P0796r1: Supporting Heterogeneous & Distributed Computing Through Affinity

Date: 2018-02-12

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

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

Reply to: gordon@codeplay.com

Changelog

P0796r1 (JAX 2017)

- Introduce proposed wording.
- Based on feedback from SG1, introduced 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 resouce.

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 memory affinity 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.

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 architectures exist 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 Non-Uniform Memory Architectures (NUMA).

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 is initialized at the beginning of the program by the initial thread and is used by multiple threads. While automatic thread migration has been implemented in some OSes, 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 is read by multiple threads, or migrate data which is 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 used 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 a parallel 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(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(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 affininty 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.

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.
- How to support dynamic topology discovery and fault tolerance.

Querying and representing the system topology

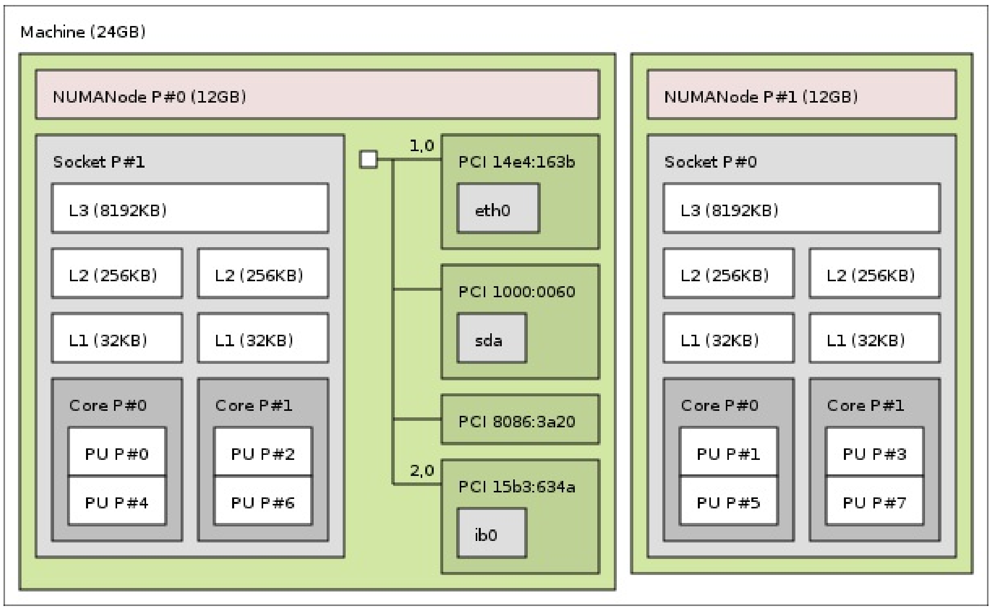
The first task in allowing C++ applications to leverage memory locality is to provide the ability to query a *system* for its *resource topology* (commonly represented as a tree or graph) and traverse its *execution resources*.

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] leaves the *execution resource* 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 topology'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 CPU + GPU system 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 the 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 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.



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 queryable 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, and is particularly false 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) *online 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 *online 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.

Lifetime considerations

As the execution context would provide a partitioning interface which returns objects describing the components of the system topology of an execution resource, it is important to consider the lifetime of these objects.

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

For these reasons *resources* must always outlive any *execution context* which is constructed from them, and any *resource* retrieved from an *execution context* must not be tied to the lifetime of that *execution context*.

The initial solution should target systems with a single addressable memory region. It should thus exclude devices like discrete GPUs. 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.

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 querying and representing the execution resources within a system, queurying 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. The interface described in this paper builds on the existing initerface for executors and execution contexts defined in the executors proposal [22].

Execution resources

An execution_resource is a light weight structure which acts as an identifier to particular piece of hardware within a system. It can be queried for whether it can allocate memory via can_place_memory and whether it can execute work via can_place_agents, and for it's name via name. An execution_resource can also represent other execution_resources, these are refered to as being *members of* that execution_resource and can be queried via resources. Additionally the execution_resource which another is a *member of* can be queried vis member_of. An execution_resource can also be queried for the concurrency it can provide; the total number of *threads of execution* supported by that *execution_resource* and all resources it represents.

> [*Note:* Note that an execution resource is not limited to resources which execute work, but also a general resource where no execution can take place but memory can be allocated such as off-chip memory. *--end note*]

> [*Note:* The intention is that the actual implementation details of a resource topology are described in an execution context when required. This allows the execution resource objects to be lightweight objects that serve as identifiers that are only referenced. *--end note*]

System topology

The system topology is made up of a number of system level execution_resources, which can be queried through this_system::resource which returns a std::vector. The execution_resources available within the system can be initialised dynamically by a runtime library, however must be done so before main is called, given that after that point the system topology cannot change.

Below *(Listing 2)* is an example of iterating over the system level resources and priniting out it's capabilities.

```
for (auto res : execution::this_system::resources()) {
    std::cout << res.name()  `\n`;
    std::cout << res.can_place_memory() << `\n`;
    std::cout << res.can_place_agents() << `\n`;
    std::cout << res.concurrency() << `\n`;
}
```

Listing 2: Example of querying all the system level execution resources

Querying relative affinity

The affinity_query class template provides an abstraction for a relative affinity value between two execution_resources, derived from a particular affinity_operation and affinity_metric. The affinity_query is templated by affinity_operation and affinity_metric and is constructed from two execution_resources. An affinity_query does not mean much on it's own, instead a relative magnitude of affinity can be queried by using comparison operators. If nessesary the value of an affinity_query can also be queried through native_affinity, though the return value of this is implementation defined.

Below *(listing 3)* is an example of how you can query the relative affinity between two execution_resources.

```
auto systemLevelResources = execution::this_system::resources();
auto memberResources = systemLevelResources.resources();

auto relativeLatency01 =
execution::affinity_query<execution::affinity_operation::read,
  execution::affinity_metric::latency>(memberResources[0],
memberResources[1]);

auto relativeLatency02 =
execution::affinity_query<execution::affinity_operation::read,
  execution::affinity_metric::latency>(memberResources[0],
memberResources[2]);

auto relativeLatency = relativeLatency01 > relativeLatency02;
```

Listing 3: Example of querying affinity between two execution_resources.

> [*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 light weight execution agents executing work on an execution_resource and any execution_resources encapsulated by it. An execution_context can then provide an executor for executing work and an allocator or polymorphic memory resource for allocating memory. The execution_context is constructed with an execution_resource, the execution_context then executes work or allocates memory for that execution_resource and an execution_resource that it represents.

Below *(Listing 4)* is an example of how this extended interface could be used to construct an *execution context* from an *execution resource* which is retrieved from the *system's resource topology*. Once an *execution context* is constructed it can then still be queried for its *execution resource* and then that *execution resource* can be further partitioned.

```
auto &resources = execution::this_system::resources();

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

auto &systelLevelResource = execContext.resource();

// resource[0] should be equal to execResource

for (auto res : systelLevelResource.resources()) {
    std::cout << res.name()  << `\n`;
}
```

Listing 4: Example of constructing an execution context from an execution resource

Binding execution and allocation to resources

When creating an execution_context from a given execution_resource, the executors and allocators associated with it are bound to that execution_resource. 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 5)*.

```
auto cList = std::execution::this_system::resources();
// FindASocketResource is a user-defined function that finds a
// resource that is a CPU socket in the given resource list
auto& socket = findASocketResource(cList);
execution_contexteC{socket} // Associated with the socket
auto executor = eC.executor(); // By transitivity, associated with the socket
too
auto socketAllocator = eC.allocator(); // Retrieve an allocator to the closest
memory node
std::vector<int, decltype(socketAllocator)> v1(100, socketAllocator);
std::generate(par.on(executor), std::begin(v1), std::end(v1), std::rand);
```

Listing 5: Example of allocating with affinity to an execution resource

The construction of an execution_context on a component 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 requires to guarantee the placement of the processor.

Only developers that care about resource placement need to care about obtaining executors and allocations 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.

Header <execution> synopsis

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

/* Execution resource */

class execution_resource {
 public:

  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_t concurrency() const noexcept;

  std::vector<resource> resources() const noexcept;

  const execution_resource member_of() const noexcept;

  std::string name() const noexcept;

  bool can_place_memory() const noexcept;
  bool can_place_agent() const noexcept;

};

/* Execution context */

class execution_context {
 public:

  using executor_type = see-below;

  using pmr_memory_resource_type = see-below;

  using allocator_type = see-below;

  execution_context(const execution_resource &) noexcept;

  ~execution_context();

  const execution_resource &resource() const noexcept;
  executor_type executor() const;

  pmr_memory_resource_type &memory_resource() const;

  allocator_type allocator() 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(execution_resource &&, execution_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&);

};

/* This system */

namespace this_system {
  std::vector<execution_resource> resources() noexcept;
}

}  // execution
}  // experimental
}  // std
```

Listing 6: Header synopsis

Class execution_resource

The execution_resource class provides an abstraction over a system's hardware capable to allocate memory, execute light weight execution agents or both. An execution_resource can represent further execution_resources, these execution_resources are said to be *members of* this execution_resource.

> [*Note:* The execution_resource is required to be implemented such that the underlying software abstraction is initialised when the execution_resource is constructed, maintained through reference counting and cleaned up on destruction of the final reference. --*end note*]

execution_resource constructors

```
execution_resource();
```

> [*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.

```
std::vector<resource> resources() const noexcept;
```

Returns: All execution_resources which are *members of* this resource.

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

Returns: The execution_resource which this resource is a *member of*.

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

Returns: An implementation defined string.

```
bool can_place_memory() const noexcept;
```

Returns: If this resource is capable of allocating memory with affinity, 'true'.

```
bool can_place_agent() const noexcept;
```

Returns: If this resource is capable of execute with affinity, 'true'.

Class execution_context

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

execution_context types

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

Requires: executor_type is an implementation defined class which satifies the general executor requires, as specified by P0443r5.

```
using pmr_memory_resource_type = see-below;
```

Requires: pmr_memory_resource_type is an implementation defined class which inherits from std::pmr::memory_resource.

```
using allocator_type = see-below;
```

Requires: allocator_type is an implementation defined allocator class.

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

```
const 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*.

Throws: An exception !this->resource().can_place_agents().

```
pmr::memory_resource &memory_resource() noexcept;
```

Returns: A reference to a polymorphic memory resource of type pmr_memory_resource_type capable of allocating with affinity to the *contained resource*.

Throws: If !this->resource().can_place_memory().

```
allocator_type allocator() const;
```

Returns: An allocator of type allocator_type capable of allocating with affinity to the *contained resource*.

Throws: If !this->resource().can_place_memory().

Class template affinity_query

The affinity_query class template provides an abstraction for a relative affinity value between two execution_resources, 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 execution_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 succesful, 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

The free function this_system::resources is provided for retrieving the execution_resources which encapsulate the hardware platforms available within the system, these are refered to as the *system level resources*.

```
std::vector<execution_resource> resources() noexcept;
```

Returns: A std::vector containing all *system level resources*.

Requires: If this_system::resources().size() > 0, this_system::resources()[0] be the execution_resource use by std::thread. The value returned by this_system::resources() be the same at any point after the invocation of main.

> [*Note:* Returning a std::vector allows users to potentially manipulate the container of execution_resources after it is returned, we may want to replace this with an alternative type which is more restrictive at a later date such as a range. --*end note*]

Future Work

Migrating data from memory allocated in one partition to another

In some cases for performance it is important to bind a memory allocation to a memory region for the duration of an a tasks execution, however in other cases it's important to be able to migrate the data from one memory region to another. This is outside the scope of this paper, however we would like to investigate this in a future paper.

Straw Poll

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

Defining memory placement algorithms or policies

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, however we would like to investigate this in a future paper.

Straw Poll

Should the interface provide standard algorithms or policies for distributing memory?

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_resources. However this may not be enough information for users to take full advance of the system, they may also want to know what kind of memory is available or the properties by which work is executed. It was 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. It has been discussed that a better approach would be to parameterise 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 and have a generic query interface using properties for querying information about an execution_resource. It's expected that an implementation may provide additional nonstandard queries that are specific to that implementation.

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?

Dynamic topology discovery

The current proposal requires that all execution_resources are initialised before main is called, therefore not allowing an execution_resource to become available or go offline at runtime. We may wish to support this in the future, however this is outside of the scope of this paper.

Straw Poll

Should we support dynamically adding and removing execution_resources at runtime?

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