# P0836R1 Introduce Parallelism to the Ranges TS

| | | |
|---|---|---|
| **Date:** | 2018-05-07 (Rapperswil) | |
| **Audience:** | LEWG, SG9, SG1, SG14 | |
| **Authors:** | Gordon Brown | <gordon@codeplay.com> |
| | Christopher Di Bella | <christopher@codeplay.com> |
| | Michael Haidl | <michael.haidl@uni-muenster.de> |
| | Toomas Remmelg | <toomas.remmelg@codeplay.com> |
| | Ruyman Reyes | <ruyman@codeplay.com> |
| | Michel Steuwer | <michel.steuwer@glasgow.ac.uk> |
| | Michael Wong | <michael@codeplay.com> |
| **Reply-to:** | Ruyman Reyes | <ruyman@codeplay.com> |

## Abstract

The current interface provided by the C++17 parallel algorithms poses some limitations with respect to parallel data access and heterogeneous systems, such as personal computers and server nodes with GPUs, smartphones, and embedded System on a Chip chipsets. This paper offers a summary of why we believe the Ranges TS[7] solves these problems, and also improves both programmability and performance on heterogeneous platforms. To the best of our knowledge, this is the first paper presented to WG21 that unifies the Ranges TS with the parallel algorithms introduced in C++17. Although there are various points of intersection, in this paper, we will focus on the *composability of functions*, and the benefit that this brings to accelerator devices via kernel fusion.

Codeplay previously discussed the challenges regarding data movement when programming heterogeneous and distributed systems in C++ during the C++ Standard meeting in Toronto in 2017[16]. Another issue is the difficulty of implementing parallel algorithms on heterogeneous systems. In this paper, we present the Ranges TS as a potential way to alleviate the programmability issue, and to also introduce some performance benefits on these platforms.

# Table of Contents

# 0 Change log

## 0.1 Changes from P0836R0 to P0836R1

- Added §1.3, which explains why non-standard-layout types are problematic.
- Added clarification to the example §2.1.
- Added discussion point for mixing execution policies in §2.1.
- Added §2.2.1, which explains why views and kernel fusion matter
    - Moved old §2.2.1 to §2.2.2, etc.
- Revised wording for §2.4 to explain why the Ranges TS does not target the parallel algorithms, and improve the wording.
- Removed the request for a `StandardLayout` concept in §3.1.1.
    - Added request for considering standard-layout types (e.g. for `std::tuple`).
- Removed support for the type-trait `std::is_contiguous_iterator` in §3.2.1.
- LEWG's decision to abandon namespace `std2` in Jacksonville requires the abandonment of §3.2.2 Add `contiguous_iterator_tag` to `std2`.
- Relaxed request for standard-layout types from authors of the Ranges TS to LEWG in §4, as this request is beyond the scope of the Ranges TS.
    - Introduced motivation for why standard-layout types are important for heterogeneous programming.
- Mentioned that research into reflection with respect to private members is recommended.

# 1 The problems with the C++ parallel algorithms' interface with Data Movement

## 1.1 The Problem of Data Movement in Parallel Algorithms

```
#include <vector>
#include "pstl/execution"
#include "pstl/algorithm"

int main()
{
   std::vector<int> data(10'000'000);
   std::fill_n(std::execution::par_unseq, begin(data), size(data), -1);
}
```

**Listing 1.1** A trivial example of the current parallel algorithms, as used in C++17, extracted from the Getting Started guide of the Intel Parallel STL.

Using a parallel execution policy, *Listing 1.1* fills a `std::vector` from the beginning of the vector to its end with the value `-1`. Since this is executing on the CPU, the individual threads of execution executing under the parallel execution policy can access the vector directly and concurrently, using a mutex or atomic operations to avoid race conditions. This particular algorithm's operation is embarrassingly parallel; in principle, no

synchronisation is necessary. Due to the usage of the `par_unseq` policy, the compiler will also try to use vectorisation to optimise operations: more than one element of the vector will be accessed from the same thread using SIMD instructions.

P0443 proposes a unified interface for execution[17], which allows this algorithm to be executed using a particular executor, and therefore in a particular execution context[1]. Executors can be used to instruct an algorithm to execute on a GPU. The exact interface for this is yet to be decided; however, a popular design described in the executors design document[5], is to have the execution policy extended with a `.on(Executor)` member function to achieve this goal.

```
#include <vector>
#include "pstl/execution"
#include "pstl/algorithm"
#include "experimental/execution"

int main()
{
   std::vector<int> data(10'000'000);
   std::fill_n(std::execution::par_unseq.on(gpu_executor()), begin(data),
      size(data), -1);
}
```

**Listing 1.2** How *Listing 1.1* might look if we were to target a GPU using an executor.

*Listing 1.2* creates what is referred to as an inline executor that describes how work is to be executed on a particular GPU. The proposed interface for executors[17] provides a range of parameterised executors for executing work in a variety of ways. These executors can then be used by control structures such as `std::async`, `std::invoke`, or in this case, parallel algorithms.

An implementer of `std::fill_n` for a distributed or heterogeneous system has limited information on the input data. Iterator categories specify how data may be accessed, but they offer neither any insight into where the data is created, nor how it is stored. For example, forward iterators offer the guarantee of multi-pass iterators (within some range); but do not tell us if we are accessing objects stored at arbitrary points in memory (e.g. `std::forward_list`, `std::map`), or if the data is stored contiguously (e.g. `std::vector`, `std::array`). Similarly, random-access iterators provide just as little information: we are not privy to whether the data is stored contiguously (e.g. `std::vector`), or if it is stored in a non-contiguous layout with random-access (e.g. `std::deque`).

Depending on the distributed or heterogeneous system being targeted, additional copies to memory or accessing memory that is not local to the processor may be necessary; both of these can have critical impacts to performance. When the execution of the parallel algorithm is dispatched to a processing unit of the heterogeneous and distributed system, the whole range may need to be transferred alongside the dispatch. A Parallel STL implementation is forced to first instantiate the entire iteration range on the calling thread, storing each element into temporary storage[2], then send the data to the processing unit for the processing to complete. Since the algorithm modifies the elements accessed, data then must be sent back to the caller of the algorithm, collected in temporary storage, and then applied again to the specified iterator range.

[1] An execution context is an abstraction of a particular execution resource.
[2] Not necessarily a temporary object.

This assumes that there are no side effects on iterator succession, which is only possible on forward iterators. Input iterators do not guarantee multiple passes because they can suffer from invalidation. But this is required for heterogeneous dispatch. A defect that was fixed before the Parallelism TS was merged into C++17 was to refine the input iterator requirement to forward iterators, to guarantee multi-pass iterator succession[13a].

A possible solution is to allow only contiguous iterators for those policies that target heterogeneous and distributed systems. Contiguous data can be sent directly to the processing unit and written back without extra intermediate storage. Current implementations are unfortunately unable to detect the use of contiguous iterators, as `std::contiguous_iterator_tag` is not present in the C++ Standard.

## 1.2 A SYCL Solution

This paper is mainly based on SYCL since it is the implementation choice for this prototype. However it is important to note that the concepts we describe here and, to some extent, the benefits are applicable to non-heterogeneous systems as well.

The SYCL Parallel STL[9] implementation implements helper-functions to avoid passing non-contiguous ranges directly to the SYCL buffer objects to store the data. This paper won't dive into the specifics of the SYCL programming model too much, more background resources can be found on the Khronos site.

```
std::vector<int> v = {3, 1, 5, 6};
cl::sycl::sycl_execution_policy<> sycl_policy;

using std::begin, std::end;
{
   cl::sycl::buffer<int> b(data(v), cl::sycl::range<1>(size(v)));
   std::sort(sycl_policy, begin(b), end(b));

   auto h = cl::sycl::gpu_selector{};
   {
      auto q = cl::sycl::queue{h};
      auto sepn1 = cl::sycl::sycl_execution_policy<class transform1>(q);
      std::transform(sepn1, begin(b), end(b), begin(b), [](const auto num) {
         return num + 1; });

      cl::sycl::sycl_execution_policy<std::negate<>> sepn4(q);
      std::transform(sepn4, begin(b), end(b), begin(b), std::negate<>{});
   } // All kernels will finish at this point
} // The buffer destructor guarantees host synchronization
```

**Listing 1.3** `std::transform` being used with the SYCL execution policy.

Algorithms that have been passed a SYCL execution policy are executed on the SYCL device that has been selected by the implementation. When passing a range of iterators to the helper-functions, they create an intermediate storage that is guaranteed to be contiguous (by performing an extra copy), and the SYCL implementation can use this pointer in an implementation-specific way that is optimal for the platform memory architecture. The different policies operate on iterators to the SYCL buffer instead of iterators to the

original container (a vector in *Listing 1.3*). This additional copy causes extra overhead that is not required when the range is contiguous (since the SYCL runtime can simply map the pointer directly).

SYCL mandates that the SYCL runtime retains an opaque ownership of the data until the buffer is destroyed. Upon the buffer's destruction, ownership is transferred back to the original vector. Iterators to SYCL buffers contain information about where the data resides. The data flow execution rules guarantee that data is available on the different processing units.

The SYCL Parallel STL implementation uses iterators only as offsets from the start of the buffer: the buffer is retrieved from the iterator to perform the actual operations.

```
/* fill.
 * Implementation of the command group that submits a fill kernel.
 * The kernel is implemented as a lambda.
 */
template <typename ExecutionPolicy, typename ForwardIt, typename T>
void fill(ExecutionPolicy& sep, ForwardIt b, ForwardIt e, const T& value) {
   auto q = cl::sycl::queue{sep.get_queue()};
   auto device = q.get_device();
   auto localRange =
      device.get_info<cl::sycl::info::device::max_work_group_size>();
   auto bufI = helpers::make_buffer(b, e);
   auto vectorSize = bufI.get_count();
   auto globalRange = sep.calculateGlobalSize(vectorSize, localRange);
   q.submit([vectorSize, localRange, globalRange, &bufI, val = value](
      cl::sycl::handler &h) mutable {
      cl::sycl::nd_range<1> r{
         cl::sycl::range<1>{std::max(globalRange, localRange)},
         cl::sycl::range<1>{localRange}
      };
      auto aI = bufI.template get_access<cl::sycl::access::mode::read_write>(h);
      h.parallel_for<typename ExecutionPolicy::kernelName>(r,
         [aI, val, vectorSize](cl::sycl::nd_item<1> id) {
            if (id.get_global(0) < vectorSize) {
               aI[id.get_global(0)] = val;
            }
      });
   });
}
```

**Listing 1.4** SYCL Parallel STL implementation for `std::fill`.

In *Listing 1.4*, the helper-function, `make_buffer`, detects when the iterators are SYCL buffer iterators and retrieves the original buffer used as input. Iterators that are not SYCL buffer iterators have their range copied on to a newly constructed buffer. The implementation details can be found in [sycl_buffers.hpp]. No assumptions can be made about the layout of the input range data; as such, temporary storage and copies must be used, which causes a noticeable performance drawback.

Copying arguments was not permitted in the Parallelism TS, but this prohibition was lifted through a Swiss National Body comment[6] which cited SYCL as one important use case to support copying. This allows parallel algorithm arguments to function objects to be copied to a separate space for non-sequenced policies.

As of C++17, the Standard defines a set of forward progress guarantees [intro.progress]: concurrent forward progress, parallel forward progress, and weakly parallel forward progress. Due to the nature of many processing units such as GPUs, there are many heterogeneous and distributed devices which can only guarantee weakly parallel forward progress.

## 1.3 Issues with non-standard-layout types

Heterogeneous code runs on multiple devices, and each device needs to communicate with other devices. There is no common ABI for C++, which makes it difficult for programs compiled for different architectures, using different toolchains, to marshall data. For example, a host CPU program might be compiled with Microsoft Visual C++, one device program could be built with LLVM, and another device with GCC. The International Standard specifies standard-layout types in such a way that data is consistently positioned across all implementations, and this means we can marshall data across different architectures. Short of implementing a stricter ABI for the standard library, standard-layout types are the only way we can ensure that communication between various devices remains consistent without manually restructuring the data, which can be quite costly.

# 2 Contrasting ranges, views, and actions with iterators

As previously mentioned in P0687R0, the already-presented limitations of the iterator-based interface have a significant impact on performance on systems where memory must be copied before applying the algorithms to the data, which can be problematic. A potential solution to this problem is to use the interface introduced in the Ranges TS and Eric Niebler's current proposal for range adaptors and utilities[15a]. Among several other modifications, the range-based algorithms' interface replaces iterator-pairs with ranges. A range is a concept that defines the requirements of a type that allows iteration over its elements by providing a `begin` iterator and a sentinel object[14a]. In a general context, a range is typically a container, but is not required to be one (for example, an input iterator abstracting over `std::istream` is a range if it is associated with some object denoting the end of the input sequence).

The Ranges TS also revises the C++14 Standard algorithms, so that they provide iterator-sentinel pairs instead of homogeneously-typed iterator-pairs. This means that we no longer need to specify the end of a range using an iterator, and can use other objects instead; provided that the object has some sort of tangible relationship with our iterator type. One such example where this is useful is doing something to the first *n* elements of a range. The C++ standard library approximates this with functions such as `std::fill_n`, but completeness requires providing this for every algorithm, and that is an enormous number of additional overloads that need to be ratified, implemented, confirmed to be correctly implemented, and maintained[3].

```
template <typename ForwardIterator, typename N, typename T>
ForwardIterator find_n(ForwardIterator first, N count, T const& value)
{
   return std::find(first, std::next(first, count), value);
}
```

**Listing 2.1** Implementation of `find_n` using Standard algorithms.

```
namespace ranges = std::experimental::ranges;
template <ranges::InputIterator I, typename T>
I find_n(I first, ranges::difference_type_t<I> count, T const& value)
{
   return ranges::find(ranges::make_counted_iterator(first, count),
      ranges::default_sentinel{}, value);
}
```

**Listing 2.2** Implementation of `find_n` using the Ranges TS.

Eric Niebler's range-v3 library is a cross-platform, experimental implementation of the Ranges TS, and is compatible with C++11. Unlike the Ranges TS reference implementation, cmcstl2, range-v3 does not rely on the Concepts TS, and is thus suitable for use with implementations that don't support Concepts.

Central to range-v3 are views and actions, which improve algorithm composability, and allow users to construct pipelines of operations using `operator|`. Views behave as range-based algorithms; but unlike algorithms, lazily perform non-modifying computation only when requested. Actions represent mutating operations and perform in-place modifications to ranges. P0789 proposes adding views to the Ranges TS.

## 2.1 Example use-case

Given two vectors, **x** and **y**, and a scalar $\alpha$, the result of the BLAS (Basic Linear Algebra Subprograms) primitive, saxpy, is defined as $\alpha\mathbf{x} + \mathbf{y}$. A natural way of writing this using the STL is to scale **x** using `std::transform`, and then add the scaled vector with **y** using a second call to `std::transform`.

```
std::vector<float> x = // ...
std::vector<float> y = // ...
float a = // ...

auto out = std::vector<float>(size(x));
{
   auto temp = std::vector<float>(size(x));
   std::transform(begin(x), end(x), begin(temp), [a](const auto x) {
      return a * x; });

   std::transform(begin(temp), end(temp), begin(y), begin(out), std::plus<>{});
}
```

**Listing 2.3** Slow-path saxpy implementation using STL.

Notice the use of the temporary vector `temp` in *Listing 2.3*: this can be expensive on accelerators. The temporary vector not only requires additional memory, but the executing code also performs additional stores and loads to access the temporary which can hinder performance. To avoid the temporary variable, the scaling operation can be manually done in a single transformation operation together with addition.

```
std::vector<float> x = // ...
std::vector<float> y = // ...
float a = // ...

auto out = std::vector<float>(size(x));
{
   std::transform(begin(x), end(x), begin(y), begin(out), [a](auto x, auto y) {
      return a * x + y; });
}
```

**Listing 2.4** Fast-path saxpy implementation using a single transformation with addition.

A similar problem arises when performing the dot-product operation on two vectors. To avoid a temporary variable, the programmer needs to be aware that `std::inner_product` is a more suitable alternative to a combination of first `std::transform` and then `std::accumulate`. This demonstrates the lack of composability of multiple STL operations. In this case, we are lucky that there is a composite algorithm.

However, there is not always a predefined function -- such as `std::inner_product` -- available for all the possible cases that application developers can concoct with Standard algorithms. For example, `std::transform` and `std::transform_reduce` are limited to one or two input ranges. If a user wants to combine more ranges in a single call, they are required to convert the input data to be formatted as an array of structures (AoS), and then apply `std::transform`.

On CPUs, the AoS format hinders the compiler's ability to perform vectorisation -- as allowed by `std::execution::par_unseq` -- since vector registers can typically only hold homogeneous data; thus, the AoS conversion might have negative performance implications. On GPUs, this format causes uncoalesced memory-accesses, since threads will be performing non-contiguous memory accesses.

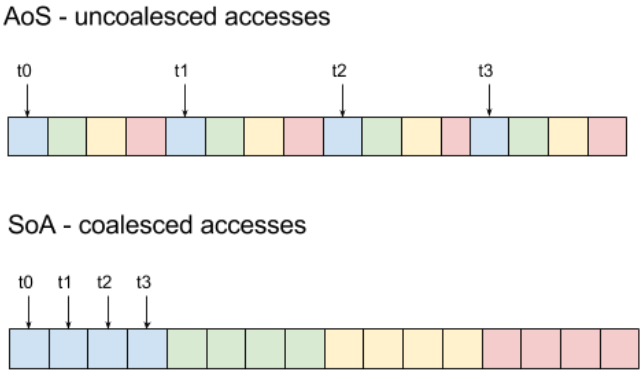


**Figure 2.1** Access pattern differences on GPU memory when using AoS vs SoA. Coalesced access is preferred for performance reasons.

Range-based algorithms and views enable developers to compose the provided algorithms more flexibly, and simultaneously offer the opportunity to increase performance by eliminating temporary storage. The lazy nature of views also enables automatic fusion of multiple algorithms into a single, efficient, computational kernel when using heterogeneous systems. By providing a compile-time function-composition mechanism, device-kernels can be generated to minimise register pressure on custom algorithms that have been written by

developers. This kernel-fusion technique is widely used in libraries, such as Eigen, to improve performance of various computational kernels.

```
std::vector<float> x = // ...
std::vector<float> y = // ...
float a =              // ...
auto ax = ranges::view::zip(view::repeat(a), x) | ranges::view::transform(mult);
auto out = ranges::view::zip(ax, y)
         | ranges::transform(plus)
         | ranges::to_vector;
```

**Listing 2.5** saxpy implementation using range-v3

A range-based interface also mitigates the problem of ensuring that an iterator-pair address the same range. A range has a beginning and a sentinel object to describe its termination, and the contents of the range can be transparently converted using range-based actions and views.

## 2.2 Implementation prototype of Parallel STL with Ranges

We have implemented a prototype of some parallel algorithms using range-v3 and SYCL. This builds upon work presented in an academic paper[4].

```
std::vector<float> x = // ...
std::vector<float> y = // ...
float a = // ...

std::vector<float> out(size(x));
{
   gstorm::sycl_exec exec;

   using std::experimental::copy; // to make the example smaller
   auto ax = ranges::view::zip(ranges::view::repeat(a), copy(exec, x))
           | ranges::view::transform(mult);
   std::experimental::transform(exec, ranges::view::zip(ax, copy(exec, y)),
      copy(exec, out), plus);
}
```

**Listing 2.6** `saxpy` implemented using SYCL and range-v3.

We first create a SYCL execution policy[3], `exec`. It provides parallel implementations of Standard algorithms, and wraps a SYCL queue attached to a device, so that it can enqueue work. `std::experimental::copy` internally creates a SYCL buffer that keeps track of the usage of memory, and holds all the required metadata. The SYCL buffer then provides access to the data from the different accelerators via accessors. Our prototype

[3] Mixing different execution policies prevents kernel fusion. This should be explicitly noted in the IS (or any TS resulting from this document). We should hold a poll for audience preference to determine how to proceed.

implements a wrapper that provides an `InputRange`-compatible interface for SYCL buffers to allow them to be used as input to views.

The calls to `copy` trigger the constructor of the underlying SYCL buffer, so that `exec` is bound to the context for guiding storage. Using a `gstorm::sycl_exec` execution policy is deliberate: given that the Standard execution policies are not aware of heterogeneous systems, they do not suffice for our purposes. Rebinding `exec` avoids extra copies when involving data movement.

`Saxpy` is then implemented as it is in *Listing 2.5*. Since views are lazy and don't perform any computation, we change the final `ranges::view::transform` to a call to `std::experimental::transform` on the execution policy, to execute the resulting operation and enqueue the kernel onto the device. It is important to understand that this code will only execute a single kernel on the device, despite the use of four view algorithms to describe the computation. As views never execute eagerly, and only algorithms and action do, this gives a very easy to understand cost model to the programmer.

The lifetime of SYCL objects ends at the end of the enclosing scope; following their lifetime rules[4], any data modified on the device will be updated on the host.

Views from range-v3 access data using iterators from the provided input range. To access device memory, iterators need to use SYCL accessors when dereferenced. We implement a SYCL-aware wrapper, `gvector`[5], that allocates a SYCL buffer, and registers itself with the execution policy. The execution policy will provide registered `gvector`s with `cl::sycl::handlers` when launching a kernel, so that they can create accessors to be used in device code, and return iterators from them using `begin` and `end`. The authors intend to propose to extend the SYCL buffer to implement the behaviour currently implemented by `gvector`.

The limitations of the SYCL programming model, as imposed by the nature of heterogeneous dispatch and multiple device support, required us to make changes to range-v3 as follows:

- Non-standard-layout `std::tuple` was replaced with an implementation that does.
- Making `ranges::view::chunk` standard-layout by removing `ranges::box` base class.
- Removing the pointer field in `ranges::view::cycle` to make it usable in SYCL.
- Adding some `constexpr` specifiers.

The limitations of the SYCL programming model impose certain restrictions on device-code:

- Cannot throw in device-code.
- Cannot use views with pointer fields or non-standard-layout types, without modification.
- Views need to support random-access for parallel usage in SYCL. This means that views such as `filter` and `remove_if` cannot be used as input to SYCL parallel algorithms without first instantiating their result, as they are specified as at most bidirectional.

### 2.2.1 Why views and kernel fusion matter

Let us assume a three-operation process is being executed on some non-CPU device. Without kernel fusion:

1. Data must be transferred from the CPU to the device.
2. The first operation is performed on the data.
3. The data is sent back to the CPU from the device.
4. Data is, again, transferred from the CPU to the device.
5. The second operation is performed on the data.
6. The data is sent back to the CPU from the device.

[4] These rules can be found in the SYCL 1.2.1 specification.
[5] `gvector` is short for 'GPU-vector'.

7. Data is, again, transferred from the CPU to the device.
8. The third operation is performed on the data.
9. The data is sent back to the CPU from the device.

This factor-of-three sequence of events is riddled with overhead. Before any device operation can start, the previous operation must send its data back to the CPU, only for it to be sent back to the device for more processing.

With their lazy computation, views naturally achieve kernel fusion, and so are very closely coupled with the benefits that kernel fusion offers us. All performance benefits described in subsequent sections are achieved because of kernel fusion, which is only made possible through views.

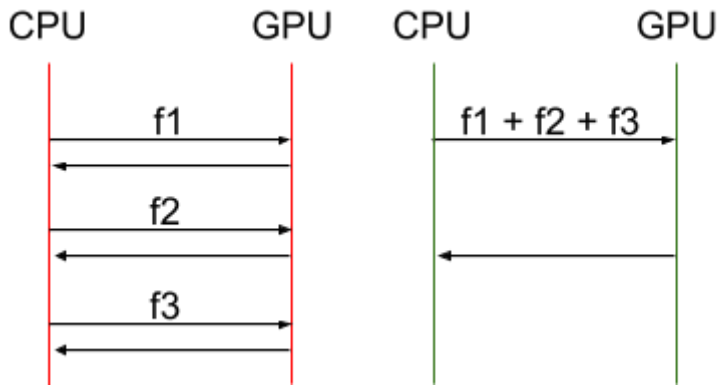


**Figure 2.2:** Graphical description of how kernel fusion works.

### 2.2.2 Calculating the Mandelbrot Set

We have run a number of performance measurements and compared our implementation to the Intel Parallel STL implementation. In particular, we would like to highlight the results we obtained from calculating the mandelbrot set. As input, it takes input image pixel locations and outputs an image containing a colorised visualisation of the mandelbrot set.

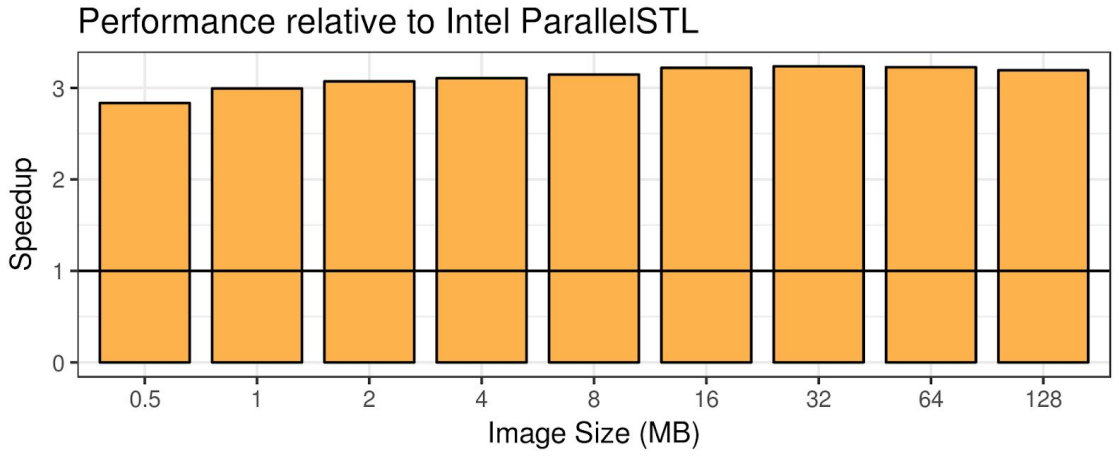


**Figure 2.3** Speedup on an Intel i7-6700K CPU

As seen in *Figure 2.3*, the SYCL ranges version of mandelbrot is significantly faster than the Intel Parallel STL, even though the Intel Parallel STL is based on Intel Threading Building Blocks that is highly optimised for CPUs. This is because the STL does not have a specialised fused function for `std::iota` with `std::transform`, and because there is no parallel version of `std::iota`. This means there are two library calls, one of which is always sequential.

## 2.3 Containers

SYCL buffers refer to a contiguous region of memory in the host. As devices can use physically distinct memory spaces to the host, SYCL 1.2.1 specifies that host pointers are invalid on the device, and may not be captured as arguments to the kernel[6]. It is also impossible to create a buffer of pointers for the same reason. Trees, graphs, linked-lists, and other sparse data structures need to be transformed to usable structures on the device.

Both SYCL 2.2 and OpenCL 2.0 support a feature known as *shared virtual memory* (SVM), which allows restricted sharing of pointers between host and devices by providing a shared address space. Due to the high cost of implementing SVM on either hardware stacks or software stacks, it is not widely available, and so a large number of vendors still offer OpenCL 1.2 instead of OpenCL 2.0.

Different versions of the CUDA programming model support partial or complete sharing of pointers between host and device, but users are still advised to use separate address spaces for performance reasons in some cases (for example, UMA performance analysis[10]).

The only containers to provide access to their underlying data are `std::array`, `std::vector`[7], strings, and string views. As of C++17, these four templates are specified as contiguous containers, use 'flat' arrays for storage, and provide the same interface for exposing the underlying storage; this makes them easy to use with SYCL. Contrastingly, a nested `std::vector<std::vector<float>>` needs to be flattened for use with SYCL.

Although `boost::container::flat_{map, set, multimap, multiset}` use flat layout for storage (using `boost::container::container_detail`, and therefore `boost::container::vector`), they do not provide access to the underlying storage. To use those containers, a heterogeneous system with no SVM access needs to traverse the entire range, copy the contents to device memory, and possibly transform their layout to an optimal structure. Even with SVM access, the architectural differences between the CPU and the accelerators will require different data structures. Views or actions can potentially transform the layout of source data structures to an optimal target structure, but this is not in the scope of this paper. We aim to demonstrate that ranges, views, and actions can be used to ease developer productivity when dealing with heterogeneous systems.

## 2.4 Concepts for parallel algorithms, views, and actions

The scope of the Ranges TS did not encompass the parallel algorithms, as they were not a part of C++14 (which the TS is based upon). Parallel algorithms, views, and actions may have different requirements to their serial counterparts. This paper doesn't propose any concepts for parallel ranges, but it does serve as a call for collaboration on concept design. Readers interested in contributing to concepts for parallel ranges might be interested in reading Chapter 33 of GPU Gems 2, which outlines the design of several GPU-friendly data structures.

[6] The same is true for OpenCL 1.2, which is the basis for SYCL 1.2.1.
[7] This excludes `std::vector<bool>`.

# 3 Proposals

## 3.1 Introduce requirements for standard-layout types where possible

The only available guarantee that heterogeneous programming models currently have for sharing data across different compiler toolchains and ABIs is standard-layout types. Until other guarantees are introduced to the International Standard that make it trivially possible for sharing data between multiple implementations, the following additions to C++ be instrumental in describing clear and concise heterogeneous programs.

### 3.1.1 Investigate which types can be required to be standard-layout

Given the problems outlined in §1.3 and §2.2, we would like to suggest requiring types (such as `std::tuple`) be standard-layout wherever possible. One of the benefits of tuple not being standard-layout is the size optimisation. However, all three major implementations choose different algorithms for that, and none of them reorder to a better layout to minimise compile-time size; as such, this benefit is wasted.

## 3.2 Add `contiguous_iterator` type-traits to C++

Contiguous iterators -- as they are in C++17 -- are not identifiable by any means, which makes them useless in all-but-theory, as there is no way to guarantee that an iterator holds an address to objects that are contiguously stored in memory.

Casey Carter published *P0944 Contiguous Ranges*[1] in the Jacksonville 2018 pre-mailing, which seeks to extend support for contiguous ranges in the Ranges TS, as a refinement of random access ranges. We support P0944R0, and wish to offer the findings outlined above as additional motivation, if necessary.

# 4 Conclusion

In this paper, we have presented some problems with the existing parallel algorithms interface when targeting heterogeneous systems, and the current SYCL Parallel STL solutions. We introduce a prototype implementation where we use range-v3 to show how range-based algorithms, views, and actions leverage many of the problems with the current parallel algorithms' interface, and enable further optimisations, such as kernel fusion (compile-time functor composability) that are not possible with the iterator interface.

We encourage the C++ Standardisation community to consider including views and actions in the next C++ Standard to facilitate adoption of C++ on heterogeneous platforms. Ranges without views and actions are insufficient to address the problems faced on these platforms.

We would like to ask that requirements be made for standard-layout types whenever possible, such as starting with `std::tuple`. Standard-layout types are currently the only guarantee that a programming model for heterogeneous systems can enforce so that data can be shared across different compilers and ABIs. Non-standard-layout types cannot be copied to a device 'as-is' for which code is compiled with a different compiler toolchain.

Finally, as an intermediate step, we encourage the Library Evolution Working Group to consider adding a mechanism for identifying contiguous iterators to the Standard for C++, so that Parallel STL implementations can detect whether iterator ranges are contiguous and can assume that data can be directly and continuously

accessed through a pointer. We also encourage LEWG to consider extending the Ranges TS to support a `ContiguousIterator` concept and a `ContiguousRange` concept. P0836 actively supports the direction P0944 suggests, as noted in §3.2.1.

# 5 Future work

Future work will continue to explore the combination of parallel algorithms with ranges, with special attention paid to fusion. We would like to explore other topics, such as data layout transformation, and concept definitions that would be meaningful for parallel algorithm implementations that target non-CPU architectures.

The authors, and the SYCL group in Khronos, will continue to work with the SYCL Parallel STL implementation, exploring the different issues that heterogeneous computing present to the C++ standard. We are looking forward to feedback to our ideas but also more general collaborations on any aspect that may facilitate the programmability of heterogeneous systems in C++.

It was also mentioned in Jacksonville that support for reflection, especially with respect to private members, is a powerful solution to our problems. We would like to recommend research be undertaken to explicitly demonstrate how static reflection can benefit heterogeneous programming.

Finally, this work has identified that typical implementations of `std::tuple` may not be suitable for heterogeneous programming. We believe there is a problem with allowing std::tuple to be non-standard-layout, which has been sufficiently exposed by our endeavour to marry parallel programming and the Ranges TS, and we would like to investigate ways to refine `std::tuple` -- and possibly other types -- so that they become suitable for heterogeneous programming.

# Appendix A Acknowledgements

We would like to thank Casey Carter, Morris Hafner, and Eric Niebler for their feedback on this paper.

We also offer our thanks to our employer, Codeplay Software, for providing time to research these issues and present this proposal to the committee.

# Appendix B Glossary

| Term | Definition |
|---|---|
| Accessor | An accessor is a class which allows a SYCL kernel function to access data managed by a buffer. |
| Action | An eagerly executed algorithm which mutates the state of its input range. It is designed to be composable by returning references to mutable ranges. |
| Buffer | The buffer class manages data for the SYCL C++ host application and the SYCL device kernels. The buffer class may acquire ownership of some host pointers passed to its constructors. |
| Command group handler | The command group handler class provides the interface for the commands that can be executed inside the command group scope. |
| Compile-time kernel fusion | A form of kernel fusion involving combining multiple components into a single kernel via expression templates. |
| Embarrassingly parallel | A task or problem in which a computation is performed on a series of data independently without any need for communication between each computation. |
| Device | A SYCL device encapsulates an OpenCL device or the SYCL host device, which can run SYCL kernels on the host. |
| Device memory | Refers to memory address spaces local to one or more heterogeneous devices. In the case of multiple devices, each device has its own address space. |
| Host | The host is the system that executes the C++ application including the SYCL API. |
| Kernel | A function which is compiled specifically for executing on a particular heterogeneous device. |
| Kernel fusion | The process of merging one or more kernels together into a single kernel in order to avoid overhead from offloading. |

| | |
|---|---|
| Offloading | The process executing a kernel on another heterogeneous device. This usually involves overhead in copying the kernel itself and any data dependencies to the target device. |
| Queue | A SYCL queue schedules kernels to be executed by a SYCL device. |
| Run-time kernel fusion | A form of kernel fusion involving manipulating kernel sources (as strings) at runtime. |
| Shared virtual memory | An address space exposed to both the host and the devices within a context. SVM causes addresses to be meaningful between the host and all of the devices within a context and therefore supports the use of pointer based data structures. |
| View | A lazily-executed algorithm which does not mutate the state of its input range. It is designed to be composable by returning ranges which lazily apply the computation specified by the view, only once they are evaluated. |

# Appendix C References


1. Carter, C. *P0944R0 Contiguous Ranges*. 2018-02-11. wg21.link/p0944.
2. Dawes, B. *P0872R0 Discussion Summary: Applying Concepts to the Standard Library*. 2017-11-25. wg21.link/p0872.
3. Di Bella, C J. *Christopher Di Bella's answer to 'What do you think of the C++'s Range TS?'*. 2018-01-02. Last revised: 2018-01-25. http://bit.ly/2nsUvvu.
4. Haidl et al. (2017). Towards Composable GPU Programming: Programming GPUs with Eager Actions and Lazy Views. *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*. http://bit.ly/2C0sX63. pp58-67.
5. Hoberock et al. *P0761R1 Executors Design Document*. 2017-10-16. wg21.link/p0761r1.
6. Hollman et al. *P0518R1 Allowing copies as arguments to function objects given to parallel algorithms in response to CH11*. 2017-03-01. wg21.link/p0518r1.
7. ISO/IEC JTC1/SC22/WG21. (2017). *ISO/IEC 21425:2017, Programming Languages -- C++ Extensions for Ranges*.
8. ISO/IEC JTC1/SC22/WG21. *N4713 Working Draft, Standard for Programming Language C++*. 2017-11-27. wg21.link/n4713.
9. Khronos Group. *KhronosGroup/SyclParallelSTL*. https://github.com/KhronosGroup/SyclParallelSTL.
10. Landaverde, R et al. (2014). An investigation of Unified Memory Access performance in CUDA. *HPEC 2014*. http://bit.ly/2nsUvvu. pp1-6.
11. Liber, N. *N3884 Contiguous Iterators: A Refinement of Random Access Iterators*. 2014-01-20. wg21.link/n4883.
12. Liber, N. *N4183 Contiguous Iterators: Pointer Conversion & Type Trait*. 2014-10-10. wg21.link/n4183.
13. Meredith, A. *P0467R2 Iterator Concerns for Parallel Algorithms*. 2017-03-02. wg21.link/p0467.
14. Niebler, E; Parent, S; Sutton, A. *N4128 Ranges for the Standard Library, Revision 1*. 2014-10-10. wg21.link/n4128.
15. Niebler, E. *P0789R1 Range Adaptors and Utilities*. 2017-11-17. wg21.link/p0789r1.
16. Reyes et al. *P0687R0: Data Movement in C++*. 2017-05-30. wg21.link/p0687r0.
17. Reyes et al. *P0443R4 A Unified Executors Proposal for C++*. 2017-11-22. wg21.link/p0443r4.