P0796r3: Supporting Heterogeneous & Distributed Computing Through Affinity

Date: 2018-10-08

Audience: SG1, LEWG, LWG, SG14

Authors: Gordon Brown, Ruyman Reyes, Michael Wong, H. Carter Edwards, Thomas Rodgers, Mark Hoemmen

Contributors: Patrice Roy, Carl Cook, Jeff Hammond, Hartmut Kaiser, Christian Trott, Paul Blinzer, Alex Voicu, Nat Goodspeed, Tony Tye, Paul Blinzer

Emails: gordon@codeplay.com, ruyman@codeplay.com, michael@codeplay.com, hedwards@nvidia.com, rodgert@twrodgers.com, mhoemme@sandia.gov

Reply to: gordon@codeplay.com

Changelog

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.

Abstract

This paper provides an initial meta-framework for the drives toward an execution and memory affinity model for C++. It accounts for feedback from the Toronto 2017 SG1 meeting on Data Movement in C++ [1] that we should define affinity for C++ first, before considering inaccessible memory as a solution to the separate memory problem towards supporting heterogeneous and distributed computing.

This paper is split into two main parts:

1. A series of executor properties which can be used to apply affinity requirements to bulk execution functions.
2. An interface for discovering the execution resources within the system topology and querying relative affinity of execution resources.

Motivation

Affinity refers to the "closeness" in terms of memory access performance, between running code, the hardware execution resource on which the code runs, and the data that the code accesses. A hardware execution resource has "more affinity" to a part of memory or to some data, if it has lower latency and/or higher bandwidth when accessing that memory / those data.

On almost all computer architectures, the cost of accessing different data may differ. Most computers have caches that are associated with specific processing units. If the operating system moves a thread or process from one processing unit to another, the thread or process will no longer have data in its new cache that it had in its old cache. This may make the next access to those data slower. Many computers also have a Non-Uniform Memory Architecture (NUMA), which means that even though all processing units see a single memory in terms of programming model, different processing units may still have more affinity to some parts of memory than others. NUMA exists because it is difficult to scale non-NUMA memory systems to the performance needed by today's highly parallel computers and applications.

One strategy to improve applications' performance, given the importance of affinity, is processor and memory *binding*. Keeping a process bound to a specific thread and local memory region optimizes cache affinity. It also reduces context switching and unnecessary scheduler activity. Since memory accesses to remote locations incur higher latency and/or lower bandwidth, control of thread placement to enforce affinity within parallel applications is crucial to fuel all the cores and to exploit the full performance of the memory subsystem on NUMA computers.

Operating systems (OSes) traditionally take responsibility for assigning threads or processes to run on processing units. However, OSes may use high-level policies for this assignment that do not necessarily match the optimal usage pattern for a given application. Application developers must leverage the placement of memory and *placement of threads* for best performance on current and future architectures. For C++ developers to achieve this, native support for *placement of threads and memory* is critical for application portability. We will refer to this as the *affinity problem*.

The affinity problem is especially challenging for applications whose behavior changes over time or is hard to predict, or when different applications interfere with each other's performance. Today, most OSes already can group processing units according to their locality and distribute processes, while keeping threads close to the initial thread, or even avoid migrating threads and maintain first touch policy. Nevertheless, most programs can change their work distribution, especially in the presence of nested parallelism.

Frequently, data are initialized at the beginning of the program by the initial thread and are used by multiple threads. While some OSes automatically migrate threads or data for better affinity, migration may have high overhead. In an optimal case, the OS may automatically detect which thread access which data most frequently, or it may replicate data which are read by multiple threads, or migrate data which are modified and used by threads residing on remote locality groups. However, the OS often does a reasonable job, if the machine is not overloaded, if the application carefully uses first-touch allocation, and if the program does not change its behavior with respect to locality.

Consider a code example *(Listing 1)* that uses the C++17 parallel STL algorithm `for_each` to modify the entries of a `valarray a`. The example applies a loop body in a lambda to each entry of the `valarray a`, using an execution policy that distributes work in parallel across multiple CPU cores. We might expect this to be fast, but since `valarray` containers are initialized automatically and automatically allocated on the master thread's memory, we find that it is actually quite slow even when we have more than one thread.

```
// C++ valarray STL containers are initialized automatically.
// First-touch allocation thus places all of a on the master.
std::valarray<double> a(N);

// Data placement is wrong, so parallel update is slow.
std::for_each(std::execution::par, std::begin(a), std::end(a),
              [=] (double& a_i) { a_i *= scalar; });

// Use future affinity interface to migrate data at next
// use and move pages closer to next accessing thread.
...
// Faster, because data are local now.
std::for_each(std::execution::par, std::begin(a), std::end(a),
              [=] (double& a_i) { a_i *= scalar; });
```

Listing 1: Parallel vector update example

The affinity interface we propose should help computers achieve a much higher fraction of peak memory bandwidth when using parallel algorithms. In the future, we plan to extend this to heterogeneous and distributed computing. This follows the lead of OpenMP [2], which has plans to integrate its affinity model with its heterogeneous model [3]. (One of the authors of this document participated in the design of OpenMP's affinity model.)

Background Research: State of the Art

The problem of effectively partitioning a system's topology has existed for some time, and there are a range of third-party libraries and standards which provide APIs to solve the problem. In order to standardize this process for C++, we must carefully look at all of these approaches and identify which we wish to adopt. Below is a list of the libraries and standards from which this proposal will draw:

- Portable Hardware Locality [4]
- SYCL 1.2 [5]
- OpenCL 2.2 [6]
- HSA [7]
- OpenMP 5.0 [8]
- cpuaff [9]
- Persistent Memory Programming [10]
- MEMKIND [11]
- Solaris pbind() [12]
- Linux sched_setaffinity() [13]
- Windows SetThreadAffinityMask() [14]
- Chapel [15]
- X10 [16]
- UPC++ [17]
- TBB [18]
- HPX [19]
- MADNESS [20][32]

Libraries such as the Portable Hardware Locality (hwloc) library provide a low level of hardware abstraction, and offer a solution for the portability problem by supporting many platforms and operating systems. This and similar approaches use a tree structure to represent details of CPUs and the memory system. However, even some current systems cannot be represented correctly by a tree, if the number of hops between two sockets varies between socket pairs [2].

Some systems give additional user control through explicit binding of threads to processors through environment variables consumed by various compilers, system commands, or system calls. Examples of system commands include Linux's `taskset` and `numactl`, and Windows' `start /affinity`. System call examples include Solaris' `pbind()`, Linux's `sched_setaffinity()`, and Windows' `SetThreadAffinityMask()`.

Problem Space

In this paper we describe the problem space of affinity for C++, the various challenges which need to be addressed in defining a partitioning and affinity interface for C++, and some suggested solutions. These include:

- How to represent, identify and navigate the topology of execution resources available within a heterogeneous or distributed system.
- How to query and measure the relative affinity between different execution resources within a system.
- How to bind execution and allocation particular execution resource(s).
- What kind of and level of interface(s) should be provided by C++ for affinity.

Wherever possible, we also evaluate how an affinity-based solution could be scaled to support both distributed and heterogeneous systems. We also have addressed some aspects of dynamic topology discovery.

There are also some additional challenges which we have been investigating but are not yet ready to be included in this paper, and which will be presented in a future paper:

- How to migrate memory work and memory allocations between execution resources.
- More general cases of dynamic topology discovery.
- Fault tolerance, as it relates to dynamic topology.

Querying and representing the system topology

The first task in allowing C++ applications to leverage memory locality is to provide the ability to query a *system* for its *resource topology* (commonly represented as a tree or graph) and traverse its *execution resources*.

The capability of querying underlying *execution resources* of a given *system* is particularly important towards supporting affinity control in C++. The current proposal for executors [22] mentions execution resources in passing, but leaves the term largely unspecified. This is intentional: *execution resources* will vary greatly between one implementation and another, and it is out of the scope of the current executors proposal to define those. There is current work [23] on extending the executors proposal to describe a typical interface for an *execution context*. In this paper a typical *execution context* is defined with an interface for construction and comparison, and for retrieving an *executor*, waiting on submitted work to complete and querying the underlying *execution resource*. Extending the executors interface to provide topology information can serve as a basis for providing a unified interface to expose affinity. This interface cannot mandate a specific architectural definition, and must be generic enough that future architectural evolutions can still be expressed.

Two important considerations when defining a unified interface for querying the *resource topology* of a *system*, are (a) what level of abstraction such an interface should have, and (b) at what granularity it should describe the typology's *execution resources*. As both the level of abstraction of an *execution resource* and the granularity that it is described in will vary greatly from one implementation to another, it’s important for the interface to be generic enough to support any level of abstraction. To achieve this we propose a generic hierarchical structure of *execution resources*, each *execution resource* being composed of other *execution resources* recursively. Each *execution resource* within this hierarchy can be used to place memory (i.e., allocate memory within the *execution resource’s* memory region), place execution (i.e. bind an execution to an *execution resource’s execution agents*), or both.

For example, a NUMA system will likely have a hierarchy of nodes, each capable of placing memory and placing agents. A system with both CPUs and GPUs (programmable graphics processing units) may have GPU local memory regions capable of placing memory, but not capable of placing agents.

Nowadays, there are various APIs and libraries that enable this functionality. One of the most commonly used is Portable Hardware Locality (hwloc). Hwloc presents the hardware as a tree, where the root node represents the whole machine and subsequent levels represent different partitions depending on different hardware characteristics. The picture below shows the output of the hwloc visualization tool (lstopo) on a 2-socket Xeon E5300 server. Note that each socket is represented by a package in the graph. Each socket contains its own cache memories, but both share the same NUMA memory region. Note also that different I/O units are visible underneath. Placement of these I/O units with respect to memory and threads can be critical to performance. The ability to place threads and/or allocate memory appropriately on the different components of this system is an important part of the process of application development, especially as hardware architectures get more complex. The documentation of lstopo [21] shows more interesting examples of topologies that appear on today's systems.

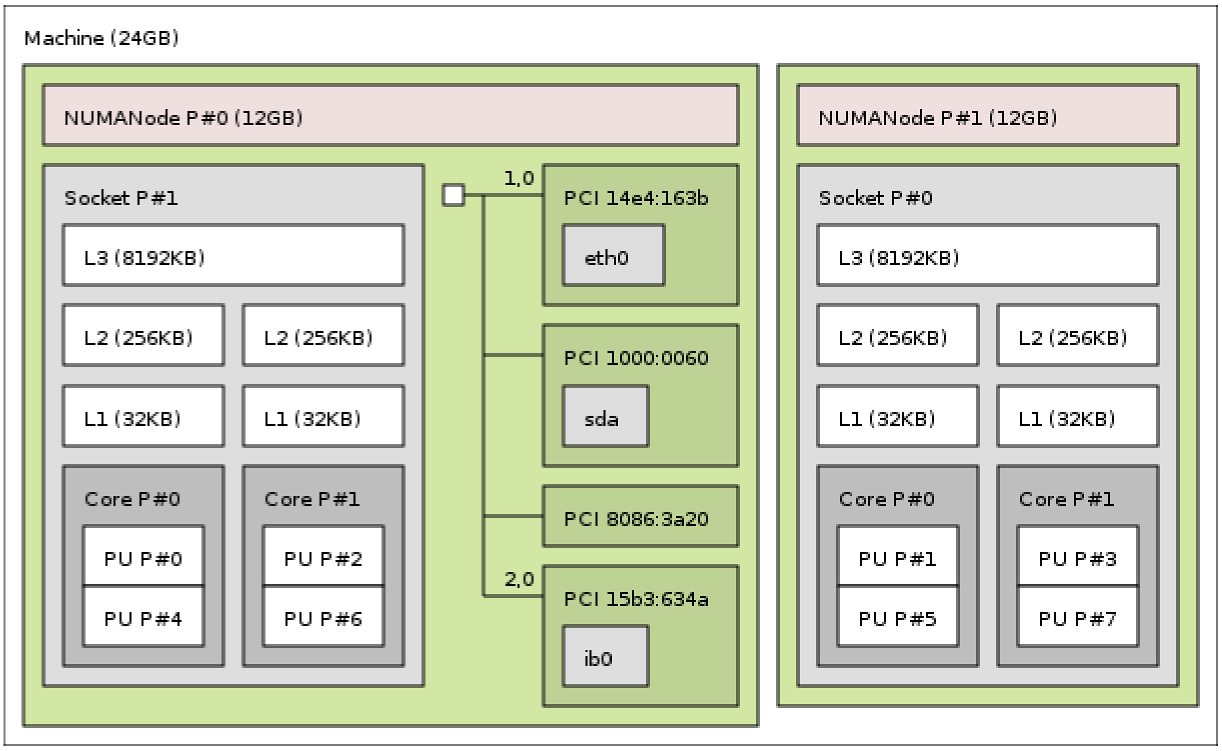

Figure 1: Example of Hwloc system topology report

The interface of `thread_execution_resource_t` proposed in the execution context proposal [23] proposes a hierarchical approach where there is a root resource and each resource has a number of child resources. However, systems are becoming increasingly non-hierarchical and a traditional tree-based representation of a *system's resource topology* may not suffice any more [24]. The HSA standard solves this problem by allowing a node in the topology to have multiple parent nodes [19].

The interface for querying the *resource topology* of a *system* must be flexible enough to allow querying all *execution resources* available under an *execution context*, querying the *execution resources* available to the entire system, and constructing an *execution context* for a particular *execution resource*. This is important, as many standards such as OpenCL [6] and HSA [7] require the ability to query the *resource topology* available in a *system* before constructing an *execution context* for executing work.

> For example, an implementation may provide an execution context for a particular execution resource such as a static thread pool or a GPU context for a particular GPU device, or an implementation may provide a more generic execution context which can be constructed from a number of CPU and GPU devices query-able through the system resource topology.

Topology discovery & fault tolerance

In traditional single-CPU systems, users may reason about the execution resources with standard constructs such as `std::thread`, `std::this_thread` and `thread_local`. This is because the C++ machine model requires that a system have **at least one thread of execution, some memory, and some I/O capabilities**. Thus, for these systems, users may make some assumptions about the system resource topology as part of the language and its supporting standard library. For example, one may always ask for the available hardware concurrency, since there is always at least one thread, and one may always use thread-local storage.

This assumption, however, does not hold on newer, more complex systems, especially on heterogeneous systems. On these systems, even the type and number of high-level resources available in a particular *system* is not known until the physical hardware attached to a particular system has been identified by the program. This often happens as part of a run-time initialization API [6] [7] which makes the resources available through some software abstraction. Furthermore, the resources which are identified often have different levels of parallel and concurrent execution capabilities. We refer to this process of identifying resources and their capabilities as *topology discovery*, and we call the point at the point at which this occurs the *point of discovery*.

An interesting question which arises here is whether the *system resource topology* should be fixed at the *point of discovery*, or whether it should be allowed to change during later program execution. We can identify two main reasons for allowing the *system resource topology* to be dynamic after the *point of discovery*: (a) *dynamic resource discovery*, and (b) *fault tolerance*.

In some systems, hardware can be attached to the system while the program is executing. For example, users may plug in a USB-compute device [31] while the application is running to add additional computational power, or users may have access to hardware connected over a network, but only at specific times. Support for *dynamic resource discovery* would let programs target these situations natively and be reactive to changes to the resources available to a system.

Other applications, such as those designed for safety-critical environments, must be able to recover from hardware failures. This requires that the resources available within a system can be queried and can be expected to change at any point during the execution of a program. For example, a GPU may overheat and need to be disabled, yet the program must continue at all costs. *Fault tolerance* would let programs query the availability of resources and handle failures. This could facilitate reliable programming of heterogeneous and distributed systems.

From a historic perspective, programming models for traditional high-performance computing (HPC) have taken different approaches to *dynamic resource discovery*. MPI (Message Passing Interface) [25] originally (in MPI-1) did not support *dynamic resource discovery*. All processes which were capable of communicating with each other would be identified and fixed at the *point of discovery*, which (from the programmer's perspective) is `MPI_Init`. PVM (Parallel Virtual Machine) [26] enabled resources to be discovered at run time, using an alternative execution model of manually spawning processes from the main process. This led MPI-2 to introduce the feature. However, MPI programs do not commonly use this feature, and generally prefer the execution model of having all processes fixed at initialization. Some distributed-memory parallel programming models for HPC support dynamic process spawning, but the typical way that HPC users access large-scale computing resources requires fixed-size batch allocations that restrict truly dynamic process spawning.

Some of these programming models also address *fault tolerance*. In particular, PVM has native support for this, providing a mechanism [27] which can notify a program when a resource is added or removed from a system. MPI lacks a native *fault tolerance* mechanism, but there have been efforts to implement fault tolerance on top of MPI [28] or by extensions[29].

Due to the complexity involved in standardizing *dynamic resource discovery* and *fault tolerance*, these are currently out of the scope of this paper. However, we leave open the possibility of accommodating both in the future, by not over constraining *resources*' lifetimes (see next section).

Reporting errors in topology discovery

As querying the topology of a system can invoke a number of different system and third-party library, we have to consider what will happen when a call to one of these fails. Firstly we want to be able to report this failure so that it can be reported or handled in user code. Secondly as there will often be more than one source of topology discovery we have to avoid short-circuiting the discovery on an error and preventing potentially valid topology information being reported to users. For example if a system were to report both Hwloc and OpenCL execution resources and one of these failed we want the other to still be able to return it's resources.

A potential solution to this could be support partial errors in topology discovery, where querying the system for it's topology could be permitted to fail but still return a valid topology structure representing the topology that was discovered successfully. The way in which these errors are reported (i.e. exceptions or error values) would have to be decided, exceptions could be problematic as it could unwind the stack before capturing important topology information so perhaps an error value based approach would be preferable.

Resource lifetime

The initial solution may only target systems with a single addressable memory region. It may thus exclude devices like discrete GPUs. However, in order to maintain a unified interface going forward, the initial solution should consider these devices and be able to scale to support them in the future. In particular, in order to support heterogeneous systems, the abstraction must let the interface query the *resource topology* of the *system* in order to perform device discovery.

The *resource* objects returned from the *topology discovery interface* are opaque, implementation-defined objects. They would not perform any scheduling or execution functionality which would be expected from an *execution context*, and they would not store any state related to an execution. Instead, they would simply act as an identifier to a particular partition of the *resource topology*. This means that the lifetime of a *resource* retrieved from an *execution context* must not be tied to the lifetime of that *execution context*.

The lifetime of a *resource* instance refers to both validity and uniqueness. First, if a *resource* instance exists, does it point to a valid underlying hardware or software resource? That is, could an instance's validity ever change at run time? Second, could a *resource* instance ever point to a different (but still valid) underlying resource? It suffices for now to define "point to a valid underlying resource" informally. We will elaborate this idea later in this proposal.

Creation of a *context* expresses intent to use the *resource*, not just to view it as part of the *resource topology*. Thus, if a *resource* could ever cease to point to a valid underlying resource, then users must not be allowed to create a *context* from the resource instance, or launch executions with that context. *Context* construction, and use of an *executor* with that *context* to launch an execution, both assert validity of the *context*'s *resource*.

If a *resource* is valid, then it must always point to the same underlying thing. For example, a *resource* cannot first point to one CPU core, and then suddenly point to a different CPU core. *Contexts* can thus rely on properties like binding of operating system threads to CPU cores. However, the "thing" to which a *resource* points may be a dynamic, possibly software-managed pool of hardware. Here are three examples of this phenomenon:

1. The "hardware" may actually be a virtual machine (VM). At any point, the VM may pause, migrate to different physical hardware, and resume. If the VM presents the same virtual hardware before and after the migration, then the *resources* that an application running on the VM sees should not change.
2. The OS may maintain a pool of a varying number of CPU cores as a shared resource among different user-level processes. When a process stops using the resource, the OS may reclaim cores. It may make sense to present this pool as an *execution resource*.
3. A low-level device driver on a laptop may switch between a "discrete" GPU and an "integrated" GPU, depending on utilization and power constraints. If the two GPUs have the same instruction set and can access the same memory, it may make sense to present them as a "virtualized" single *execution resource*.

In summary, a *resource* either identifies a thing uniquely, or harmlessly points to nothing. The section that follows will justify and explain this.

Permit dynamic resource lifetime

We should not assume that *resource* instances have the same lifetime as the running application. For example, some hardware accelerators like GPUs require calling an initialization function before a running application may use the accelerator, and calling a finalization function after using the accelerator. The software interface for the accelerator may not even be available at application launch time. For instance, the interface may live in a dynamic library that users may load at run time. In the case of a pool of CPU cores managed by the operating system, the application might have to request access to the pool at run time, and the operating system may have to do some work in order to reserve CPU cores and set them up for use in the pool. Applications that do not use the pool should not have to pay this setup cost. The more general cases of dynamic resource discovery and fault tolerance, that we discussed above, also call for dynamic *resource* lifetimes.

Resources should not reference count

We considered mandating that *execution resources* use reference counting, just like `shared_ptr`. This would clearly define resources' lifetimes. However, there are several arguments against requiring reference counting.

1. Holding a reference to the *execution resource* would prevent execution from shutting down, thus (potentially) deadlocking the program.
2. Not all kinds of *resources* may have lifetimes that fit reference counting semantics. Some kinds of GPU *resources* only exist during execution, for example; those *resources* cannot be valid if they escape the scope of code that executes on the GPU. In general, programming models that let a "host" processor launch code on a "different processor" have this issue.
3. Reference counting could have unattractive overhead if accessed concurrently, especially if code wants to traverse a particular subset of the *resource topology* inside a region executing on the GPU (e.g., to access GPU scratch memory).
4. Since users can construct arbitrary data structures from *resources* in a *resource hierarchy*, the proposal would need another *resource* type analogous to `weak_ptr`, in order to avoid circular dependencies that could prevent releasing *resources*.
5. There is no type currently in the Standard that has reference-counting semantics, but does not have `shared_` in its name (e.g., `shared_ptr` and `shared_future`). Adding a type like this sets a bad precedent for types with hidden costs and correctness issues (see (4)).

What does validity of a resource mean?

Here, we elaborate on what it means for a *resource* to be "valid." This proposal lets users encounter a *resource* either while traversing the *resource topology*, or through a *context* that uses the *resource*. "Viewing" the *resource* in the *resource topology* implies a lower level of "commitment" or "permanence" than using the *resource* in a *context*. In particular,

1. Querying the system topology returns a structure of opaque identifiers, the `execution_resource`s, representing a snapshot of the current state of the *system*.
2. The query may require temporarily initializing underlying resources, but those underlying resources need not stay active after the query.
3. Ability to iterate a *resource*'s children in the *resource topology* need not imply ability to create a *context* from that *resource*.
4. Creating a *context* from a *resource* asserts *resource* validity. If the *resource* is invalid, *context* creation must fail. (Compare to how MPI functions report an error if they are called

after `MPI_Finalize` has been called on that process.)
5. Use of a *context* to launch execution asserts *resource* validity, and must thus fail if the *resource* is no longer valid.

Here is a concrete example. Suppose that company "Aleph" makes an accelerator that can be viewed as a *resource*, and that has its own child *resources*. Users must call `Aleph_initialize()` in order to see the accelerator and its children as *resources* in the *resource topology*. Users must call `Aleph_finalize()` when they are done using the accelerator.

Questions:

1. What should happen if users are traversing the *resource topology*, but never use the accelerator's *resource* (other than to iterate past it), and something else concurrently calls `Aleph_finalize()`?
2. What should happen if users are traversing the accelerator's child *resources*, and something else concurrently calls `Aleph_finalize()`?
3. What should happen if users try to create an *execution context* from the accelerator's *resource*, after `Aleph_finalize()` has been called?
4. What should happen to outstanding *execution contexts* that use the accelerator's *resource*, if something calls `Aleph_finalize()` after the *context* was created?

Answers:

1. Nothing bad must happen. Topology queries return a snapshot. Users must be able to iterate past an invalidated *resource*. If users are iterating a *resource* R's children and one child becomes invalid, that must not invalidate R or the iterators to its children.
2. Iterating the children after invalidation of the parent must not be undefined behavior, but the child *resources* remain invalid. Attempts to view and iterate the children of the child *resources* may (but need not) fail.
3. *Context* creation asserts *resource* validity. If the *resource* is invalid, *context* creation must fail. (Compare to how MPI functions report an error if they are called after `MPI_Finalize` has been called on that process.)
4. Use of a *context* in an *executor* to launch execution asserts *resource* validity, and must thus fail if the *resource* is not longer valid.

Querying the relative affinity of partitions

In order to make decisions about where to place execution or allocate memory in a given *system's resource topology*, it is important to understand the concept of affinity between different *execution resources*. This is usually expressed in terms of latency between two resources. Distance does not need to be symmetric in all architectures.

The relative position of two components in the topology does not necessarily indicate their affinity. For example, two cores from two different CPU sockets may have the same latency to access the same NUMA memory node.

This feature could be easily scaled to heterogeneous and distributed systems, as the relative affinity between components can apply to discrete heterogeneous and distributed systems as well.

Proposal

Overview

In this paper we propose an interface for discovering the execution resources within a system, querying the relative affinity metric between those execution resources, and then using those execution resources to allocate memory and execute work with affinity to the underlying hardware those execution resources represent. The interface described in this paper builds on the existing interface for executors and execution contexts defined in the executors proposal [22].

Interface granularity

In this paper is split into two main parts:

- A series of executor properties describe desired behavior when using parallel algorithms or libraries. These properties provide a low granularity and is aimed at users who may have little or no knowledge of the system architecture.
- A series of execution resource topology mechanisms for discovering detailed information about the system's topology and affinity properties which can be used to hand optimize parallel applications and libraries for the best performance. These mechanisms provide a high granularity and is aimed at users who have a high knowledge of the system architecture.

Executor properties

Bulk execution affinity

In this paper we propose an executor property group called `bulk_execution_affinity` which contains the nested properties `none`, `balanced`, `scatter` or `compact`. Each of these properties, if applied to an *executor* enforce a particular guarantee of execution agent binding to the *execution resources* associated with the *executor* in a particular pattern.

Below *(Listing 2)* is an example of executing a parallel task over 8 threads using `bulk_execute`, with the affinity binding `bulk_execution_affinity.scatter`.

```
{
  auto exec = executionContext.executor();

  auto affExec = execution::require(exec, execution::bulk,
    execution::bulk_execution_affinity.scatter);

  affExec.bulk_execute([](std::size_t i, shared s) {
    func(i);
  }, 8, sharedFactory);
}
```

Listing 2: Example of using the bulk_execution_affinity property

Execution resource topology

System topology

The **system topology** is comprised of a directed acyclic graph (DAG) of **execution resources** and **memory resources**, representing unique hardware and software components available within the system capable of executing work, and representing addressable memory regions, respectively. The root node of the DAG is the **system execution resource** and represents the entire system.

Each **execution resource** may have any number of child **execution resources** representing a finer granularity of the parent **execution resource**. Every **execution resource** within the **system topology** (including the **system execution resource**) is exposed via an `execution_resource` object.

Each **execution resource** may point to a number of **memory resources** representing the memory regions on which memory can be allocated from that **execution resource**.

Each **memory resource** may also have any number of child **memory resources** representing a finer granularity of the parent **memory resource**. A **memory resource** can be pointed to by multiple **execution resources**. Every **execution resource** within the **system topology** is exposed via an `memory_resource` object.

The **system topology** can be discovered by calling `this_system::discover_topology`. This will discover all **execution resources** and **memory resources** available within the system and construct the **system topology** DAG, describing a read-only snapshot at the point of the call, and then return an `execution_resource` object exposing the **system execution resource**.

A call to `this_system::discover_topology` may invoke C++ library, system or third party library API calls required to discover certain **execution resources**. However, `this_system::discover_topology` must be thread safe and must initialize and finalize any OS or third-party state before returning.

Below *(Figure 2)* is an example of what a typical **system topology** could look like.

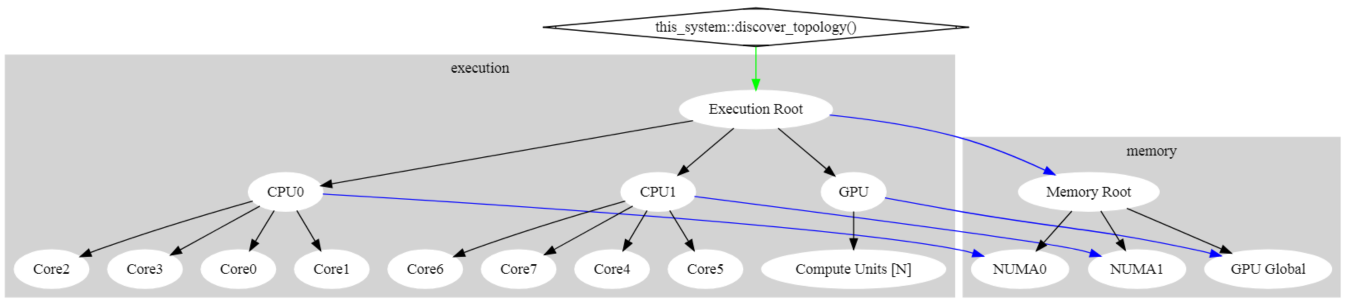

Figure 2: Example system topology DAG

Execution resources

An `execution_resource` is a lightweight structure which identifies a particular **execution resource** within a snapshot of the **system topology**. It can be queried for a name via `name`.

An `execution_resource` object can be queried for a pointer to its parent `execution_resource` via `member_of`, and can also iterate over its children `execution_resource`s via `begin` and `end` or access a particular child via `operator[]`.

An `execution_resource` object can also be queried for the amount concurrency it can provide, the total number of **threads of execution** supported by the associated **execution resource**.

An `execution_resource` object can be queried for a pointer to the root `memory_resource` representing the **memory resource** which the associated **execution resource** can access via `memory_resource`.

> [*Note:* There may be a number of **memory resources** which an **execution resource** can access, but the `memory_resource` pointer returned from `memory_resource` should represent the most coarse grained of these. We may want to expand on this interface in the future. -- *end note*]

Below *(Listing 3)* is an example of iterating over every **execution resource** within the **system topology**.

```
void print_topology(const execution::execution_resource &resource, int indent = 0)
{
  for (int i = 0; i < indent; i++) { std::cout << "  "; }
  std::cout << resource.name() << ": " << resource.concurrency() << "\n";
  for (const execution::execution_resource child : resource) {
    print_topology(child, indent + 1);
  }
}

int main(int argc, char * argv[]) {
  auto systemResource = this_system::discover_topology();
  print_topology(systemResource);
}
```

Listing 3: Example of printing out all execution resources

Memory resources

An `memory_resource` is a lightweight structure which inherits from `pmr::memory_resource` and identifies a particular **memory resource** within a snapshot of the **system topology**. It can be queried for a name via `name`.

A `memory_resource` object can be queried for a pointer to its parent `memory_resource` via `member_of`, and can also iterate over its children `memory_resource`s via `begin` and `end` or access a particular child via `operator[]`.

An allocator capable of allocating memory in the memory region of the **memory resource** represented by a `memory_resource` object by constructing a `pmr::polymorphic_allocator` from the `memory_resource` object.

Querying relative affinity

The `affinity_query` class template provides an abstraction of the relative affinity between an `execution_resource` and a `memory_resource` for a particular `affinity_operation` and `affinity_metric`. The `affinity_query` takes the `affinity_operation` and `affinity_metric` as template parameters, and is constructed from an `execution_resource` and a `memory_resource`.

An `affinity_query` is not generally meaningful on its own. Instead, users are meant to compare two `affinity_query`s via comparison operators, in order to get a relative magnitude of affinity. If necessary, the underlying value of an `affinity_query` can be queried through `native_affinity`, though the return value of this is implementation defined.

Below *(listing 4)* is an example of how to query the relative affinity between an `execution_resource` and a `memory_resource`.

```
auto systemResource = this_system::discover_topology();

auto relativeLatency0 =
execution::affinity_query<execution::affinity_operation::read,
  execution::affinity_metric::latency>(systemResource[0],
systemResource.memory_resource());

auto relativeLatency1 =
execution::affinity_query<execution::affinity_operation::read,
  execution::affinity_metric::latency>(systemResource[1],
systemResource.memory_resource());

auto relativeLatency = relativeLatency1 > relativeLatency0;
```

Listing 4: Example of querying affinity between an `execution_resource` and a `memory_resource`.

> [*Note:* This interface for querying relative affinity is a very low-level interface designed to be abstracted by libraries and later affinity policies. *--end note*]

Execution context

The `execution_context` class provides an abstraction for managing a number of lightweight execution agents executing work on an `execution_resource` and any `execution_resource`s encapsulated by it. An `execution_context` can then provide an executor for submitting work to the `execution_context`. The `execution_context` is constructed with an `execution_resource`.

Below *(Listing 5)* is an example of how this extended interface could be used to construct an `execution_context` from an `execution_resource` which is retrieved from `this_system::discover_topology`.

```
auto systemResource = std::this_system::discover_topology();

execution::execution_context execContext(systemResource[0]);

auto &execResource = execContext.resource();
```

```
// systemResource[0] should be equal to execResource
```

Listing 5: Example of constructing an `execution_context` from an `execution_resource`

When creating an `execution_context` from a given `execution_resource`, the executors associated with it are bound to that `execution_resource` and any `execution_resource`s encapsulated by it. For example, when creating an `execution_resource` from a CPU socket resource, all executors associated with the given socket will spawn execution agents with affinity to the socket partition of the system *(Listing 6)*.

```
auto systemResource = std::this_system::discover_topology();

// find_socket_resource is a user-defined function that finds a resource that is
// a CPU socket in the given resource list
auto socket = find_socket_resource(systemResource);

// Create an execution_context and executor associated with the CPU socket
execution_context context{socket};
auto executor = context.executor();

// Create an allocator from the memory resource associated with the GPU socket
pmr::polymorphic_allocator<int> alloc{socket.memory_resource()};

pmr::vector<int> vec(100, alloc);
std::generate(par.on(executor), std::begin(vec), std::end(vec), genFunc);
```

Listing 6: Example of executing and allocating with affinity

The construction of an `execution_context` on an `execution_resource` implies affinity (where possible) to the given resource. This guarantees that all executors created from that `execution_context` can access the resources and the internal data structures required to guarantee the binding of execution agents.

Only developers that care about resource placement need to care about obtaining executors from the correct `execution_context` object. Existing code for vectors and STL (including the Parallel STL interface) remains unaffected.

If a particular policy or algorithm requires to access placement information, the resources associated with the passed executor can be retrieved via the link to the `execution_context`.

Current resource

The `execution_resource` which underlies the current thread of execution can be queried through `this_thread::get_resource`.

> [*Note:* Binding *threads of execution* can provide performance benefits when used in a way which compliments the application, however incorrect usage can lead to denial of service and therefore can cause loss of performance. *--end note*]

Header `<execution>` synopsis

```
namespace std {
namespace experimental {
namespace execution {

/* Bulk execution affinity properties */

struct bulk_execution_affinity_t;

constexpr bulk_execution_affinity_t bulk_execution_affinity;

/* Execution resource */

class execution_resource {
 public:

  using value_type = execution_resource;
  using pointer = execution_resource *;
  using const_pointer = const execution_resource *;
  using iterator = see-below;
  using const_iterator = see-below;
  using reference = execution_resource &;
  using const_reference = const execution_resource &;
  using size_type = std::size_t;

  execution_resource() = delete;
  execution_resource(const execution_resource &);
  execution_resource(execution_resource &&);
  execution_resource &operator=(const execution_resource &);
  execution_resource &operator=(execution_resource &&);
  ~execution_resource();

  size_type size() const noexcept;

  const_iterator begin() const noexcept;
  const_iterator end() const noexcept;

  const_reference operator[](std::size_t child) const noexcept;

  const_pointer member_of() const noexcept;

  size_t concurrency() const noexcept;

  std::string name() const noexcept;

  memory_resource::pointer memory_resource() const noexcept;

};
```

```
/* Memory resource */

class memory_resource : public pmr::memory_resource {
 public:

  using value_type = memory_resource;
  using pointer = memory_resource *;
  using const_pointer = const memory_resource *;
  using iterator = see-below;
  using const_iterator = see-below;
  using reference = memory_resource &;
  using const_reference = const memory_resource &;
  using size_type = std::size_t;

  memory_resource() = delete;
  memory_resource(const memory_resource &);
  memory_resource(memory_resource &&);
  memory_resource &operator=(const memory_resource &);
  memory_resource &operator=(memory_resource &&);
  ~memory_resource();

  size_type size() const noexcept;

  const_iterator begin() const noexcept;
  const_iterator end() const noexcept;

  const_reference operator[](std::size_t child) const noexcept;

  const_pointer member_of() const noexcept;

  std::string name() const noexcept;

};

/* Execution context */

class execution_context {
 public:

  using executor_type = see-below;

  execution_context(const execution_resource &) noexcept;

  ~execution_context();

  execution_resource &resource() const noexcept;

  executor_type executor() const;

};
```

```
/* Affinity query */

enum class affinity_operation { read, write, copy, move, map };
enum class affinity_metric { latency, bandwidth, capacity, power_consumption };

template <affinity_operation Operation, affinity_metric Metric>
class affinity_query {
 public:

  using native_affinity_type = see-below;
  using error_type = see-below

  affinity_query(const execution_resource &, const memory_resource &) noexcept;

  ~affinity_query();

  native_affinity_type native_affinity() const noexcept;

  friend expected<size_t, error_type> operator==(const affinity_query&, const
affinity_query&);
  friend expected<size_t, error_type> operator!=const affinity_query&, const
affinity_query&);
  friend expected<size_t, error_type> operator<(const affinity_query&, const
affinity_query&);
  friend expected<size_t, error_type> operator>(const affinity_query&, const
affinity_query&);
  friend expected<size_t, error_type> operator<=(const affinity_query&, const
affinity_query&);
  friend expected<size_t, error_type> operator>=(const affinity_query&, const
affinity_query&);

};

}  // execution

/* This system */

namespace this_system {
  execution_resource discover_topology();
}

/* This thread */

namespace this_thread {
  std::experimental::execution::execution_resource get_resource() noexcept;
}

}  // experimental
}  // std
```

Listing 7: Header synopsis

Bulk execution affinity properties

The `bulk_execution_affinity_t` property describes what guarantees executors provide about the binding of *execution agent*s to the underlying *execution resource*s.

bulk_execution_affinity_t provides nested property types and objects as described below. These properties are behavioral properties as described in [22] so must adhere to the requirements of behavioral properties and the requirements described below.

Nested Property Type	Nested Property Name	Requirements
bulk_execution_affinity_t::none_t	bulk_execution_affinity_t::none	A call to an executor's bulk execution function may or may not bind the *execution agent*s to the underlying *execution resource*s. The affinity binding pattern may or may not be consistent across invocations of the executor's bulk execution function.

Nested Property Type	Nested Property Name	Requirements
bulk_execution_affinity_t::scatter_t	bulk_execution_scatter_t::scatter	A call to an executor's bulk execution function must bind the *execution agent*s to the underlying *execution resource*s such that they are distributed across the *execution resource*s where each *execution agent* far from it's preceding and following *execution agent*s. The affinity binding pattern must be consistent across invocations of the executor's bulk execution function.

Nested Property Type	Nested Property Name	Requirements
bulk_execution_affinity_t::compact_t	bulk_execution_compact_t::compact	A call to an executor's bulk execution function must bind the *execution agent*s to the underlying *execution resource*s such that they are in sequence across the *execution resource*s where each *execution agent* close to it's preceding and following *execution agent*s. The affinity binding pattern must be consistent across invocations of the executor's bulk execution function.

Nested Property Type	Nested Property Name	Requirements
bulk_execution_affinity_t::balanced_t	bulk_execution_balanced_t::balanced	A call to an executor's bulk execution function must bind the *execution agent*s to the underlying *execution resource*s such that they are in sequence and evenly spread across the *execution resource*s where each *execution agent* is close to it's preceding and following *execution agent*s and all *execution resource*s are utilized. The affinity binding pattern must be consistent across invocations of the executor's bulk execution function.

> [*Note:* The requirements of the `bulk_execution_affinity_t` nested properties do not enforce a specific binding, simply that the binding follows the requirements set out above and that the pattern is consistent across invocations of the bulk execution functions. *--end note*]

> [*Note:* If two *executor*s `e1` and `e2` invoke a bulk execution function in order, where `execution::query(e1, execution::context) == query(e2, execution::context)` is `true` and `execution::query(e1, execution::bulk_execution_affinity) == query(e2, execution::bulk_execution_affinity)` is `false`, this will likely result in `e1` binding *execution*

> *agents* if necessary to achieve the requested affinity pattern and then `e2` rebinding to achieve the new affinity pattern. *--end note*]

> [*Note:* The terms used for the `bulk_execution_affinity_t` nested properties are derived from the OpenMP properties [33] including the Intel specific balanced affinity binding [[34] --*end note*]

Class `execution_resource`

The `execution_resource` class provides an abstraction over an **execution resource**, that can execute work on lightweight execution agents. An `execution_resource` can represent further `execution_resource`s. We say that these `execution_resource`s are *members of* this `execution_resource`.

> [*Note:* Creating an `execution_resource` may require initializing the underlying software abstraction when the `execution_resource` is constructed, in order to discover other `execution_resource`s accessible through it. However, an `execution_resource` is nonowning. *--end note*]

`execution_resource` member types

```
iterator
```

Requires: `iterator` satisfies the `Cpp17RandomAccessIterator` requirements and `is_same_v<iterator_traits<iterator>::value_type, execution_resource::value_type>` is well-formed and resolves to `true`.

```
const_iterator
```

Requires: `const_iterator` satisfies the `Cpp17RandomAccessIterator` requirements and `is_same_v<iterator_traits<const_iterator>::value_type, execution_resource::value_type>` is well-formed and resolves to `true`.

`execution_resource` constructors

```
execution_resource() = delete;
```

> [*Note:* An implementation of `execution_resource` is permitted to provide non-public constructors to allow other objects to construct them. *--end note*]

`execution_resource` assignment

```
execution_resource(const execution_resource &);
execution_resource(execution_resource &&);
execution_resource &operator=(const execution_resource &);
execution_resource &operator=(execution_resource &&);
```

`execution_resource` destructor

```
~execution_resource();
```

`execution_resource` operations

```
size_t concurrency() const noexcept;
```

Returns: The total concurrency available to this resource. More specifically, the number of *threads of execution* collectively available to this `execution_resource` and any resources which are *members of*, recursively.

```
size_type size() const noexcept;
```

Returns: The number of child `execution_resource`s.

```
const_iterator begin() const noexcept;
```

Returns: A const iterator to the beginning of the child `execution_resource`s.

```
const_iterator end() const noexcept;
```

Returns: A const iterator to the end of the child `execution_resource`s.

```
const_reference operator[](std::size_t child) const noexcept;
```

Returns: A const reference to the specified child `execution_resource`s.

```
const_pointer member_of() const noexcept;
```

Returns: The parent `execution_resource`.

```
std::string name() const noexcept;
```

Returns: An implementation defined string.

Class `memory_resource`

The `memory_resource` class provides an abstraction which represents a **memory resource**, that can allocate memory. A `memory_resource` can represent further `memory_resource`s. We say that these `memory_resource`s are *members of* this `memory_resource`.

The `memory_resource` class must inherit from the `pmr::memory_resource` class.

> [*Note:* Creating an `memory_resource` may require initializing the underlying software abstraction when the `memory_resource` is constructed, in order to discover other `memory_resource`s accessible through it. However, an `memory_resource` is nonowning. *--end note*]

`memory_resource` member types

```
iterator
```

Requires: `iterator` satisfies the `Cpp17RandomAccessIterator` requirements and `is_same_v<iterator_traits<iterator>::value_type, execution_resource::value_type>` is well-formed and resolves to `true`.

```
const_iterator
```

Requires: `const_iterator` satisfies the `Cpp17RandomAccessIterator` requirements and `is_same_v<iterator_traits<const_iterator>::value_type, execution_resource::value_type>` is well-formed and resolves to `true`.

`memory_resource` constructors

```
memory_resource() = delete;
```

> [*Note:* An implementation of `memory_resource` is permitted to provide non-public constructors to allow other objects to construct them. *--end note*]

`memory_resource` assignment

```
memory_resource(const memory_resource &);
memory_resource(memory_resource &&);
memory_resource &operator=(const memory_resource &);
memory_resource &operator=(memory_resource &&);
```

`memory_resource` destructor

```
~memory_resource();
```

`memory_resource` operations

```
size_type size() const noexcept;
```

Returns: The number of child `memory_resource`s.

```
const_iterator begin() const noexcept;
```

Returns: A const iterator to the beginning of the child `memory_resource`s.

```
const_iterator end() const noexcept;
```

Returns: A const iterator to the end of the child `memory_resource`s.

```
const_reference operator[](std::size_t child) const noexcept;
```

Returns: A const reference to the specified child `memory_resource`s.

```
const_pointer member_of() const noexcept;
```

Returns: The parent `memory_resource`.

```
std::string name() const noexcept;
```

Returns: An implementation defined string.

Class `execution_context`

The `execution_context` class provides an abstraction for managing a number of lightweight execution agents executing work on an `execution_resource` and any `execution_resource`s encapsulated by it. The `execution_resource` which an `execution_context` encapsulates is referred to as the *contained resource*.

`execution_context` member types

```
using executor_type = see-below;
```

Requires: `executor_type` is an implementation defined class which satisfies the general executor requires, as specified by [22].

`execution_context` constructors

```
execution_context(const execution_resource &) noexcept;
```

Effects: Constructs an `execution_context` with the provided resource as the *contained resource*.

`execution_context` destructor

```
~execution_context();
```

Effects: May or may not block to wait any work being executed on the *contained resource*.

`execution_context` operators

```
execution_resource &resource() const noexcept;
```

Returns: A const-reference to the *contained resource*.

```
executor_type executor() noexcept;
```

Returns: An executor of type `executor_type` capable of executing work with affinity to the *contained resource*.

Class template `affinity_query`

The `affinity_query` class template provides an abstraction for a relative affinity value between two `execution_resource`s, derived from a particular `affinity_operation` and `affinity_metric`.

`affinity_query` types

```
using native_affinity_type = see-below;
```

Requires: `native_affinity_type` is an implementation defined integral type capable of storing a native affinity value.

```
using error_type = see-below;
```

Requires: `error_type` is an implementation defined integral type capable of storing the an error code value.

`affinity_query` constructors

```
affinity_query(const execution_resource &, const memory_resource &) noexcept;
```

`affinity_query` destructor

```
~affinity_query();
```

`affinity_query` operators

```
native_affinity_type native_affinity() const noexcept;
```

Returns: Unspecified native affinity value.

`affinity_query` comparisons

```
friend expected<size_t, error_type> operator==(const affinity_query&, const
affinity_query&);
friend expected<size_t, error_type> operator!=const affinity_query&, const
affinity_query&);
friend expected<size_t, error_type> operator<(const affinity_query&, const
affinity_query&);
friend expected<size_t, error_type> operator>(const affinity_query&, const
affinity_query&);
friend expected<size_t, error_type> operator<=(const affinity_query&, const
affinity_query&);
friend expected<size_t, error_type> operator>=(const affinity_query&, const
affinity_query&);
```

Returns: An `expected<size_t, error_type>` where,

- if the affinity query was successful, the value of type `size_t` represents the magnitude of the relative affinity;
- if the affinity query was not successful, the error is an error of type `error_type` which represents the reason for affinity query failed.

> [*Note:* An affinity query is permitted to fail if affinity between the two execution resources cannot be calculated for any reason, such as the resources are of different vendors or communication between the resources is not possible. *--end note*]

> [*Note:* The comparison operators rely on the availability of the `expected` class template (see P0323r4: std::expected [30]), if this does not become available then an alternative error/value construct will be adopted instead. *--end note*]

Free functions

`this_system::discover_topology`

The free function `this_system::discover_topology` is provided for discovering the **system topology**.

```
execution_resource discover_topology();
```

Returns: An `execution_resource` object exposing the **system execution resource**.

Requires: If `this_system::discover_topology().size() > 0`, `this_system::discover_topology()[0]` be the `execution_resource` use by `std::thread`. Calls to `this_system::discover_topology()` may not introduce a data race with any other call to `this_system::discover_topology()`.

Effects: Discovers all **execution resources** available within the system and constructs the **system topology** DAG, describing a read-only snapshot at the point of the call.

Throws: Any exception thrown as a result of **system topology** discovery.

`this_thread::get_resource`

The free function `this_thread::get_resource` is provided for retrieving the `execution_resource` underlying the current thread of execution.

```
std::experimental::execution::execution_resource get_resource() noexcept;
```

Returns: The `execution_resource` underlying the current thread of execution.

> [*Note:* The `execution_resource` underlying the current thread of execution may not necessarily be reachable from "top-level" resources visible through `this_system`. *--end note*]

Future Work

How should we define the execution context?

This paper currently defines the execution context as a concrete type which provides the essential interface requires to be constructed from an `execution_resource` and to provide an `executor`.

However going forward there are a few different directions the execution context could take:

- A) The execution context could be **the** standard execution context type, which can be used polymorphically in place of any concrete execution context type in a similar way to the polymorphic executor [22]. This approach allows it to interoperate well with any concrete execution context type, however it may be very difficult to define exactly what this type should look like as the different kinds of execution contexts are still being developed and all the different requirements are still to be fully understood.
- B) The execution context could be a concrete executor type itself, used solely for the purpose of being constructed from and managing a set of `execution_resource`s. This approach would allow the execution context to be tailored specifically for its intended purpose, but would hinder interoperability with other concrete execution context types.
- C) The execution context could be simply a concept, similar to `OnewayExecutor` or `BulkExecutor` for executors, that requires the execution context type to provide the required interface for managing *execution_resource*s. This approach would allow for any concrete execution context type to support the necessary interface for managing execution resources by simply implementing the requirements of the concept. It would also avoid defining any concrete or generic execution context type.

Straw Poll
Should the execution context be a generic polymorphic execution context, as described above in option A?
Should the execution context be a concrete type specifically for the purpose of managing execution resources, as described above in option B?
Should the execution context be a concept, as described above in option C?

Who should have control over bulk execution affinity?

This paper currently proposes the `bulk_execution_affinity_t` properties and it's nested properties for allowing an *executor* to make guarantees as to how *execution agent*s are bound to the underlying *execution resource*s. However providing control at this level may lead to *execution agent*s being bound to *execution resource*s within a critical path. A possible solution to this is to allow the *execution context* to be configured with `bulk_execution_affinity_t` nested properties, either instead of the *executor* property or in addition. This would allow the binding of *threads of execution* to be performed at the time of the *execution context* creation.

Straw Poll

Straw Poll
Should the *execution context* be able to manage the binding of all *threads of execution* which it manages using the `bulk_execution_affinity_t` nested properties?
Should the *executor* be able to manage the binding of all *execution agent*s which it manages using the `bulk_execution_affinity_t` nested properties?
Should both the *execution context* and the *executor* be able to manage the binding of *threads of execution* and subsequently *execution agent*s using the `bulk_execution_affinity_t` nested properties?

Migrating data from memory allocated in one partition to another

With the ability to place memory with affinity comes the ability to define algorithms or memory policies which describe at a higher level how memory is distributed across large systems. Some examples of these are pinned, first touch, and scatter. This is outside the scope of this paper, though we would like to investigate this in a future paper.

Straw Poll
Should the interface provide a way of migrating data between partitions?

Level of abstraction

The current proposal provides an interface for querying whether an `execution_resource` can allocate and/or execute work, it can provide the concurrency it supports and it can provide a name. We also provide the `affinity_query` structure for querying the relative affinity metrics between two `execution_resource`s. However, this may not be enough information for users to take full advantage of the system. For example, they may also want to know what kind of memory is available or the properties by which work is executed. We decided that attempting to enumerate the various hardware components would not be ideal, as that would make it harder for implementers to support new hardware. We think a better approach would be to parameterize the additional properties of hardware such that hardware queries could be much more generic.

We may wish to mirror the design of the executors proposal [22] and have a generic query interface using properties for querying information about an `execution_resource`. We expect that an implementation may provide additional nonstandard, implementation-specific queries.

Straw Poll
Is this the correct approach to take? If so, what should such an interface look like and what kind of hardware properties should we expose?

Acknowledgments

Thanks to Christopher Di Bella, Toomas Remmelg, and Morris Hafner for their reviews and suggestions.

References

[1] P0687r0: Data Movement in C++

[2] The Design of OpenMP Thread Affinity

[3] Euro-Par 2011 Parallel Processing: 17th International, Affinity Matters

[4] Portable Hardware Locality

[5] SYCL 1.2.1

[6] OpenCL 2.2

[7] HSA

[8] OpenMP 5.0

[9] cpuaff

[10] Persistent Memory Programming

[11] MEMKIND

[12] Solaris pbind()

[13] Linux sched_setaffinity()

[14] Windows SetThreadAffinityMask()

[15] Chapel

[16] X10

[17] UPC++

[18] TBB

[19] HPX

[20] MADNESS

[21] Portable Hardware Locality Istopo

[22] A Unified Executors Proposal for C++

[23] P0737r0 : Execution Context of Execution Agents

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

[25] MPI

[26] Parallel Virtual Machine

[27] Building Fault-Tolerant Parallel Applications

[28] Post-failure recovery of MPI communication capability

[29] Fault Tolerance in MPI Programs

[30] p0323r4 std::expected

[31]: Intel® Movidius™ Neural Compute Stick

[32] MADNESS: A Multiresolution, Adaptive Numerical Environment for Scientific Simulation

[33] OpenMP topic: Affinity

[34] Balanced Affinity Type