

The C++ standard is in a crucial position for heterogeneous and distributed computing domains. It is the common point between a number of different programming languages, models and libraries targeting a wide

range of different architectures. This means that C++ has a unique opportunity to provide a single standard that not only covers the requirements of a single domain, but all of them, allowing for a convergence within the ecosystem and much more interoperability across different architectures.

For example, a unified C++ interface for topology discovery could provide access to GPUs from Nvidia, AMD, Intel, and ARM via their respective open standards or proprietary frameworks. At the same time, it could give access to NUMA-aware systems via Hwloc.

Another example of this is that while Hwloc is highly used in many domains, it now does not always accurately represent existing systems. This is because Hwloc presents their topology as strictly hierarchical, which no longer accurately describes many systems. A unified C++ interface does not need to be bound to the limitations of a single library, and can provide a much broader representation of a system's execution resource topology.

5. Proposed direction

Overview

This paper aims to build on the unified executors proposal, detailed in P0443 [14], so this proposal and any others that stem from it will target P0443 as a baseline, and aim to integrate with its direction as closely as possible.

This paper proposes an interface which provides an abstract representation of a system's resources and their connections, and various properties, capabilities and limitations. This abstract representation will allow a diverse range of architectures both available now and to come in the future to be represented in C++, but without tying the C++ abstract machine to specific hardware definitions like "CPU" or "GPU".

The abstract representation will be an opaque representation of execution, memory, network and I/O resources and their connections to each other which can be traversed in a number of different ways using topology traversal policies, such as a containment hierarchy view, a memory-centric view or a network-centric view.

The idea is that the C++ standard remains abstract and generic with simply a few standard topology traversal policies, while domains can create topology traversal policies which provide ways of discovering resources unique to their domain. For example a GPU vendor such as NVidia may want to define a policy which recognizes their GPUs or a mobile or embedded platform vendor such as ARM may want to define a policy which recognizes unique SoC architectures such as ARM bit little.

This proposal will also propose a mechanism, likely extending the properties mechanism of the unified executors proposal, for querying properties, capabilities and limitations of the various resources within a system topology and their connections with each other.

Finally this proposal will also propose an interface for creating creating executors and allocators or memory resources from a collection of resources discovered within the system topology.

Designed by example

The ultimate goal for this proposal is to allow algorithm implementors the ability to author algorithms that are portable across a wide range of architectures and systems, without them having to know the architecture it's

running on at all, and being able to operate solely in terms of an abstract system topology representation and it's properties.

Therefore this proposal will take the approach of designing the interfaces described above using examples to test the suitability of the approach.

Properties

The first step in this is to identify the kinds of properties that various different kinds of systems and architectures would require in order to identify the kind of architecture of the system and optimise the algorithm for it. To this end this proposal proposes a pseudo list of properties that are expected to be or would like to be proposed in some form.

For each resource within the system topology it would be beneficial to query:

- Resource type, such as execution resource, memory resource, I/O resource, network resource, etc.
- Connections it has with other resources (including multiple connections to the same resource).
- Available concurrency.
- Ability to be partitioned into other resources, and the possible levels of granularity of the partitioning.
- Support for SIMD execution and the available SIMD ABIs and widths.
- Hardware concurrency (mapped to the existing C++ function fo the same name).
- Hardware constructive interference size (mapped to the existing C++ function fo the same name).
- Hardware destructive interference size (mapped to the existing C++ function fo the same name).
- Preferred bulk execution shape.
- Maximum bulk execution shape.
- Available memory.
- Preferred memory allocation multiples.
- Available affinity patterns.
- Available work subdivision patterns.
- Support for exceptions.

For connections between resources within the system topology it would beneficial to query:

- The type of connection, such as PCIe, DMA, etc.
- Whether the connection can be used to access data.
- The difference in depth, to represent hierarchical topologies.
- The latency of the connection.
- The bandwidth of the connection.

For groups of resources within the system topology it would beneficial to query:

- Support for pinned or shared memory.
- Ability to composed a shared executor.

> [*Note:* This proposal is not proposing these properties, it is simply identifying a list of queries that would be useful in some form. --*end note*]

6. Proposal

Header <system> synopsis

```
namespace experimental {

/* system_topology */

class system_topology {

  system_topology() = delete;

};

/* system_resource */

class system_resource {

  /* to be defined */

};

/* traverse_topology */

template <class T>
to-be-decided<system_resource> traverse_topology(const system_topology &, const T
&) noexcept;

/* this_system::discover_topology */

namespace this_system {

system_topology discover_topology();

}  // namespace this_system

}  // experimental
```

Terms of art

The term *system resource* refers to a hardware or software abstraction of an execution, memory, network or I/O resource within a system.

The term *system topology* refers to a possibly cyclic graph of *execution resources* connected to the abstract machine, and their various properties.

> [*Note:* The current definition of *system topology* is currently incomplete and will be developed over the course of this proposal as the various C++ domains are represented. *--end note*]

The term *topology traversal policy* refers to a policy that describes the way in which a *system topology* is traversed in order to to produce a collection of *system resources*.

Class `system_topology`

The `system_topology` class provides an abstraction of a read-only snapshot of the *system topology* at a particular point in time. A `system_topology` object may not maintain or otherwise be associated with the lifetime of operating system or third party library resources.

`system_topology` constructors

```
system_topology() = delete;
```

Effects: Explicitly deleted.

Class `system_resource`

The `system_resource` class provides an abstraction of a read-only snapshot of a *system resource* from the *system topology* at a particular point in time. A `system_resource` object may not maintain or otherwise be associated with the lifetime of operating system or third party library resources.

> [*Note:* The `system_resource` class is intended to reflect the properties of a *system resource* and it's relationships with other *system resources*, however the precise definition is still to be decided. *--end note*]

Free functions

`this_system::discover_topology`

The free function `this_system::discover_topology` performs runtime discovery of the *system topology* and returns a `system_topology` object.

```
namespace this_system {
  system_topology discover_topology();
}  // namespace this_system
```

Returns: A `system_topology` object representing a snapshot of the *system topology* at the current point in time.

Requires: Calls to `this_system::discover_topology()` may not introduce a data race with any other call to `this_system::discover_topology()`.

Effects: Performs runtime discovery of the system topology and constructs a `system_topology` object. May invoke the operating system or third party libraries in discovering topology information, but must release any resources acquired for this purpose before returning.

Throws: Any exception thrown as a result of performing runtime discovery of the system topology.

`traverse_topology`

The free function `traverse_topology` performs a traversal of a `system_topology` object using a *topology traversal policy* specified by the tag type `T` and returns a sequence of `system_resource` objects.

```
template <class T>
to-be-decided<system_resource> traverse_topology(const system_topology &, const T
&) noexcept;
```

Returns: A sequence of `system_resource` objects representing the *system resources* matching the criteria of the *topology traversal policy*.

Effects: Traverses the `system_topology` object provided and identifies any *system resources* which match the criteria of the *topology traversal policy*, adding a single `system_resource` to the sequence returned for each match found.

Throws: May not throw.

> [*Note:* The exact representation of *system resources* returned by `traverse_topology` is still to be decided as this will have implications on lifetimes. One option is to return a container of *system_resource* objects by-value such as a `vector`, however this would require some form of reference counting. Another option is to return a reference to a reference to the *system_resource* objects via a `span` or a `ranges::view`, however this would require the `system_topology` object to remain alive. -- *end note*]

7. Open questions

- How granular should topology discovery be, should they whole topology be discovered in a single operation or should it be done in multiple nested operations, only discovering what is needed at each layer?
- What kind of *topology traversal policies* would people list to see standardized?
- How should we support notification of a topology update, polling or callback?
- Should we also provide an interface for compile-time topology discovery?

References

[1] The Design of OpenMP Thread Affinity

[2] Portable Hardware Locality

[3] SYCL 1.2.1

[4] OpenCL 2.2

[5] HSA

[6] OpenMP 5.0

[7] cpuaff

[8] MEMKIND

[9] Solaris pbind()

[10] Linux sched_setaffinity()

[11] Windows SetThreadAffinityMask()

[12] TBB

[13] HPX

[14] A Unified Executors Proposal for C++

[15] Exposing the Locality of new Memory Hierarchies to HPC Applications

[16] MPI

[17] Supporting Heterogeneous & Distributed Computing Through Affinity

[18] Executor properties for affinity-based execution

[19] Kokkos project

[20] CUDA

[21] Heterogeneous Memory Management

[22] OpenCL 2.x Shared Virtual Memory

Appendix A: Changelog from P0796

P0796r3 (SAN 2018)

- Remove reference counting requirement from `execution_resource`.
- Change lifetime model of `execution_resource`: it now either consistently identifies some underlying resource, or is invalid; context creation rejects an invalid resource.ster
- Remove `this_thread::bind` & `this_thread::unbind` interfaces.
- Make `execution_resource`s iterable by replacing `execution_resource::resources` with `execution_resource::begin` and `execution_resource::end`.
- Add `size` and `operator[]` for `execution_resource`.
- Rename `this_system::get_resources` to `this_system::discover_topology`.
- Introduce `memory_resource` to represent the memory component of a system topology.
- Remove `can_place_memory` and `can_place_agents` from the `execution_resource` as these are no longer required.
- Remove `memory_resource` and `allocator` from the `execution_context` as these no longer make sense.
- Update the wording to describe how execution resources and memory resources are structured.
- Refactor `affinity_query` to be between an `execution_resource` and a `memory_resource`.

P0796r2 (RAP 2018)

- Introduce a free function for retrieving the execution resource underlying the current thread of execution.
- Introduce `this_thread::bind` & `this_thread::unbind` for binding and unbinding a thread of execution to an execution resource.
- Introduce `bulk_execution_affinity` executor properties for specifying affinity binding patterns on bulk execution functions.

P0796r1 (JAX 2018)

- Introduce proposed wording.
- Based on feedback from SG1, introduce a pair-wise interface for querying the relative affinity between execution resources.
- Introduce an interface for retrieving an allocator or polymorphic memory resource.
- Based on feedback from SG1, remove requirement for a hierarchical system topology structure, which doesn't require a root resource.

P0796r0 (ABQ 2017)

- Initial proposal.
- Enumerate design space, hierarchical affinity, issues to the committee.