P1436r2: Executor properties for affinity-based execution

Date: 2019-11-23

Audience: SG1, SG14

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

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

Reply to: gordon@codeplay.com

Acknowledgements

This paper is the result of discussions from man contributors within SG1, SG14 and the heterogeneous C++ group, including Patrice Roy, Carl Cook, Jeff Hammond, Hartmut Kaiser, Christian Trott, Paul Blinzer, Alex Voicu, Nat Goodspeed, Tony Tye, Chris Kohlhoff and Domagoj Šarić.

Changelog

P1436r3 (PRA 2020)

- Rename `bulk_execution_affinity_t::scatter_t` to `bulk_execution_affinity_t::spread_t`.
- Rename `bulk_execution_affinity_t::compact_t` to `bulk_execution_affinity_t::close_t`.
- Refine the wording of the `bulk_execution_affinity_t` properties to clarify the requirements on binding and chunking, based on feedback from SG1.

P1436r2 (BEL 2019)

- Alter the wording on the `bulk_execution_affinity_t` properties so they are now hints that request the executor provide a particular pattern of binding, rather than a guarantee.

P1436r1 (COL 2019)

- Introduce wording to clarify when two invocations of bulk_execute are expected to have consistent binding.
- Introduce wording to describe how bulk_execute should handle an execution context failing to provide the guaranteed binding.
- Update the wording of bulk_execution_affinity.scatter and bulk_execution_affinity.balance to better describe the expected binding pattern.

P1436r0 (KON 2019)

- Separation of high-level features from P0796r3 [35].
- Update motivational examples.
- Introduce new executor property `concurrency_t`.
- Introduce new executor property `execution_locality_intersection_t`.
- Introduce new executor property `memory_locality_intersection_t`.
- Update direction for future work.

P0796r3 (SAN 2018)

- Remove reference counting requirement from `execution_resource`.
- Change lifetime model of `execution_resource`: it now either consistently identifies some underlying resource, or is invalid; context creation rejects an invalid resource.ster
- Remove `this_thread::bind` & `this_thread::unbind` interfaces.

- Make `execution_resource`s iterable by replacing `execution_resource::resources` with `execution_resource::begin` and `execution_resource::end`.
- Add `size` and `operator[]` for `execution_resource`.
- Rename `this_system::get_resources` to `this_system::discover_topology`.
- Introduce `memory_resource` to represent the memory component of a system topology.
- Remove `can_place_memory` and `can_place_agents` from the `execution_resource` as these are no longer required.
- Remove `memory_resource` and `allocator` from the `execution_context` as these no longer make sense.
- Update the wording to describe how execution resources and memory resources are structured.
- Refactor `affinity_query` to be between an `execution_resource` and a `memory_resource`.

P0796r2 (RAP 2018)

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

P0796r1 (JAX 2018)

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

P0796r0 (ABQ 2017)

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

Abstract

This paper is the result of a request from SG1 at the 2018 San Diego meeting to split P0796: Supporting Heterogeneous & Distributed Computing Through Affinity [35] into two separate papers, one for the high-level interface and one for the low-level interface. This paper focusses on the high-level interface: a series of properties for querying affinity relationships and requesting affinity on work being executed. P0437 will focus on the low-level interface: a mechanism for discovering the topology and affinity properties of a given system, however this paper was not submitted in this mailing.

The aim of this paper is to provide a number of executor properties that if supported allow the user of an executor to query and manipulate the binding of *execution agents* and the underlying *execution resources* of the *threads of execution* they are run on.

Motivation

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

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

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

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

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

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

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 heterogeneity model [3]. (One of the authors of this document participated in the design of OpenMP's affinity model.)

Motivational examples

To identify the requirements for supporting affinity we have looked at a number of use cases where affinity between memory locality and execution can provide better performance.

Consider the following code example *(Listing 1)* where the C++17 parallel STL algorithm `for_each` is used to modify the elements of a `std::vector data` on an *executor* that will execute on a NUMA system with a number of CPU cores. However the memory is allocated by the `std::vector` default allocator immediately during the construction of `data` on memory local to the calling thread of execution. This means that the memory allocated for `data` may have poor locality to all of the NUMA regions on the system, other than the one in which the constructor executed. Therefore, accesses in the parallel `for_each` made by threads in other NUMA regions will incur high latency. In this example, this is avoided by migrating `data` to have better affinity with the NUMA regions on the system using an *executor* with the `bulk_execution_affinity.spread` property applied, before it is accessed by the `for_each`. Note that a mechanism for migration is not yet specified in this paper, so this example currently uses an arbitrary vendor API, `vendor_api::migrate`. Our intention is that a future revision of this paper will specify a standard mechanism for migration

```
// NUMA executor representing N NUMA regions.
numa_executor exec;

// Storage required for vector allocated on construction local to current thread
// of execution, (N == 0).
std::vector<float> data(N * SIZE);

// Require the NUMA executor to bind its migration of memory to the underlying
// memory resources in a spread pattern.
auto affinityExec = std::execution::require(exec,
  bulk_execution_affinity.spread);

// Migrate the memory allocated for the vector across the NUMA regions in a
// spread pattern.
vendor_api::migrate(data, affinityExec);
```

```
// Placement of data is local to NUMA region 0, so data for execution on other
// NUMA nodes must is migrated when accessed.
std::for_each(std::execution::par.on(affinityExec), std::begin(data),
  std::end(data), [=](float &value) { do_something(value): });
```

Listing 1: Migrating previously allocated memory.

Now consider a similar code example *(Listing 2)* where again the C++17 parallel STL algorithm `for_each` is used to modify the elements of a `std::vector data` on an *executor* that will execute on a NUMA system with a number of CPU cores. However, instead of migrating `data` to have affinity with the NUMA regions, `data` is allocated within a bulk execution by an *executor* with the `bulk_execution_affinity.spread` property applied so that `data` is allocated with affinity. Then when the `for_each` is called with the same executor, `data` maintains its affinity with the NUMA regions.

```
// NUMA executor representing N NUMA regions.
numa_executor exec;

// Reserve space in a vector for a unique_ptr for each index in the bulk
// execution.
std::vector<std::unique_ptr<float[SIZE]>> data{};
data.reserve(N);

// Require the NUMA executor to bind its allocation of memory to the underlying
// memory resources in a spread pattern.
auto affinityExec = std::execution::require(exec,
  bulk_execution_affinity.spread);

// Launch a bulk execution that will allocate each unique_ptr in the vector with
// locality to the nearest NUMA region.
affinityExec.bulk_execute([&](size_t id) {
  data[id] = std::make_unique<float>(); }, N, sharedFactory);

// Execute a for_each using the same executor so that each unique_ptr in the
// vector maintains its locality.
std::for_each(std::execution::par.on(affinityExec), std::begin(data),
  std::end(data), [=](float &value) { do_something(value): });
```

Listing 2: Aligning memory and process affinity.

Background Research

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 migrate memory work and memory allocations between execution resources.
- How to query affinity properties between different *executors*.
- How to bind execution and allocation particular *executors*.

Wherever possible, we also evaluate how an affinity-based solution could be scaled to support both distributed and heterogeneous systems.

State of the art

The *affinity problem* existed for some time, and there are a number 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 ideas are suitable for adoption into C++. 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()`.

Relative affinity of execution resources

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 hardware and software 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 a system's 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 can be scaled to heterogeneous and distributed systems, as the relative affinity between components can apply to discrete heterogeneous and distributed systems as well.

Inaccessible memory

The initial solution proposed by this paper may only target systems with a single addressable memory region. It may therefore exclude certain heterogeneous devices such as some discrete GPUs. However, in order to maintain a unified interface going forward, the initial solution should consider these devices and be able to scale to support them in the future.

Proposal

Overview

In this paper we propose executor properties that can be used for querying the affinity between different hardware and software resources within a system available that are available to executors and to require binding of *execution agents* to the underlying hardware or software resources in order to achieve performance through data locality. These properties provide a low granularity and is aimed at users who may have little or no knowledge of the system architecture.

The interface described in this paper builds on the existing interface for executors and execution contexts defined in the executors proposal [22].

Execution resources

An *execution resource* represents an abstraction of a hardware or software layer that guarantees a particular set of affinity properties, where the level of abstraction is implementation-defined. An implementation is permitted to migrate any

underlying resources providing it guarantees the affinity properties remain consistent. This allows freedom for the implementor but also consistency for users.

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

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

In summary, an *execution resource* either identifies a thing uniquely, or harmlessly points to nothing.

Header `<execution>` synopsis

Below *(Listing 3)* is a proposed extension to the `<execution>` header.

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

// Bulk execution affinity properties

struct bulk_execution_affinity_t;

constexpr bulk_execution_affinity_t bulk_execution_affinity;

// Concurrency property

struct concurrency_t;

constexpr concurrency_t concurrency;

// Execution locality intersection property

struct execution_locality_intersection_t;

constexpr execution_locality_intersection_t<DestExecutor>;

// Memory locality intersection property

struct memory_locality_intersection_t;

constexpr memory_locality_intersection_t memory_locality_intersection;

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

Listing 3: Header synopsis

Bulk execution affinity properties

We propose an executor property group called `bulk_execution_affinity` which contains the nested properties `none`, `balanced`, `spread` and `close`. Each of these properties, if applied to an *executor* provides a hint to the `executor` that

requests a particular binding of *execution agents* to the *execution resources* associated with the *executor* in a particular pattern.

Example

Below is an example *(Listing 4)* of executing a parallel task over 8 threads using `bulk_execute`, with the affinity binding `bulk_execution_affinity.spread`. We request affinity binding using `prefer` and then check to see if the executor is able to support it using `query`.

```
{
  bulk_executor exec;

  auto affExec = execution::prefer(exec,
    execution::bulk_execution_affinity.spread);

  if (execution::query(affExec, execution::bulk_execution_affinity.spread)) {
    std::cout << "bulk_execute using bulk_execution_affinity.spread"
      << std::endl;
  }

  execution::bulk_execute(affExec, [](std::size_t i) {
    func(i);
  }, 8);
}
```

Listing 4: Example of using the bulk_execution_affinity property

Proposed Wording

The `bulk_execution_affinity_t` properties are a group of mutually exclusive behavioral properties (as defined in P0443 [22]) which provide a hint to the *executor* to, if possible, bind the *execution agents* created by a bulk invocation from an *executor*, to the underlying *execution resources* in a particular pattern relative to their physical closeness.

The `bulk_execution_affinity_t` nested properties are defined using the following terms of art:

- *Available concurrency*; which is defined the number of *execution resources* available to an *executor* which can be bound to *execution agents* concurrently, assuming no contention.
- *Locality distance*; which is defined an implementation-defined metric for measuring the relative affinity between *execution resources* whereby *execution resources* with a lower *locality distance* are likely to have similar latency in memory access operations, for a given memory location.

The `bulk_execution_affinity_t` nested properties also refer to the subdivision of *execution resources*, which is an implementation-defined method of subdividing the *available concurrency*, generally based on groupings of *execution resources* with the lowest *locality distance* to each other.

> [*Note:* An alternative term of art for *locality distance* could be *locality interference*. --*end note*]

The `bulk_execution_affinity_t` property provides nested property types and objects as described below, where:

- `e` denotes an executor object of type `E`,
- `f` denotes a function object of type `F&&`,
- `s` denotes a shape object of type `execution::executor_shape<E>`,
- `sf` denotes a function object of type `SF`, and
- a call to `execution::bulk_execute(e, f, s)` creates a consecutive sequence of work-items from `0` to `s-1`, mapped to the available concurrency of `e`, that is a number of execution resources, which are subdivided in some implementation-defined way.

Nested Property Type	Nested Property Name	Requirements

Nested Property Type	Nested Property Name	Requirements
`bulk_execution_affinity_t::none_t`	`bulk_execution_affinity_t::none`	A call to `execution::bulk_execute(e, f, s)` is not required to bind the created *execution agents* for the work-items of the iteration space specified by `s` to *execution resources*.
`bulk_execution_affinity_t::spread_t`	`bulk_execution_spread_t::spread`	A call to `e.bulk_execute(f, s, sf)` should aim to bind the created *execution agents* for the work-items of the iteration space specified by `s` to *execution resources* such that the average locality distance of adjacent work-items in the same subdivision of the available concurrency is maximized and the average locality distance of adjacent work-items in different subdivisions of the available concurrency is maximized. The binding of all *execution agents* to all *execution resources* must not result in the difference between the number of *execution agents* assigned to any *execution resources* being greater than `1`. If `e` is not able to fulfil this aim the it should fall back to `bulk_execution_affinity_t::none_t`.
`bulk_execution_affinity_t::close_t`	`bulk_execution_close_t::close`	A call to `e.bulk_execute(f, s, sf)` should aim to bind the created *execution agents* for the work-items of the iteration space specified by `s` to *execution resources* such that the average locality distance between adjacent work-items is minimized. The binding of all *execution agents* to all *execution resources* must not result in the difference between the number of *execution agents* assigned to any *execution resources* being greater than `1`. If `e` is not able to fulfil this aim the it should fall back to `bulk_execution_affinity_t::none_t`.

Nested Property Type	Nested Property Name	Requirements
`bulk_execution_affinity_t::balanced_t`	`bulk_execution_balanced_t::balanced`	A call to `e.bulk_execute(f, s, sf)` should aim to bind the created *execution agents* for the work-items of the iteration space specified by `s` to *execution resources* such that the average locality distance of adjacent work-items in the same subdivision of the available concurrency is minimized and the average locality distance of adjacent work-items in different subdivisions of the available concurrency is maximized. The binding of all *execution agents* to all *execution resources* must not result in the difference between the number of *execution agents* assigned to any *execution resources* being greater than `1`. If `e` is not able to fulfil this aim the it should fall back to `bulk_execution_affinity_t::none_t`.

> [*Note:* Note: The subdivision of the available concurrency is implementation-defined. *--end note*]

> [*Note:* Note: If the number of work-items specified by `s` is larger than the available concurrency, the manner in which that iteration space is subdivided into a consecutive sequence of work-items is implementation-defined. *--end note*]

> [*Note:* It's expected that the default value of `bulk_execution_affinity_t` for most executors be `bulk_execution_affinity_t::none_t`. *--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. Rebinding *execution agents* to *execution resources* may take substantial time and may affect performance of subsequent code. *--end note*]

Concurrency property

We propose a query-only executor property called `concurrency_t` which returns the maximum potential concurrency available to *executor*.

Example

Below is an example *(Listing 5)* of querying an executor for the maximum concurrency it can provide via `concurrency`.

```
{
  executor exec;

  auto maxConcurrency = execution::query(exec, execution::concurrency);
}
```

Listing 5: Example of using the concurrency property

Proposed Wording

The `concurrency_t` property *(Listing 6)* is a query-only property as defined in P0443 [22].

```
struct concurrency_t
{
  static constexpr bool is_requirable = false;
  static constexpr bool is_preferable = false;

  using polymorphic_query_result_type = size_t;

  template<class Executor>
    static constexpr decltype(auto) static_query_v
      = Executor::query(concurrency_t());
};
```

Listing 6: Proposed specification for concurrency_t

The `concurrency_t` property can be used only with `query`, which returns the maximum potential concurrency available to the executor. If the value is not well defined or not computable, `0` is returned.

The value returned from `execution::query(e, concurrency_t)`, where `e` is an executor, shall not change between invocations.

> [*Note:* The expectation here is that the maximum available concurrency for an *executor* as described here is equivalent to calling `this_thread::hardware_concurrency()` *--end note*]

Execution locality intersection property

We propose a query-only executor property called `execution_locality_intersection_t` which returns the maximum potential concurrency that ia available to both of two *executors*.

Example

Below is an example *(Listing 7)* of querying whether two *executors* have overlapping maximum concurrency using `execution_locality_intersection`.

```
{
  executor_a execA;
  executor_b execB;

  auto concurrencyOverlap = execution::query(execA,
    execution::execution_locality_intersection(execB));
}
```

Listing 7: Example of using the concurrency property

Proposed Wording

The `execution_locality_intersection_t` property *(Listing 8)* is a query-only property as defined in P0443 [22].

```
struct execution_locality_intersection_t
{
  static constexpr bool is_requirable = false;
  static constexpr bool is_preferable = false;

  using polymorphic_query_result_type = size_t;

  template<class Executor, class DestExecutor>
    static constexpr decltype(auto) static_query_v
      = Executor::query(execution_locality_intersection_t{}(DestExecutor{}));

  template <class DestExecutor>
```

```
  size_t operator()(DestExecutor &&d);
};
```

Listing 8: Proposed specification for execution_locality_intersection_t

The `execution_locality_intersection_t` property can be used only with `query`, which returns the maximum potential concurrency available to both *executors*. If the value is not well defined or not computable, `0` is returned.

The value returned from `execution::query(e1, execution_locality_intersection_t(e2))`, where `e1` and `e2` are executors, shall not change between invocations.

> [*Note:* The expectation here is that the maximum available concurrency for an *executor* as described here is equivalent to calling `this_thread::hardware_concurrency()` *--end note*]

Memory locality intersection property

We propose a query-only executor property called `execution_locality_intersection_t` which specifies whether two *executors* share a common memory locality, such that memory allocated by those *executors* both have similar affinity.

This is useful for determining whether memory local to one *executor* would require migration in order to be local to another *executor*.

Example

Below is an example *(Listing 9)* of querying whether two *executors* have common memory locality `execution_locality_intersection`.

```
{
  executor_a execA;
  executor_b execB;

  auto concurrencyOverlap = execution::query(execA,
    execution::execution_locality_intersection(execB));
}
```

Listing 9: Example of using the concurrency property

Proposed Wording

The `memory_locality_intersection_t` property *(Listing 10)* is a query-only property as defined in P0443 [22].

```
struct memory_locality_intersection_t
{
  static constexpr bool is_requirable = false;
  static constexpr bool is_preferable = false;

  using polymorphic_query_result_type = bool;

  template<class Executor, class DestExecutor>
    static constexpr decltype(auto) static_query_v
      = Executor::query(memory_locality_intersection_t{}(DestExecutor{}));

  template <class DestExecutor>
  bool operator()(DestExecutor &&d);
};
```

Listing 10: Proposed specification for memory_locality_intersection_t

The `memory_locality_intersection_t` property can be used only with `query`, which returns `true` if both *executors* share a common address space, and `false` otherwise. If the value is not well defined or not computable, `false` is returned.

The value returned from `execution::query(e1, memory_locality_intersection_t(e2))`, where `e1` and `e2` are executors, shall not change between invocations.

Future Work

There are a number of additional features which we are considering for inclusion in this paper but are not ready yet.

Iteration space subdivision property

It is defined in this proposal for the `bulk_execution_affinity_t` properties that when the size in an invocation of `execution::bulk_execute` is greater than the *available concurrency* then it is implementation defined how that iteration space is subdivided into a consecutive sequence of work-items. The authors of this proposal intend to propose a follow up property for specifying how an iteration space should be subdivided into chunks in this case.

Migrating data

This paper currently provides a mechanism for detecting whether two *executors* share a common memory locality. However, it does not provide a way to invoke migration of data allocated local to one *executor* into the locality of another *executor*.

We envision that this mechanic could be facilitated by a customization point on two *executors* and perhaps a `span` or `mdspan` accessor.

Supporting different affinity domains

This paper currently assumes a NUMA-like system, however there are many other kinds of systems with many different architectures with different kinds of processors, memory and connections between them.

In order to accurately take advantage of the range of systems available now and in the future we will need some way to parameterize or enumerate the different affinity domains which an executor can structure around.

Furthermore, in order to have control over those affinity domains we need a way in which to mask out the components of that domain that we wish to work with.

However, whichever option we opt for, it must be in such a way as to allow further additions as new system architectures become available.

Acknowledgments

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

References

[1] P0687: 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] P0737 : 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

[35] Supporting Heterogeneous & Distributed Computing Through Affinity

[36] System topology discovery for heterogeneous & distributed computing