

P0796r2: Supporting Heterogeneous & Distributed Computing Through Affinity

Date: 2018-05-07

Audience: SG1, 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

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

Reply to: gordon@codeplay.com

Changelog

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; firstly a series of executor properties which can be used to apply affinity requirements to bulk execution functions, and secondly 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 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 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.

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*.

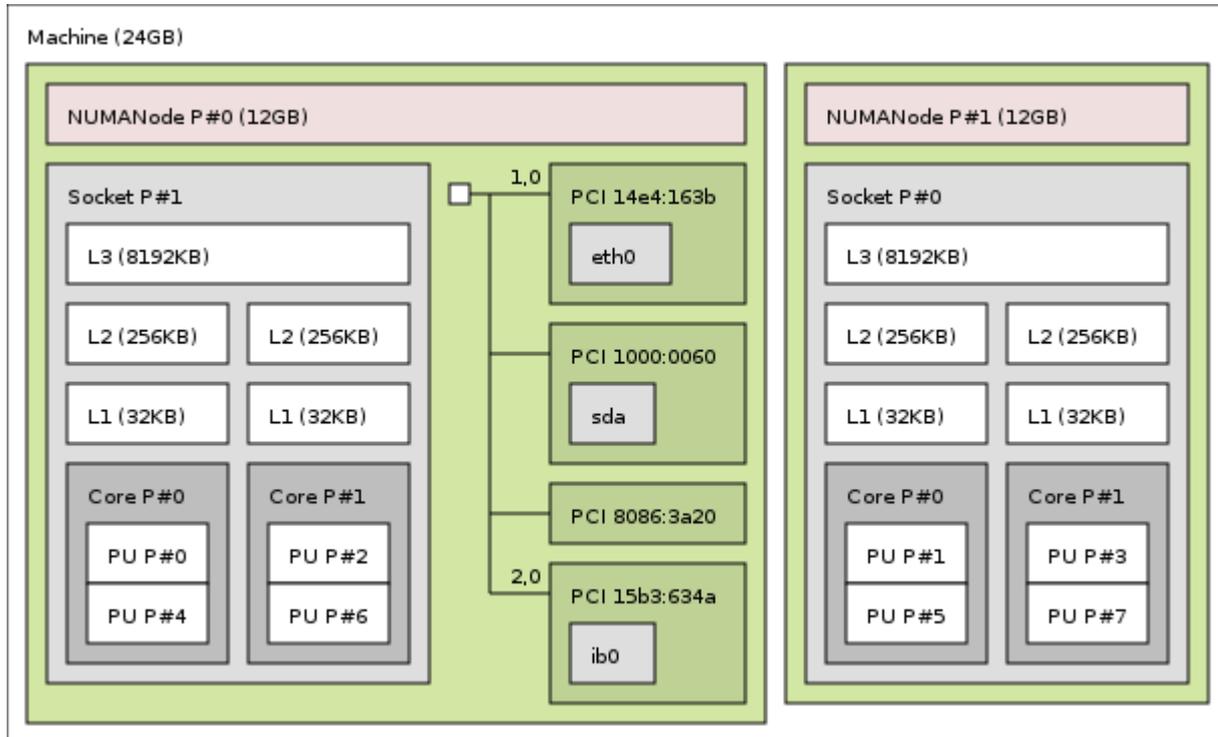
The capability of querying underlying *execution resources* of a given *system* is particularly important towards supporting affinity control in C++. The current proposal for executors [22] leaves the *execution resource* largely unspecified. This is intentional: *execution resources* will vary greatly 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 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 (Istopo) 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 Istopo [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 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) *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, 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. 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 optimise 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

Execution resources

An `execution_resource` is a lightweight 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`, whether it can execute work via `can_place_agents`, and for its name via `name`. An `execution_resource` can also represent other `execution_resources`. We call these *members of* that `execution_resource`, and can be queried via `resources`. Additionally the `execution_resource` which another is a *member of* can be queried via `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::get_resources* which returns a *std::vector*. A run-time library may initialize the *execution_resources* available within the system dynamically. However, this must be done before *main* is called, given that after that point, the system topology may not change.

Below (*Listing 3*) is an example of iterating over the system-level resources and printing out their capabilities.

```
for (auto res : execution::this_system::get_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 3: 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*. This value depends on a particular *affinity_operation* and *affinity_metric*. As a result, the *affinity_query* is templated on *affinity_operation* and *affinity_metric*, and is constructed from two *execution_resources*. An *affinity_query* is not meant to be meaningful on its own. Instead, users are meant to compare two queries with comparison operators, in order to get a relative magnitude of affinity. If necessary, the value of an *affinity_query* can also 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 two *execution_resources*.

```
auto systemLevelResources = execution::this_system::get_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 4: Example of querying affinity between two `execution_resource`s.

[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_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`. Then, the `execution_context` may execute work or allocate memory for that `execution_resource` and an `execution_resource` that it represents.

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 the *system's resource topology*. Once an *execution context* is constructed it can then still be queried for its *execution resource*, and that *execution resource* can be further partitioned.

```

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

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

auto &systemLevelResource = execContext.resource();

// resource[0] should be equal to execResource

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

```

Listing 5: Example of constructing an execution context from an execution resource

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 6*).

```

auto cList = std::execution::this_system::get_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_contextC{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 6: 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`.

Current resource

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

Binding to execution

A *thread of execution* can be requested to bind to a particular `execution_resource` for a particular *execution agent* by calling `this_thread::bind` if that `execution_resource` is able to place agents. If the current *thread of execution* is successfully bound to the specified `execution_resource` it will return `true` otherwise it will return `false`. If the *thread of execution* is successfully bound to the specified `execution_resource` then `execution_resource` returned by `this_thread::get_resource` must be equal to the `execution_resource` provided to `this_thread::bind`. Subsequently a *thread of execution* can be unbound by calling `this_thread::unbind`.

[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:

    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&);

};

} // execution

/* This system */

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

/* This thread */

namespace this_thread {
```

```

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

bool bind(execution_resource eR) noexcept;
bool unbind(execution_resource eR) 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 agents* to the underlying *execution resources*.

`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 |
|------------------------------------------------|----------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>bulk_execution_affinity_t::none_t</code> | <code>bulk_execution_affinity_t::none</code> | A call to an executor's bulk execution function may or may not bind the <i>execution agents</i> to the underlying <i>execution resources</i> . 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 <i>execution agents</i> to the underlying <i>execution resources</i> such that they are distributed across the <i>execution resources</i> where each <i>execution agent</i> far from its preceding and following <i>execution agents</i> . 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 | <p>A call to an executor's bulk execution function must bind the <i>execution agents</i> to the underlying <i>execution resources</i> such that they are in sequence across the <i>execution resources</i> where each <i>execution agent</i> close to its preceding and following <i>execution agents</i>. The affinity binding pattern must be consistent across invocations of the executor's bulk execution function.</p> |

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

[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 *executors* `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 a system's hardware, that can allocate memory and/or execute lightweight execution agents. An `execution_resource` can represent further `execution_resources`. We say that these `execution_resources` are *members of* this `execution_resource`.

[*Note:* The `execution_resource` is required to be implemented such that the underlying software abstraction is initialized 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 lightweight execution agents executing work on an `execution_resource` and any `execution_resources` encapsulated by it. The `execution_resource` which an `execution_context` encapsulates is referred to as the *contained resource*.

`execution_context` types

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

Requires: `executor_type` is an implementation defined class which satisfies 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 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::get_resources`

The free function `this_system::get_resources` is provided for retrieving the `execution_resources` which encapsulate the hardware platforms available within the system. We refer to these resources as the *system level resources*.

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

Returns: An `std::vector` containing all *system level resources*.

Requires: If `this_system::get_resources().size() > 0`, `this_system::get_resources()[0]` be the `execution_resource` use by `std::thread`. The value returned by `this_system::get_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 at a later date with an alternative type which is more restrictive, such as a range or span. --end note]

`this_thread::bind` & `this_thread::unbind`

The free functions `this_thread::bind` and `this_thread::unbind` are provided for binding / unbinding the current *thread of execution* to / from a particular `execution_resource`.

```
bool bind(execution_resource eR) noexcept;
```

Returns: `true` if the requested binding was successful, otherwise `false`.

Requires: `eR.can_place_agents() == true`.

Effects: If successful, binds the current *thread of execution* to the specified `execution_resource`.

```
bool unbind(execution_resource eR) noexcept;
```

Returns: `true` if the requested unbinding was successful, otherwise `false`.

Requires: `eR.can_place_agents() == true`.

Effects: If successful, unbinds the current *thread of execution* from the specified `execution_resource`.

`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.

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 affinity-based

`allocator` or `pmr::memory_resource` and `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_resources`. This approach would allow the execution context to be tailored specific for it's original purpose, however it would be more difficult to support interoperability with other concrete execution context types.
- C) The execution context could be simply a concept, similar to `OnewayExecutor` or `BulkExecutor`, for executors, where it requires the execution context type to provide the required interface for managing `execution_resources`. This approach would allow for any concrete execution context type to support necessary interface for managing execution resources by simply implementing the requirements of the concept, and would 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_agents` are bound to the underlying `execution_resources`. However providing control at this level may lead to `execution_agents` being bound to `execution_resources` 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

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_agents` which it manages using the `bulk_execution_affinity_t` nested properties?

Straw Poll

Should both the *execution context* and the *executor* be able to manage the binding of *threads of execution* and subsequently *execution agents* 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_resources`. 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 implementors 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 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?

Dynamic topology discovery

The current proposal requires that all `execution_resources` are initialized before `main` is called. This therefore does not permit an `execution_resource` to become available or go off-line at run time. We may wish to support this in the future, however this is outside of the scope of this paper at the moment.

Straw Poll

Should we support dynamically adding and removing `execution_resources` at run time?

Acknowledgements

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