P2155r0: Policy property for describing adjacency

Date: 2020-04-15

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@appliantology.com, mhoemmen@stellarscience.com, tscogland@llnl.gov

Reply to: gordon@codeplay.com

Acknowledgements

This paper is the result of discussions from many contributors within SG1, SG14 and the heterogeneous C++ group.

Preface

After a lengthy discussion with SG1 during the Prague 2019 ISO C++ meeting it was decided to take a different approach to P1436r3 [1]. This paper is a continuation of prior work in P1436r3 [2] focusing specifically on the the `adjacency` properties, previously named `bulk_execution_affinity`.

Motivation

The typical computer systems targeted by the C++ abstract machine have a uniform address space, meaning a pointer can be dereferenced anywhere in the application. However, the cost to access different regions of that address space are most often not uniform. The position of a memory allocation relative to the thread which accesses the memory, otherwise referred to as memory affinity can have a significant impact on the performance of concurrent applications.

Most computer systems have numerous level of cache and many systems also have a Non-Uniform Memory Architecture (NUMA), where they have a single address space that is virtually mapped across a network of interconnected nodes. NUMA exists specifically because it is difficult to scale non-NUMA memory systems to the performance needed by today's highly parallel applications.

When a memory allocation is shared among threads, you can encounter false sharing depending on which threads are reading and writing that location. When this happens then the cost of reading from that location can be much higher than would ordinarily be expected. This is further complicated as operating systems often switch out hardware threads from one thread or process to another.

Furthermore, inter-node memory operations requires node to node communication and when there are a large number of operations this can quickly cause saturation.

As the C++ abstract machine has no concept of caches or the positioning of OS threads it is impossible for C++ applications to reason about this without using a third party programming model such as OpenMP [3].

One strategy to improve an applications' performance, given the importance of affinity, is thread and memory placement. Keeping a processes thread(s) 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 maximally utilizing cores and to exploit the full performance of the memory subsystem on NUMA architectures.

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 programmers 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 [3].

Proposal

This paper proposes a new group of properties (as described in P1393r0 [4]) that allow users to specify how a callable in a bulk algorithm such as the `bulk_execute` algorithm in P0443r12 [2] will perform in light of the placement of the work-items it invokes. This information can then be used by an executor implementation which is aware of the property to efficiently map the work-items it invokes to the underlying resources which it represents.

Shift to descriptive properties

The previous revision of this proposal; P1436r3 [1], took a prescriptive approach. It suggested the `bulk_execution_affinity` properties, which targeted executors, providing a hint to the executor implementation that it should aim to map the work-items invoked by a bulk algorithm to the underlying resources of the executor in scatter or gather pattern to achieve optimal performance for the callable being invoked.

This proposal switches perspective on this, and instead takes a descriptive approach. It suggests the `adjacency` properties, which instead target execution policies, providing a hint to the executor implementation which expresses the performance implications of the callable on the mapping of the work-items invoked by a bulk algorithm, allowing the implementation to choose the appropriate action.

This shift moves towards a more algorithmic-based approach to the customization of execution. It both removes the need for users to understand the internal workings of various executors and allows more freedom in executor implementations to choose how to interpret the hint.

The properties of the previous revision of this paper, `bulk_execution_affinity` , also included the `balanced` property which differentiated between round-robin and bin-packed mapping of work-items to the underlying resources. This variant is not reflected in this paper, as it requires some further investigation, so it will be continued in later revisions of [1].

Example

Below is an example of how the `adjacency` properties can be used with the `indexed_for` algorithm to perform first touch initialization. For the purposes of demonstration this example uses sender-based algorithms from P1897r2 [5].

```
// Create a vector of unique_ptr and reserve space for one for each index in the
// bulk algorithm.
std::vector<std::unique_ptr<float>> data{};
data.reserve(SIZE);

// Create a NUMA-aware executor.
numa_executor numaExec;

// Create an iota view to represent the iteration space of the bulk algorithm.
auto indexRng = ranges::iota_view{SIZE};

// Create a new execution policy which requires the adjacency.constructive
// property on the std::par execution policy so that it provides a hint to the
// executor that the algorithm would benefit from constructive interference in
// the mapping of work-items to execution resources.
auto adjacencyPar = std::execution::require(std::par, adjacency.constructive);

// Define a callable that will be invoked for each work-item in the bulk
// algorithm and perform first touch initialization.
auto initialize = [=](size_t idx, std::vector<unique_ptr<float>> &value) {
  value[idx] = std::make_new<float>(0.0f);
}

// Define a callable that will be invoked for each work-item in the bulk
```

```
// algorithm abd perform some computation.
auto compute = [=](size_t idx, std::vector<unique_ptr<float>> &value) {
    do_something(value[idx]);
}

// Compose a sender that takes the input data, schedules using the NUMA-aware
// executor and then calls the indexed_for algorithm twice, once to initialize
// and then again to perform the computation.
auto sender = std::execution::just(data)
            | std::execution::via(numaExec)
            | std::execution::indexed_for(indexRng, adjacencyPar, initialize)
            | std::execution::indexed_for(indexRng, adjacencyPar, compute);

// Submit the sender and wait on the result.
std::execution::sync_wait(sender, std::execution::sink_receiver{});
```

Listing 1: Using the `adjacency properties on a NUMA-aware executor

Execution resources

As a pre-requisite for the `adjacency` properties, this paper also proposes a definition for the underlying resources of an executor.

Proposed wording

Synopsis

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

struct adjacency_t {

  struct no_implication_t;
  struct constructive_t;
  struct destructive_t;

  constexpr no_implication_t no_implication;
  constexpr constructive_t constructive;
  constexpr destructive_t destructive;

};

constexpr adjacency_t adjacency;

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

Listing 2: Addition to execution header synopsis

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. 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 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.

Property group `adjacency`

The `adjacency_t` properties are a group of mutually exclusive behavioral properties which provide a hint to an executor, via an *execution policy* which describes the implication of adjacent work-items of the callable of a bulk algorithm, allowing the implementation to choose the appropriate action.

Locality interference specifies an implementation-defined metric for measuring the relative interference between *execution resources*. *Execution resources* with a higher *locality interference* with each other share more common memory resources, connections or subsumptions of a hierarchy, and have similar latency or bandwidth in memory access operations, for a given memory location, relative to other *execution resources*.

> [*Note:* For example, two hardware threads on the same CPU core would have a higher *locality interference* whereas two hardware threads on different CPU cores or in different NUMA regions would have a lower *locality interference*. *--end note*]

When a bulk algorithm is invoked with a callable `f`, a size `s` and using a policy `p`, where `query(p, adjacency_t) == adjacency_t::constructive_t`, `p` must communicate a hint to the executor being scheduled on that any adjacent work-items in the sequence `0` to `s-1` which are mapped to *execution resources* with higher *locality interference* will result in higher constructive interference for `f`, relative to other *execution resources*.

When a bulk algorithm is invoked with a callable `f`, a size `s` and using a policy `p`, where `query(p, destructive_t) == adjacency_t::destructive_t`, `p` must communicate a hint to the executor being scheduled on that any adjacent work-items in the sequence `0` to `s-1` which are mapped to *execution resources*

with higher *locality interference* will result in higher destructive interference for `f`, relative to other *execution resources*.

> [*Note:* Subsequent invocations of a bulk algorithm with the same size `s` and the same policy `p`, scheduling on the same executor `e` should aim to map work- items to *execution resources* consistently with previous invocations. --*end note*]

> [*Note:* It's expected that the default value of `adjacency_t` for most executors be `adjacency_t::no_implication_t`. --*end note*]

References

[1] Executor properties for affinity-based execution

[2] A Unified Executors Proposal for C++

[3] The Design of OpenMP Thread Affinity

[4] A General Property Customization Mechanism

[5] Towards C++23 executors: A proposal for an initial set of algorithms