Parallelizing the Standard Algorithms Library | N3408=12-0098

Jared Hoberock Michael Garland Olivier Giroux Vinod Grover
Ujval Kapasi Jaydeep Marathe
{jhoberock, mgarland, ogiroux, vgrover, ukapasi, jmarathe}@nvidia.com

2012-09-21

This document proposes to augment the C++ Standard Algorithms Library to allow parallel implementations based on our experience with the working prototype of such a library, Thrust.

1 Motivation for Parallel Algorithms

Modern processor architectures are inexorably embracing both increasing levels of parallelism and heterogeneous processing elements. Both trends reflect the need to deliver both greater performance and better energy efficiency. However, both trends also present numerous challenges to programmers for which they are often unprepared. These architectural trends appear durable rather than ephemeral. To remain relevant, C++ must evolve to better equip programmers in their struggle to harness the power of modern parallel architectures to solve their computational problems.

C++ programmers equipped with only the lowest-level support for concurrency such as `std::thread` and `std::atomic` face a handful of difficult challenges. While ad hoc approaches to parallelism occasionally suffice to exploit the relatively modest parallelism of multicore processors, massively multithreaded processors force us to craft more durable solutions to the challenges facing C++ programmers. The most frequent of these challenges include: expressing sufficient fine-grained parallelism to efficiently utilize the machine, determining an appropriate degree of physical parallelism to match the underlying architecture, and avoiding the many pitfals of shared state in the presence of concurrency.

Achieving good performance across a broad spectrum of machines in the face of such challenges requires a higher level of abstraction than is found in C++11. In this document, we propose an approach to parallelism that insulates the C++ programmer from the low-level, hardware-specific details that often make parallel programming difficult while still delivering high performance in practice. This approach is embodied in the Thrust C++ template library, which provides a collection of parallel algorithms modeled on and compatible with the C++ Standard Algorithms Library. Parallel programs written in this style implicitly express parallelism via operations on data collections, place responsibility for determining the appropriate degree of parallelism in the hands of the library implementation, and almost entirely avoid the hazards of shared state.

C++ provides standard support for concurrency in the form of functionality like `std::async` and `std::mutex`. These primitives, while useful and necessary for building multithreaded task scheduling systems, remain out of reach to programmers unwilling or unable to target such a low-level abstraction. Because it is widely appreciated that concurrency is a domain notoriously difficult to master, we believe that it is unlikely that a mass audience of programmers will become productive at applying low-level concurrency primitives to target their codes portably to a broad class of parallel architectures. Instead, standard and broadly-accessible functionality should be constructed to bridge the gap between the abundant parallelism implicit in many applications and the concurrent resources of the target architecture.

2 Summary of Proposed Functionality

Absent from C++ is standard functionality for the programmer to implicitly encode the high-level parallel structure of their program such that a parallel implementation may accelerate it on her behalf. However, high-level algorithms, often with serial semantics, do exist in the C++ Standard Algorithms Library.

2.1 A New Suite of Standard Parallel Algorithms

To complement C++'s existing support for concurrency and sequential algorithms, we propose to augment the existing C++ Standard Algorithms Library with additional algorithms and semantics which may be accelerated by parallel architectures informed by our past experience with the suite of algorithms present in the Thrust library:

- Searching
  - find
  - find_if
  - find_if_not
  - mismatch
  - partition_point

  - Binary Search
    - lower_bound
    - upper_bound
    - binary_search
    - equal_range

  - Vectorized Binary Search
    * lower_bound
    * upper_bound
    * binary_search

- Merging
  - merge
  * merge_by_key

- Sorting
  - sort
  * sort_by_key
  - stable_sort
  * stable_sort_by_key

- Reordering
  - Partitioning
    - partition
    - partition_copy
    - stable_partition
    * stable_partition_copy

  - Stream Compaction
    - copy_if
    - remove
    - remove_copy
    - remove_if
    - remove_copy_if
    - unique
    - unique_copy
    * unique_by_key
    * unique_by_key_copy

- Reductions
  * reduce
  * reduce_by_key

  - Counting
    - count
    - count_if

  - Comparisons
    - equal

  - Extrema
    - min_element
    - max_element
    - minmax_element

  - Transformed Reductions
    - inner_product
    * transform_reduce

  - Logical
    - all_of
    - any_of
    - none_of

  - Predicates
    - is_partitioned
    - is_sorted
    - is_sorted_until

- Copying
  - copy
  - copy_n
  - move
  - swap_ranges
  - uninitialized_copy
  - uninitialized_copy_n

  - Gathering
    * gather
    * gather_if

  - Scattering
    * scatter
    * scatter_if

- Prefix Sums
  * inclusive_scan
  * exclusive_scan

  - Segmented Prefix Sums
    * inclusive_scan_by_key
    * exclusive_scan_by_key

  - Transformed Prefix Sums
    * transform_inclusive_scan
    * transform_exclusive_scan

  - Set Operations on Sorted Lists
    - set_difference
    * set_difference_by_key
    - set_intersection
    * set_intersection_by_key
    - set_symmetric_difference
    * set_symmetric_difference_by_key
    - set_union
    * set_union_by_key

- Transformations
  - adjacent_difference
  - generate
  - generate_n
  * sequence
  - transform
  * transform_if

- Filling
  - fill
  - fill_n
  - uninitialized_fill
  - uninitialized_fill_n

  - Modifying
    - for_each
    * for_each_n

  - Replacing
    - replace
    - replace_if
    - replace_copy
    - replace_copy_if

Proposed new algorithms are denoted with a *.

A precise specification of the Thrust algorithms API is available online.

2.2 Allowing the Programmer to Request Parallelization

In order to allow the programmer to request a parallel algorithm launch, we propose to introduce for each algorithm in the previous table overloads of the form:

```
namespace std
{
  template<typename InputIterator1, typename InputIterator2, typename Function>
    InputIterator2
      transform(to-be-determined-launch-policy-type policy,
                InputIterator1 first, InputIterator1 last,
                InputIterator2 result,
                Function op);
}
```

As well as the entities

```
namespace std
{
  to-be-determined-type seq;

  to-be-determined-type par;
}
```

To illustrate how a programmer would control the parallelization of an algorithm launch, consider this SAXPY example code:

```
#include <vector>
#include <algorithm>

int main()
{
  auto n = 1 << 20;
  std::vector<float> x(n, 7), y(n, 13);
  auto a = 1.f;

  // parallelize SAXPY
  std::transform(std::par, x.begin(), x.end(), y.begin(),
    [=](float xi, float yi){
      return a * xi + yi;
  });

  // sequentialize SAXPY
  std::transform(std::seq, x.begin(), x.end(), y.begin(),
    [=](float xi, float yi){
      return a * xi + yi;
  });

  // leave the decision of parallelization up to the implementation
  std::transform(x.begin(), x.end(), y.begin(),
    [=](float xi, float yi){
      return a * xi + yi;
  });

  return 0;
}
```

To further customize the details of an algorithm launch, we encourage library vendors to provide implementation-specific launch policies. For example, a Threading Building Blocks-derived implementation

may wish to allow the programmer to request a "grain size" in order to better adapt the problem to the system's computational resources:

```
// parallelize SAXPY with a 10k grain size:
std::transform(tbb_launch(10000), x.begin(), x.end(), y.begin(),
  [=](float xi, float yi){
    return a * xi + yi;
});
```

This proposal is founded on the substantial experience we accumulated designing and implementing the Thrust parallel algorithms library using a variety of threading models which allow portable implementations for multicore CPUs and GPUs. While we expect that a standard parallel algorithms library would be different in a number of respects from Thrust's current incarnation, we believe that our experience with Thrust as a working model will be valuable in informing a standardization process.

Readers interested in the rationale for our proposal's design may refer to the remainder of this document, where we justify the standardization of our proposal and consider a number of rejected alternative designs.

3 Why Standardize a Library of Parallel Algorithms?

Before discussing the rationale of our proposal's design, it is useful to understand what advantages a library-based parallel algorithms standardization strategy offers compared to alternative approaches to parallelism. Indeed, the proliferation of parallel architectures brings with it a cornucopia of parallel programming systems. Many of these are either directly accessible from C++, or from a non-standard dialect. At first glance, some of these might appear to be interesting candidates for standardization.

3.1 Libraries Reduce Risk

One compelling advantage of a library-based approach to parallelism is that it would involve inherently less risk than a design which proposed to alter the core language of C++. It is often difficult to predict precisely how novel language constructs will interact with seemingly unrelated parts of C++. By contrast, our proposal to parallelize an existing suite of well-understood algorithms carries with it substantially less uncertainty. Moreover, the uncertainty which remains may be resolved through investigation which does not necessitate the enlistment of scarce experimental compiler resources. A library-only solution is likely to be delivered promptly for the same reason.

3.2 Uncertain Alternatives

Another argument in favor of standardizing a library of parallel algorithms is that it remains unclear what a competing approach, language-based or otherwise, could be. After several years of post-parallelism we believe there remains no candidate which is at once portable, featureful, and high performance. We discuss several of them here in turn.

Intel Threading Building Blocks (TBB) and libraries like it offer robust schedulers for task level parallelism. For example, `tbb::parallel_for` repeatedly subdivides a range of elements into partitions small enough to be consumed by a sequential task, which is provided by the programmer. These tasks may wait in a queue until a serial processor is ready to consume them. Any further parallelization within a task, for example, via vectorization, is left to the discretion of the programmer. TBB works well for architectures with a modest number of logical processing units. However, such a scheme does not map well to architectures which require fine grained data parallel work. For example, it is unclear how such a model could be accelerated by present day GPUs.

In most modern processor architectures, an increasing amount of computational power is delivered via vector instruction sets. These facilities remain outside the reach of standard C++ programmers, but there exist many non-standard ways to access them, varying from native assembly, intrinsics, loop pragmas, and even domain-specific languages. Such solutions work well for fine-grained parallelization of simple arithmetic-heavy loops. However, none provide a general purpose solution which targets the entire range of C++ constructs, owing primarily to the limited features of the vector ISAs themselves. For example, function calls from a vectorized loop are often either impossible, or execute serially in a decellerated mode. Notably, systems like NVIDIA CUDA and Intel SPMD Program Compiler (ISPC) address this issue by requiring special annotation and compilation of "elementary" functions which are invokable from implicitly vectorized contexts. Such functions often come with substantial restrictions. In time, the eventual maturation of processor architectures may allow such functionality to be exposed in a standard, portable fashion.

Owing to GPUs' more featureful vector instruction sets, language extensions such as CUDA allow the programmer to express herself using a growing number of C++ language features. CUDA requires the programmer to instruct the compiler, through the `__device__` annotation, which functions may be invoked from a parallel context executing on the GPU. Because they execute on a GPU, these functions come with additional features and restrictions. For example, `__device__` functions offer fast synchronization with other threads executing on the same logical core. On the other hand, some important C++ language features, such as exceptions, are currently entirely absent from the capabilities of these functions. `__device__` functions

cannot invoke normal C++ functions, which complicates integration with large, existing codebases. While CUDA is useful for targeting GPUs, it remains unclear whether current multicore CPUs can be a good target.

In general, we believe that a truly portable, featureful, and efficient low-level programming model remains an open question. Standardizing an existing one would be premature as it would necessarily exclude some important subset of parallel architectures. However, as parallel programming models and the parallel architectures they target evolve, we believe eventually a candidate for standardization will emerge. In the near term, we are confident that a suite of high-level, portable, parallel algorithms based on the working model of Thrust effectively services a large number of important use cases.

3.3 Vendor Neutrality

Another argument in favor of standardizing a library of parallel algorithms is that such a strategy does not favor a particular hardware vendor, or even a particular threading model. While none of the previously discussed low-level parallel programming models are good candidates for standardization, any one of them, when viewed from the perspective of a single vendor, would be a fine candidate for implementing a library of parallel algorithms. While the parallel algorithms library would present a standard interface to the programmer, the library implementor is free to employ any set of non-standard features provided by the compiler and target architecture. For example, GNU's implementation might employ OpenMP, while Intel's might employ TBB, while NVIDIA's might employ CUDA. These implementations could be tuned by experts with specialized archtectural knowledge to make informed decisions about how best to decompose a program to utilize nodes, sockets, cores, hardware threads, vector lanes, and architecture-specific caches. Such implementations could be expressed using any set of non-standard programming constructs the vendor's compiler provided. Because of the library's standard interface separating the programmer and implementation, she need not be aware of these non-standard details.

3.4 Future Proof

A final argument in favor of standardizing a library of parallel algorithms is that the choice of a suite of parallel algorithms with correctly-designed semantics is largely orthogonal to the choice of underlying concurrency primitives which may comprise the implementation. The semantics of the existing standard algorithms have allowed us to implement several conformant library backends targeting very different non-standard underlying threading models. Should ISO C++ choose to adopt our proposal, it would not preclude the standardization of novel lower-level concurrency primitives at a future point.

4 Thrust as a Proof of Concept

The Thrust project provides an existence proof that a library of efficient and portable parallel algorithms is imminently realizable. Thrust is a mature open source C++ parallel algorithms library used in production in a diverse set of application domains, and is also bundled with the NVIDIA CUDA Toolkit as a product officially supported by NVIDIA. Thrust's suite of parallel algorithms currently targets GPUs and multicore CPUs by leveraging NVIDIA CUDA, OpenMP, and Intel TBB in its algorithm implementations. In addition to these three parallel "backends", Thrust also provides a sequential standard C++ implementation for each of the algorithms it provides. Thrust's architecture is user-extensible, providing a skilled programmer the means to customize an existing backend or build a completely novel backend based on a new parallel substrate easily.

4.1 Example Codes

Thrust programs should look familiar to any C++ programmer familiar with the standard algorithms library. The following code sample generates random numbers serially and transfers them to a parallel device where

they are sorted in parallel:

```
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <vector>
#include <algorithm>
#include <cstdlib>

int main()
{
  // generate 32M random numbers serially
  std::vector<int> vec(32 << 20);
  std::generate(vec.begin(), vec.end(), std::rand);

  // transfer data to the parallel device
  thrust::device_vector<int> d_vec = h_vec;

  // sort data in parallel
  thrust::sort(d_vec.begin(), d_vec.end());

  // copy data back to the CPU
  thrust::copy(d_vec.begin(), d_vec.end(), vec.begin());

  return 0;
}
```

This code sample computes a histogram:

```
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <thrust/reduce_by_key.h>
#include <thrust/inner_product.h>
#include <thrust/functional.h>
#include <thrust/iterator/constant_iterator.h>
#include <vector>
#include <algorithm>
#include <cstdlib>

int main()
{
  // generate 32M random numbers serially
  std::vector<int> data(32 << 20);
  std::generate(data.begin(), data.end(), std::rand);

  // transfer data to the parallel device
  thrust::device_vector<int> d_data = data;

  // sort data to bring equal elements together
  thrust::sort(d_data.begin(), d_data.end());

  // to count the number of histogram bins, count the
  // number of unique values in the input
  int num_bins =
```

```
    thrust::inner_product(data.begin(), data.end() - 1,
                          data.begin() + 1, data.end(),
                          1,
                          thrust::plus<int>(),
                          thrust::not_equal_to<int>());

  // allocate a histogram
  thrust::device_vector<int> histogram_values(num_bins);
  thrust::device_vector<int> histogram_counts(num_bins);

  // reduce each run of equivalent values into its count
  thrust::reduce_by_key(data.begin(), data.end(),
                        thrust::constant_iterator<int>(1),
                        histogram_values.begin(),
                        histogram_counts.begin());

  // histogram_values contains each unique element in the data
  // histogram_counts contains the count of each unique element in the data

  return 0;
}
```

As these code examples demonstrate, the Thrust library provides novel algorithms, iterators, containers, and other functionality for expressing parallel programs in an idiomatic modern C++ style. However, this document focuses its attention on Thrust's suite of parallel algorithms.

4.2 Legacy Interoperability

We believe that the constraints under which Thrust was created contributed to a design that is a good candidate for standardization. When designing Thrust, our goal was an interface which would feel very familiar to existing C++ programmers, especially those conversant in the STL. Not only did this allow us to target a parallel algorithms library at a broad audience of existing C++ programmers, it allowed us to leverage the corpus of pre-existing STL documentation and educational resources which also largely apply to Thrust. The fine-grained similarity between Thrust and the existing C++ Standard Algorithms Library also allows Thrust to interoperate with legacy C++ codebases in a straightforward manner. In fact, for existing codebases which rely heavily on applying standard algorithms to arrays, Thrust can sometimes be a drop-in replacement.

4.3 Standard Conventions

Thrust diverges from established C++ idioms and conventions only where absolutely necessary to deliver parallel functionality. For example, employing the fully-general `Iterator` concept to specify function parameters may not be the most concise way to pose the invocation of a parallel algorithm, because parallel algorithms are logically restricted to random access ranges. It makes no sense, for example, to attempt to parallelize an algorithm such as `transform` on a linked list, whose traversal is implicitly sequential. Restricting the function interface to the standard `RandomAccessIterator` concept, or perhaps a hypothetical `RandomAccessRange` concept, might seem like a prudent design decision.

Nevertheless, when designing Thrust's algorithm interface, we adopted the standard convention of requiring iterators as function parameters. This allows Thrust to opportunistically parallelize an algorithm when appropriate to the category of iterator but deliver a correct result when the iterator's traversal would preclude parallelism. For example, `thrust::for_each` is equally applicable to both `std::list` and `std::vector`, but may be parallelized only for the latter. Choosing to require iterators as parameters also avoided the problem

of having to invent an unfamiliar and non-standard `Range` concept. In this way, Thrust compromises on interface terseness for the sake of generality and familiarity.

4.4 Backend Agnosticism

We also wished to provide an interface which would allow the programmer to be as agnostic as possible regarding the particular parallel backend being targeted. This constraint ensures portability between different backends such as CUDA and OpenMP but requires the interface to be devoid of any compiler-specific or backend-specific extra-lingual construct. For example, expert CUDA users sometimes wish to specify a particular CUDA "block size" that an algorithm like `thrust::for_each` should employ when mapping the algorithm to the GPU's multiprocessors. However, such a request would be meaningless to Thrust's TBB backend. Likewise, while a "grain size" hint might be meaningful to TBB, neither CUDA nor OpenMP have any such notion. In general, we have observed that most existing widespread threading systems require the tuning of platform-specific parameters.

The exception to this rule is Thrust's requirement that user-defined functors be decorated with CUDA's `__host__ __device__` annotation, which is required by the NVIDIA compiler to generate code for both the CPU and GPU. Since this is the only way to deliver the functionality for the CUDA backend, Thrust makes this concession. However, when targeting OpenMP or TBB on other compilers, such annotation is unnecessary. As both the NVIDIA compiler and GPU architecture mature, we hope to be able to relax this requirement so that user-authored code which interoperates with Thrust may be truly backend-agnostic and portable.

Where Thrust diverges from the specifications of the C++ standard algorithms is in semantics and naming. We explain our rationale in the next section.

5 Algorithms with Parallel Semantics

Though Thrust's algorithms are generally derived from those found in the standard library, it's important to understand where they differ and why. Where Thrust diverges from the standard algorithms, it does so in a way to allow a parallel implementation.

5.1 `std::for_each` versus `thrust::for_each` and `thrust::for_each_n`

`std::for_each` assumes the function object it receives contains state which will be mutated inside a serial loop. It returns a copy of this state as a result. `std::for_each_n` does not exist.

By contrast, `thrust::for_each` and `thrust::for_each_n` instantiate many copies of their function object parameter in parallel. In such a setting, shared mutable state within the function object is a performance hazard at best. At worst, it is impossible for some parallel substrates to achieve. Instead, `thrust::for_each` and `thrust::for_each_n` exist exclusively for the sake of the side effects they may induce on the elements of their input ranges.

For consistency with other higher-level algorithms, `thrust::for_each` and `thrust::for_each_n` return the end of their input range. This simplifies the implementation of algorithms such as `thrust::generate_n` which can be lowered onto `thrust::for_each_n` in a straightforward manner.

5.2 `std::accumulate` versus `thrust::reduce`

The standard algorithm `std::accumulate` computes, in serial, the generalized notion of a sum.

`std::accumulate` takes an input range of values to sum, as well as an optional user-defined binary function `binary_op`. `std::accumulate` is defined to operate by initializing an accumulator variable `acc` and repeatedly

executing in order the read-modify-write: `acc = binary_op(acc, *iter)`. These semantics are explicitly serial and allow the user-specified `binary_op` to be mathematically non-associative and non-commutative.

Thrust has no `accumulate` algorithm. Instead, it introduces the analogous `thrust::reduce`, which requires stricter semantics from its user-specified sum operator to allow a parallel implementation. Specifically, `thrust::reduce` requires mathematical associativity and commutativity of its user-specified sum operator. This allows the algorithm implementor discretion to parallelize the sum.

We chose the name `reduce` for this algorithm because we believe that most existing parallel programmers are familiar with the idea of a parallel reduction. Other names for this algorithm exist, e.g., `fold`. However, we did not select a name derived from `fold` because other languages tend to impose a non-associative directionality to the operation. [cf. Haskell's `foldl` & `foldr`, Scala's `foldLeft` & `foldRight`]

5.3 `std::inner_product` versus `thrust::inner_product`

`std::inner_product` is defined to compute a generalized inner product of its input ranges in a serial accumulation fashion similarly to `std::accumulate`. Like `thrust::reduce`, `thrust::inner_product` computes a reduction over its input ranges and requires both associativity and commutativity of its user-defined binary sum and multiplication operators.

Thrust retains the name `inner_product` for this algorithm even though its non-standard semantics allow for parallelism. Our rationale was that unlike the name `accumulate`, `inner_product` does not connote a serial accumulation implementation. Additionally, a mathematical inner product is precisely the operation that this algorithm computes.

5.4 `std::partial_sum` versus `thrust::exclusive_scan` and `thrust::inclusive_scan`

The standard algorithm `std::partial_sum` computes, for each element of an input range, the generalized sum over the prefix of the range ending at that element. It is explicitly defined to do so in serial. To update the sum, `std::partial_sum` employs an in order accumulation method similar to `std::accumulate`. Again, this accumulation semantic allows for non-associativity and non-commutativity.

Thrust has no `partial_sum` algorithm. Instead, it introduces two analogous algorithms, `thrust::exclusive_scan` and `thrust::inclusive_scan`. Like `thrust::reduce`, these algorithms require associativity and commutativity of their user-defined binary sum operators to allow for parallelization.

Thrust has two algorithms implementing this operation to capture two different use cases. For each element, `thrust::inclusive_scan` includes the element in the corresponding sum in the output range. By contrast, `thrust::exclusive_scan` excludes it.

To capture a common use case, Thrust introduces an overload for `thrust::exclusive_scan` which takes an initial seed value for the sum. This both handles the case of an empty input range and allows the user to specify a "zero element". No such overload exists (nor has reason to exist) for `thrust::inclusive_scan` and `std::accumulate`.

Like `std::partial_sum`, the beginning of the output ranges of `thrust::inclusive_scan` and `thrust::exclusive_scan` may coincide with the beginning of their input ranges, but may not overlap otherwise.

We selected the names `exclusive_scan` and `inclusive_scan` because "scan" is an operation well-known to parallel programmers. A name derived from "prefix-sum" also might have been appropriate. However, we were concerned that longer names derived from "prefix-sum" would have become verbose. `exclusive_scan` and `inclusive_scan` also follow the convention that most algorithm names are verbs.

6 Algorithm Variants

As well as providing parallel analogues to implicitly serial algorithms from the standard library, Thrust also provides functionality that while similar, has no standard counterpart. These algorithms provide an interface that accomodate data organization and use cases which are important in parallel contexts.

6.1 Algorithms which Decouple Keys from Values

Many standard algorithms take a user-defined `Predicate` (e.g. `std::copy_if`) or `Compare` (e.g., `std::sort`) function object which allows the algorithm implementation to reorder its range of input data based on criteria provided by the function object. To customize the ordering, the user need only to customize the operator, or customize the type of input data, or both. A very common use case is to reorder a range of input data based on a key associated with each input value. Using standard algorithms, one reasonable realization would be to include the key inside the type of each value to be reordered:

```
struct my_record
{
  float value; int key;
};
```

To communicate the existence of the key to a standard sorting algorithm, the user would also provide an ordering function:

```
bool compare_records(const my_record &x, const my_record &y)
{
  return x.key < y.key;
}
```

Sorting a `std::vector` of `my_record` by key is now straightforward:

```
std::vector<my_record> records;
...
std::sort(records.begin(), records.end(), compare_records);
```

This data organization is often called an "array of structs" (AOS). For many important use cases, the AOS data organization is a performance hazard because it wastes memory bandwidth when only a portion of the struct is required for a particular computation. Though AOS is particularly troublesome on parallel architectures, whose vector nature extends to the memory system, AOS is often a hazard even on sequential processors. Instead, the data organization "struct of arrays" (SOA) often achieves significantly better performance on a wide range of architectures. Furthermore, accessing data organized in a SOA format is rarely significantly worse than an AOS layout.

With SOA, the idea is to decouple data values from their keys, or more generally, to decouple all of a struct's members into separate arrays. For the above example, instead of a single `std::vector<my_record>`, we would separate values from their keys into both a `std::vector<float>` and `std::vector<int>`:

```
std::vector<float> values;
std::vector<int> keys;
```

The problem with this organization is that it is awkward or impossible to use `std::sort` to sort the ranges using elements from the `keys` array as the sorting key.

To accomodate this frequently-occuring use case, Thrust introduces the non-standard algorithm `thrust::sort_by_key`:

```
std::vector<float> values;
std::vector<int> keys;
...
thrust::sort_by_key(keys.begin(), keys.end(), values.begin());
```

This SOA sort is likely to achieve higher performance than the previous AOS sort simply due to the better bandwidth utilization enabled by the decoupled data organization. However, we can go further. Because `sort_by_key` has access to the type of the actual sorting keys, in many common cases Thrust will exploit the type's binary representation to dispatch a high performance radix sorting algorithm.

Besides performance concerns, the SOA data organization expected by `thrust::sort_by_key` offers other desirable characteristics. For example, the user is not required to introduce a novel comparison function simply to inspect each record's key. Neither is the user required to intrude upon the definition of `my_record` to accomodate a sort. Of course, should the user wish to customize the comparison function used in a key-value sort, Thrust provides a overload analogous to `std::sort`:

```
std::vector<float> values;
std::vector<int> keys;
...
bool compare_most_significant_byte(int x, int y)
{
  unsigned int shift = sizeof(int) - 1;
  return (x >> shift) < (y >> shift);
}
...
thrust::sort_by_key(keys.begin(), keys.end(), values.begin(), compare_most_significant_byte);
```

The standard algorithms library would benefit from additional algorithm names and overloads of existing names which would accomodate an SOA data organization. For standard algorithms which take a user-provided predicate function, Thrust provides overloads which decouple the data of the input range from a "stencil" range to which the predicate is applied. For example, Thrust complements the standard algorithm `std::copy_if` with an additional overload:

```
template<typename InputIterator,
         typename OutputIterator,
         typename Predicate>
  OutputIterator
    copy_if(InputIterator first, InputIterator last,
            OutputIterator result, Predicate pred);

template<typename InputIterator1,
         typename InputIterator2,
         typename OutputIterator,
         typename Predicate>
  OutputIterator
    copy_if(InputIterator1 first, InputIterator1 last,
            InputIterator2 stencil,
            OutputIterator result, Predicate pred);
```

Similarly, Thrust complements standard algorithms which receive sorted ranges with additional key-value algorithms which are suffixed with `_by_key`. For example, Thrust's `stable_sort` family of algorithms:

```
template<typename RandomAccessIterator>
```

```
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template<typename RandomAccessIterator, typename Compare>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);

template<typename RandomAccessIterator1, typename RandomAccessIterator2>
  void stable_sort_by_key(RandomAccessIterator1 keys_first, RandomAccessIterator1 keys_last,
                          RandomAccessIterator2 values_first);

template<typename RandomAccessIterator1, typename RandomAccessIterator2, typename Compare>
  void stable_sort_by_key(RandomAccessIterator1 keys_first, RandomAccessIterator1 keys_last,
                          RandomAccessIterator2 values_first,
                          Compare comp);
```

Not only does the `_by_key` suffix emphasize the different data organization, it distinguishes the overload of `stable_sort` with `comp` from the overload of `stable_sort_by_key` without `comp` which would otherwise be ambiguous.

Introducing a `stencil` overload of a predicate-consuming algorithm or a `_by_key` version of a sorted range-consuming algorithm does not always add value. For example, the hypothetical algorithm `is_sorted_by_key` need not exist because `is_sorted` is sufficient for either a key-value or a keys-only organization of data; receiving an additional range of values would only be superfluous. Similarly, a hypothetical overload of `is_partitioned` which received an additional `stencil` input range need not exist.

6.2 Algorithms Which Consume a Sequence of Logical Subsequences

Another commonly occurring parallel use case is the application of a common operation to a sequence of logical subsequences. Even when the average size of a subsequence is relatively small, in aggregate, these bulk operations can be a significant source of parallelism. Providing parallel programmers with standard algorithms which capture this use case is therefore extremely valuable.

Thrust repurposes the key-value data organization to describe the layout of such logically-nested sequences. For example, Thrust provides the algorithm `thrust::reduce_by_key` which perform reductions, in parallel, on a logically segmented array. In `thrust::reduce_by_key`, for each segment a reduction is executed whose semantics are equivalent to those of `thrust::reduce`. The reduced value of each segment is stored to an output sequence, along with the key identifying the subsequence from whence it came.

To distinguish segments within the input array, `thrust::reduce_by_key` employs an optional user-provided `BinaryPredicate` to determine key equality. These semantics are derived from `std::unique`, the standard algorithm which "uniquifies" a sequence. In fact, `thrust::unique` is itself a special form of `thrust::reduce_by_key`, and is implemented with `thrust::reduce_by_key` via a straightforward lowering.

It is important to note that, like `std::unique`, two non-adjacent segments with equivalent keys are still considered distinct segments. In other words, `thrust::reduce_by_key` does not first sort its input, require already sorted input, or even assume that the type of its input keys have an ordering. Instead, `thrust::reduce_by_key` discovers discontinuities in the key sequence by applying an optional user-provided function object for key equality.

To justify the inclusion of nested primitives in the standard algorithms library, it is worth considering the alternative of relying on the composition of lower-level parallel primitives to achieve similar results. After all, orthogonality of library features is a worthwhile goal. For example, `thrust::reduce_by_key`'s semantics are equivalent to a nested application of `thrust::reduce` from `thrust::transform`:

```
vector<int>             input_keys;
```

```
vector<pair<int*,int*>> input_value_segments;
vector<int>             output_keys;
vector<int>             output_values;
...
transform(input_keys.begin(), input_keys.end(), value_segments.begin(),
  make_zip_iterator(make_tuple(output_keys.begin(), output_values.begin())),
  [=](int key, pair<int*,int*> val_seg){
    return make_tuple(key, reduce(val_seg.first, val_seg.second));
  }
});
```

This implementation is both unwieldy and assumes the existence of the non-standard `zip_iterator`. However, barring those concerns, the efficiency of such an approach is questionable for a few reasons. As the average segment length shrinks, we expect the the performance of this code to be dominated by the launch overhead of `reduce`. In addition to introducing additional parallelism with each call to `reduce`, many implementations will also allocate temporary memory to implement each reduction's partial sums. Both of these side effects can have a critical impact on performance. For machines which expect wide parallelism, utilization of parallel resources will be low.

`thrust::reduce_by_key`'s interface avoids this problem by allowing the programmer to express maximal parallelism in a single algorithm invocation rather than by relying on a hypothetical process which would be required to discover parallelism late from incomplete information. Encoding the subsequences in a single sequence of keys allows an implementation to both load balance and allocate temporary memory en masse rather than in a piecemeal fashion.

In addition to a key-value reduction, Thrust also provides the key-value prefix-sums `exclusive_scan_by_key` and `inclusive_scan_by_key`.

6.3 Indirect Copy Algorithms

Because of their frequent need in parallel applications, Thrust provides algorithms for performing indirect copies from or to random access sequences. For example, `thrust::gather` takes a sequence of indices, a random access source range, and assigns to an output range elements from the source indexed by elements taken from the sequence of indices. `thrust::scatter` works in the opposite direction; from sequences of elements and indices, `thrust::scatter` assigns a source element to the position in a random access output sequence specified by its associated index. In this way, `thrust::gather` and `thrust::scatter` are the inverses of each other.

Thrust also provides the variants `thrust::gather_if` and `thrust::scatter_if` which use stencil sequences to conditionally control whether or not an element is gathered or scattered, respectively.

6.4 Fusion Algorithms

In many parallel settings, main memory is separated from processing cores by a gulf of latency. Unlike in serial settings, caches are often a poor remedy because many parallel algorithms offer little reuse or have memory access patterns which cannot easily be captured by a cache. For example, parallel maps (e.g. `thrust::transform`) and reductions (e.g. `thrust::reduce`) have no reuse as they operate by streaming through a data set and accessing each element of the stream only once. Because memory bandwidth is a scarce resource, a good parallel algorithms library should offer the programmer opportunities for bandwidth conservation.

One such conservation technique is "kernel fusion". With kernel fusion, the idea is to "fuse" together two or more dependent parallel kernels, for example a parallel map followed by a reduction, into a single operation which loads from and stores to main memory exactly once. Without fusion, a parallel map consuming a

length N array costs at least N loads and N stores, with a bandwidth cost of 2N. A parallel reduction of the map's product would cost another N loads. We assume the cost of the reduction's scalar store is negligible. The bandwidth cost of the aggregate operation is then 3N.

A fused operation would combine the parallel map's element-wise function application with the reduction's load. As the reduction loads elements from the input array it immediately transforms them before applying the reduction's binary operation. Because this fused operation occurs on the granularity of array elements, it can be implemented within the processor's registers without performing a round trip to main memory. In this way, the bandwidth cost of the fused operation is only N loads. Because many parallel maps and reductions are bound by the speed of memory bandwidth, we would expect this fused algorithm to perform three times faster than the alternative.

Because this algorithmic pattern is so common, and the performance advantage so compelling, Thrust provides the algorithm `thrust::transform_reduce`. `thrust::transform_reduce`'s interface and semantics are similar to `thrust::reduce` but adds a `UnaryFunction` parameter which is applied to each element of the input sequence before application of its `BinaryFunction`. Similarly, the algorithms `thrust::transform_exclusive_scan` and `thrust::transform_inclusive_scan` fuse the application of a unary function to their input sequences before computing a prefix sum.

We chose not to include a hypothetical algorithm `thrust::transform_reduce_by_key` which would fuse a parallel map with a key-value reduction owing to uncertainties about semantic specification and concerns about an unwiedly interface. Other Thrust algorithms also lack fused `transform_` variants. However, conserving memory bandwidth is still an important consideration for such algorithms.

Rather than proliferate algorithmic entry points with subtly different semantics, we opted for an orthogonal design. Thrust programmers may achieve kernel fusion with any algorithm by drawing from a library of "fancy iterators" derived from the Boost Iterator Library. In fact, `thrust::transform_reduce` is implemented via a straightforward lowering to `thrust::reduce` combined with `thrust::transform_iterator`. While this document does not propose to standardize functionality from this iterator library, we believe it is worth future consideration as fancy iterators add tremendous value to Thrust.

7 Immediately Parallelizable Standard Algorithms

It's worth noting that the semantics of most standard algorithms allow for immediate parallelization. Thrust provides parallel implementations of the following algorithms with standard semantics:

```
Modifying Sequence Operations  Non-Modifying Sequence Operations  Partitioning Operations
 * copy                          * all_of                          * is_partitioned
 * copy_if                       * any_of                          * partition
 * copy_n                        * none_of                         * partition_copy
 * adjacent_difference           * count                           * partition_point
 * fill                          * count_if                        * stable_partition
 * fill_n                        * mismatch                       Sorting
 * transform                     * equal                           * is_sorted
 * generate                      * find                            * is_sorted_until
 * generate_n                    * find_if                         * sort
 * remove                        * find_if_not                     * stable_sort
 * remove_if                    Set Operations on Sorted Ranges   Binary Search Operations
 * remove_copy                   * merge                          on Sorted Ranges
 * remove_copy_if                * set_difference                  * lower_bound
 * replace                       * set_intersection                * upper_bound
 * replace_if                    * set_symmetric_difference        * binary_search
 * replace_copy                  * set_union                       * equal_range
 * replace_copy_if              Minimum/Maximum Operations
```

```
 * swap_ranges                    * max_element
 * reverse                        * min_element
 * reverse_copy                   * minmax_element
 * unique                        Numeric Operations
 * unique_copy                    * adjacent_difference
```

While the parallelization of this body of algorithms may seem daunting, in practice this large set of algorithms may be generically lowered onto a small set of fundamental primitives:

`for_each`	`merge`	`reduce`
`for_each_n`	`set_union`	`reduce_by_key`
`exclusive_scan`	`set_difference`	`stable_sort`
`inclusive_scan`	`set_intersection`	`stable_sort_by_key`
	`set_symmetric_difference`	

Table 1: Thrust's Fundamental Primitives

Performance optimization of any algorithm is of course still possible through a specialized implementation.

8 Remaining Work

While we are confident that a robust standard parallel algorithms library modeled on Thrust is deliverable within a short time frame, there remains work necessary to harmonize it with the existing C++ standard library. The problems presented in this section either do not apply to Thrust per se or have not yet been addressed in practice. While we consider these issues to be remaining work, it is useful to describe the qualities we believe good solutions to these problems should have.

8.1 Presentation

From the programmer's perspective, possibly the most interesting question is how a new standard parallel algorithms library would be accessed. Because the future scalability of C++ codes hinges on how well programmers will take advantage of parallel computational resources, we believe that standard parallel algorithms should be as accessible as possible. For example, we believe a solution which would disincentivise the application of a parallelized version of `for_each` in favor of a serial `for` loop ought to be treated with extreme skepticism. To this end, a good solution to the problem of presentation must be simple and concise. In this section we consider the advantages and disadvantages of several approaches to presentation.

8.1.1 Implicit Versus Explicit Parallelism

Before investigating specific solutions to the problem of exposing parallel functionality, it's worth noting that any approach will offer the programmer some combination of either implicit or explicit parallelism. Explicit approaches to parallelism place the responsibility of parallelization firmly in the hands of the programmer: to achieve parallelization, the programmer must actively build it into her program. An implicit approach would give the library more of the responsibility of choosing when to parallelize. On the one hand, an explicit approach follows the C++ convention of giving the programmer only what she asked for. On the other hand, in many cases requiring the programmer to actively opt in to parallelism will be premature pessimisation when the programmer is passive. Ultimately, we believe a standard solution will have a balance of implicit and explicit parallelism as both use cases are vital.

8.1.2 Novel Algorithm Overloads

Informed by our experience with Thrust, we believe that the best way to expose a standard library of parallel algorithms would be by introducing novel overloads of existing algorithm names. Inspired by the form of `std::async`, the first parameter of these overloads would act as a policy to allow the programmer to indicate her parallelization intent. Revisiting the SAXPY example:

```
std::vector<float> x, y, z;
float a;
...
// parallelize SAXPY
std::transform(std::par, x.begin(), x.end(), y.begin(), z.begin(),
  [=](float x, float y){
    return a * x + y;
});

// serialize SAXPY
std::transform(std::seq, x.begin(), x.end(), y.begin(), z.begin(),
  [=](float x, float y){
    return a * x + y;
});
```

The algorithm's first argument, here passed by a terse, standard name, indicates whether or not the SAXPY should be parallelized. In the case of `std::transform`, omitting the argument would leave the parallelization decision to the implementation. Algorithms with explicitly sequential semantics, for example, `std::accumulate`, need not have these overloads. Different vendors would be encouraged to introduce implementation-defined policies or "hints" to customize the parallelization based on the facilities available to the implementation. In Thrust, we generalize this binary decision of sequential/parallel algorithm launch to several different backends.

The downside of this approach is the inevitable increase in API complexity which may confuse newcomers. Additionally, it seems reasonable to repurpose the asynchronous launch policy functionality present in the existing `std::launch` type. However, the type of `std::launch` is integral. While a bitmask might be appropriate for customizing the launch policy of a single serial, asynchronous task, it seems inadequate for customizing the launch of a potentially complex parallel algorithm. In practice, because the performance of a parallel algorithm often depends critically on implementation-specific tunables, answering the question of what additional state the first parameter should carry is important. In particular, allowing the user to control the allocation of temporary memory by customizing this parameter is particularly vital.

8.1.3 A Novel Namespace

One straightforward solution to the problem of presentation would be to mirror the names of the standard algorithms in a hypothetical `std::parallel` namespace. Algorithms in this namespace would be understood to execute in a parallel fashion. To achieve parallelization, the programmer would actively invoke algorithms from this namespace. All algorithms in the `std` namespace would execute sequentially as usual. This solution is similar to a subset of the functionality offered by the GNU `libstdc++`'s "parallel mode".

While introducing a novel namespace in many ways might be the simplest solution, it is not clear that it is optimal. While a hypothetical `std::parallel` namespace would preserve the legacy serial execution of algorithms in `std`, it is not a very forward-looking solution. For many future applications, we believe that the common case will be increasingly parallel. In fact, some vendors might argue that the legacy, sequential implementation of algorithms ought to be quarantined in a hypothetical `std::sequential` namespace. Ultimately, both sequential and parallel algorithms will be vital to future codes, so sequestering either set of them to a namespace seems arbitrary.

Aside from the question of the common case of future codes, a novel namespace also fails the test of concision. If a programmer is expected to prefix a parallel algorithm with a namespace, all things being equal, we believe she will be less likely to prefer it to the terser invocation of a sequential algorithm, or even an ad hoc `for` loop. In general, `std` contains few nested namespaces, presumably due to this concision principle. The standard library should not proliferate namespaces unnecessarily.

Finally, it is unclear how well a unique namespace for parallel algorithms would interact with argument-dependent lookup. With a namespace solution, programmers might receive either unexpected serialization or parallelization from the unintended application of ADL.

8.1.4 A Novel Prefix

A weaker version of the namespace solution would be to introduce a novel prefix for parallel algorithm names. For example `parallel_`. This is the solution offered by some existing libraries. For example, TBB provides the algorithms `tbb::parallel_for` and `tbb::parallel_sort`. As suggested by the names, these algorithms are parallel analogues of the algorithms `std::for_each` and `std::sort`.

The problem with this approach is that the semantics of most standard algorithms are already implicitly parallel, as we observed earlier. Prefixing them with something like `parallel_` seems redundant. As an example, it is doubtful that there would be value in distinguishing between `std::count` and a hypothetical `std::parallel_count` at the granularity of function name when the implementation of both would likely be through a lowering to `reduce`. Moreover, other novel algorithms we have proposed for standardization, for example, `reduce` is already understood to be parallel. A prefix to distinguish its parallel nature would be superfluous and simply add verbosity. Finally, a distinct name makes the choice of parallelization a static one which must be made at the time of compilation. Enabling the programmer to make a dynamic decision seems like a worthwhile goal.

8.1.5 A Novel Iterator Category

Since the choice of selecting either the sequential or parallel implementation of a standard algorithm is very similar to traditional C++ algorithm dispatch, it makes sense to consider a solution which would dispatch a parallel algorithm implementation based on the type of iterator category tags. For example, the hypothetical tag `std::parallel_iterator_tag` could be derived from the existing `std::random_access_iterator_tag`. Algorithms invoked on iterators with this tag would then be parallelized using existing means of algorithm dispatch.

There are many problems with this approach. The first is that there is no standard way to rebind the type of an existing iterator's traversal category. In other words, it is unclear how one would parallelize algorithms on iterators from existing random access containers such as `std::vector` and `std::deque`. At best, it would require the introduction of new functionality which would be burdensome for the programmer. To illustrate, assume the existence of a function `parallel` which takes a random access iterator and returns an iterator which acts like the original but whose category is `std::random_access_iterator_tag`, allowing the parallelization of an algorithm.

Invoking a parallel form of `std::transform` to implement SAXPY with such functionality would be burdensome and repetitive:

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

Furthermore, suppose that the programmer omits the transformation of one or more of an algorithm's iterator arguments. Should the algorithm be parallelized? Serialized? Should this invocation result in a compile time error? Any answer seems arbitrary.

8.1.6 A Novel Parallel Vector Container

One way to avoid the verbosity of the above invocation of `std::transform` would be to introduce a special parallel vector container. For example, algorithms on iterators from `parallel_vector` would be understood to be parallelized. Such a solution would simplify the preceding example:

```
std::parallel_vector<float> x, y, z;
float a;
...
std::transform(x.begin(), x.end(), y.begin(), z.begin(),
  [=](float x, float y){
    return a * x + y;
});
```

This solution eliminates verbosity and repetition. However, it requires copying existing data to a special type of container simply to parallelize an algorithm on that data. This solution would be needlessly wasteful. Moreover, it does not answer the question of whether this special kind of iterator is interoperable with normal random access iterators. Because a good solution must interoperate seamlessly with all random access sequences, this points to a solution which does not confuse the decision of parallelization with the type of iterator parameter.

It's worth noting that Thrust provides a superficially similar solution. While Thrust introduces the novel vector container `thrust::device_vector`, we are not proposing similiar functionality for standardization in this document as it solves the orthogonal problem of accessing data which exists in a remote memory space.

In summary, we believe that adopting an approach which allows the customization of algorithm launch through overloads is the most general way to deliver parallel functionality to C++ programmers. Existing parallelizable algorithm behavior need not change, but can, at the discretion of the implementation. Moreover, this approach is general enough that it does not preclude the additional adoption of any previously discussed alternative. We are confident that all such solutions could coexist within a single standard library as they currently do in Thrust.

8.2 Asynchrony

Perhaps the most interesting remaining open question is how to specify the synchronization semantics of a standard library of parallel algorithms. An easy solution to this problem would be to insist that the parallel algorithms block with the calling thread of execution identically to their serial counterparts. However, this specification would defeat many of the performance advantages of parallelization in the first place.

8.2.1 Asynchrony in Thrust

Currently, Thrust's algorithm implementations are mostly synchronous, owing to the semantics of its current implementation via primitives such as `#pragma omp for` and `tbb::parallel_for`. However, Thrust's CUDA backend is opportunistically asynchronous. CUDA presents a disjoint view of memory resources which is split between main system memory addressable by the CPU and memory addressable by the GPU. Treating the memory natively addressable by the GPU as a remote address space from the point of view of threads of execution on the CPU allows a relaxed form of synchrony. In normal cases, the CPU cannot observe the results of a GPU computation without explicitly copying those results back into the CPU's memory space.

Because these implicit synchronization points are easily detectable by the CUDA runtime, Thrust's CUDA backend can provide a consistent view of memory without eager synchronization. When a CPU thread invokes a Thrust algorithm on iterators which point into the GPU's remote memory space, simple algorithms derived from `thrust::for_each` can often launch without blocking with the calling thread of execution. For such simple algorithms, the result returned to the caller is known a priori as it is just a copy of the `last` iterator parameter. Other more complex algorithms which require internal temporary allocations, e.g. `thrust::sort`, may block within the algorithm implementation due to the synchronization semantics of CUDA's memory allocation primitives. Additionally, many of Thrust's algorithms are reductions (e.g. `thrust::count`) or return an otherwise data dependent result (e.g. `thrust::copy_if`). Such semantics are naturally blocking. Nevertheless, we are interested to provide additional support for asynchrony for all of Thrust's algorithms across all backends. We believe that similar support should exist within a standard parallel algorithms library.

8.2.2 Standard Approaches to Asynchrony

We believe that a good solution to asynchrony should be accessible to all C++ programmers and should interoperate with existing support for asynchrony within the C++ standard library. The existing model presented by `std::future` and `std::async` provides a good starting point. Indeed, one way to achieve an asynchronous algorithm launch would be to invoke such a launch through `std::async`:

```
std::vector<int> vec;
...
auto sum = std::async([](std::vector<int>::iterator begin, std::vector<int>::iterator end){
  return std::reduce(std::par, begin, end);
}, vec.begin(), vec.end());

std::cout << "sum is " << sum.get() << std::endl;
```

Such a solution achieves the desired result, but it is unsatisfactory for a number of reasons. The first most obvious problem is the verbosity required to properly name the types of the lambda's arguments. The alternative would be to instantiate `reduce` and launch it directly:

```
std::vector<int> vec;
...
auto sum = std::async(std:::reduce<std::vector<int>::iterator>, std::par, vec.begin(), vec.end());

std::cout << "sum is " << sum.get() << std::endl;
```

Either solution introduces quite a bit of syntactic noise for such a simple operation. Algorithms with longer argument lists would quickly become burdensome.

Another problem with this approach is more subtle. To achieve asynchrony with `reduce`, we have potentially introduced a new thread of execution through `std::async`. In many interesting situtations this is not what we require. Many parallel architectures follow a client/server model, with a sequential thread of operation issuing asynchronous commands to be parallelized on a server. In these situations, introducing client-side parallelism via `std::async` in order to achieve server-side parallelism via `std::reduce` would be inappropriate.

8.2.3 Generalizing from `std::async`

To both minimize verbosity and allow the programmer to precisely express the parallelism of interest, we propose to generalize `std::async`'s interface to the parallel algorithms library. This would entail modifying each parallel algorithm's interface to return future values as well as introducing a launch policy parameter. For example, the interface for the `std::reduce` family of algorithms might look similar to:

```
template<typename InputIterator>
  future<typename iterator_traits<InputIterator>::value_type>
    reduce(std::launch policy,
           InputIterator first, InputIterator last);

template<typename InputIterator, typename T>
  future<T>
    reduce(std::launch policy,
           InputIterator first, InputIterator last,
           T init);

template<typename InputIterator, typename T, typename BinaryFunction>
  future<T>
    reduce(std::launch policy,
           InputIterator first, InputIterator last,
           T init, BinaryFunction binary_op);
```

We believe such an interface is a good first step towards allowing the programmer to naturally express an asynchronous algorithm launch:

```
std::vector<int> vec;
...
auto sum = std::reduce(std::par, vec.begin(), vec.end());

std::cout << "sum is " << sum.get() << std::endl;
```

8.2.4 Customizing an Algorithm Launch

In the preceding example, we've assumed that the hypothetical standard policy `std::par` instructs the library to asynchronously parallelize the computation with full discretion given to the implementation about the details of parallelization. The value of `std::par` could be thread local, and sequences of algorithm launches on this thread local variable would be implicitly understood to have sequential dependency. However, ideally the type of the launch policy would be rich enough to allow the user to encode details independent of `std::par` which could customize the launch to the architecture. While some of these launch details could be standardized analogously to the existing `std::launch` we expect vendors will wish to provide implementation-specific policies and hints to customize launches to their particular architecture.

For example, an algorithm implementation built on a thread pool such as OpenMP or TBB may wish to allow the programmer to request how many threads of the pool should participate to implement the algorithm. To exploit affinity, implementations built on multicore CPU architectures may wish to allow the user to specify on which cores an algorithm should execute. Similarly, GPGPU vendors may require the user to specify on which GPU (or subset of GPUs) in a multi-GPU system to employ. Analogously, cluster-based implementations of the library might require a set of cluster nodes to enlist. Other more abstract, portable policies which required less detailed knowledge of the underlying architecture would be possible as well. For example, a hint such as oversubscription rate could be adapted to a broad variety of architectures.

Temporary allocations required by many parallel algorithm implementations are inevitable. In addition to specifying what computational resources are required for an algorithm launch, the launch policy must also be expressive enough to encode how memory resources should be allocated. In many interesting settings, the performance of default allocation schemes becomes a critical bottleneck. It makes sense to allow expert programmers to customize these allocations, and the launch policy provides a reasonable place to locate that decision. In Thrust, this is achieved by allowing the user to optionally customize the behavior of the standard library functions `get_temporary_buffer` and `return_temporary_buffer`, the mechanisms by which the

existing standard serial algorithms internally allocate and deallocate temporary memory. We believe this allocation scheme could be standardized along with the parallel algorithms themselves.

In addition to the temporary allocations implicit in many parallel algorithms, customizing the allocation of asynchronous state contained by the resultant `std::future` is also interesting. For some parallel systems which adopt a client/server architecture, it may be impractical for the returned `std::future` to refer to memory easily accessible from the client. In these cases, it makes more sense to locate the state in a place convenient to the server.

To be concrete, consider the asynchronous state associated with the result of `thrust::reduce`. When executed on a GPU, it is expensive to locate that state in CPU memory because it is distant from the GPU. A better approach would allocate storage for the result in memory local to the GPU, and perform an expensive copy to the calling thread only when explicitly requested by the invoking CPU thread through `std::future::get`. Allowing the programmer to the customize the `std::future`'s allocation policy through the launch parameter would nicely capture such use cases.

Because a parallel algorithm launch may depend on the side effects produced by some previous launch, a final concern the asynchronous launch policy should encode is the existence of any asynchronous ancestor event which must be complete before the algorithm launch may proceed. In other words, the launch parameter should allow the programmer to communicate the existence of a `std::future` on which the launch explicitly depends.

To tie all these dependencies into a single algorithm call site, new functionality could be introduced for encapsulating them. For example, suppose the programmer from the previous `reduce` example wished to launch on GPU 0 of her system with a custom temporary allocator:

```
std::vector<float> vec;
int gpu0;
my_temporary_allocator<float> alloc;
float *gpu_ptr;
...
auto event = asynchronous_copy_to_gpu_memory(vec.data(), vec.size(), gpu_ptr);

auto sum = std::reduce(std::launch(event, alloc, gpu0), gpu_ptr, gpu_ptr + vec.size());

std::cout << "sum is " << sum.get() << std::endl;
```

In this way, deviating from the common `std::reduce(std::par, ...)` introduces modest syntactic noise at the call site.

8.2.5 Open Questions

The previous examples focused primarily on the case of a parallel reduction whose result was needed immediately by the invoking thread of execution. In practice, algorithm dependencies are more complex.

Often the output of an algorithm controls the flow of execution and selects which algorithm must be invoked next. To illustrate the point concretely, consider a function `foo` which must operate on sorted data:

```
std::future<void> foo(float *data, size_t n)
{
  // is the data sorted?
  auto data_is_sorted = std::is_sorted(std::par, data, data + n);

  // if not, we need to sort it before continuing
  std::future<void> sort_event;
```

```
  if(!data_is_sorted.get())
  {
    sort_event = std::sort(std::par, data, data + n);
  }

  // depend on the sort_event before consuming the data
  return asynchronously_consume_sorted_data(std::launch(sort_event), data, data + n);
}
```

This example blocks with the calling thread at `is_sorted.get()`. However, the control flow is simple and could be encoded with a richer set of primitives based on `std::future` which would avoid blocking with the controlling thread.

Besides control flow, the arguments of algorithms may themselves depend on future values. Consider a function which partitions and then transforms the true partition of an array:

```
template<typename Iterator, typename Predicate, typename Function>
  std::future<Iterator>
    partition_then_transform_trues(Iterator first, Iterator last,
                                   Predicate pred, Function f)
{
  auto middle = std::partition(std::par, first, last, pred);

  return std::transform(std::par, first, middle.get(), f);
}
```

This implementation of `partition_then_transform_trues` blocks with the invoking thread of execution when it retrieves the value of `middle` to use for `std::transform`'s `last` parameter. In general, any subset of arguments to an algorithm may be a future value. Because we foresee that these patterns of asynchrony will be frequent, we believe an improved interface should automatically encode the dependency on behalf of the caller to avoid synchronization and syntactic noise at the call site.

One approach to this problem would be to allow any parameter of a parallel algorithm to be a `std::future`. One way to encode this generically would be:

```
template<typename FutureOrLaunchPolicy,
         typename FutureOrIterator1,
         typename FutureOrIterator2,
         typename FutureOrT>
  std::future<see-below>
    reduce(FutureOrLaunchPolicy &&policy,
           FutureOrIterator1 &&first, FutureOrIterator2 &&last,
           FutureOrT &&init);
```

This interface could receive either `std::future`s containing the values of parameters or the values of the parameters themselves should they be immediately available. When a parameter is not a `std::future`, a local copy would be made to preserve the value semantics of algorithm parameters. The type of the resulting `std::future` depends on the type of `FutureOrT`.

This sort of flexible interface would transform the previous example to something more agnostic about asynchrony inside the function body:

```
template<typename Iterator, typename Predicate, typename Function>
  std::future<Iterator>
```

```
    partition_then_transform_trues(Iterator first, Iterator last,
                                   Predicate pred, Function f)
{
  auto middle = std::partition(std::par, first, last, pred);

  return std::transform(std::par, first, middle, first, f);
}
```

The parallelism is expressed at the highest level possible with the details of asynchrony hidden within the algorithm implementations themselves.

While we believe such an interface is sufficient to move the problem of blocking on algorithm arguments from the invoking thread to the implementation of the launch of the algorithm, we are concerned that the complexity imposed on the algorithm interface could be intimidating. Ultimately, we seek an interface which allows client code to look as similar as possible to sequences of synchronous algorithm invocations while avoiding unnecessary synchronization.

Satisfying answers to these remaining asynchrony questions are important not only for the usability of a standard parallel algorithms library, but also because we believe that the interface will likely serve as a model for modern libraries of parallel code.

8.3 Additional Algorithms

We believe that the set of high-level parallel algorithms that Thrust provides flexibly compose in powerful ways to build parallel programs but we do not claim that they are a comprehensive solution to all parallel programming problems. For example, while the algorithms which Thrust inherits from the standard algorithms library target one dimensional sequences well, they present an awkward interface for consuming higher dimensional data structures. Linearizing grid-like data structures with dense iterator expressions can be challenging and unproductive. Our rationale for avoiding higher dimensional algorithms in Thrust has been that support for higher dimensional data structures such as matrices and images would be too domain-specific and better served through more targeted libraries.

However, a standard parallel algorithms library would benefit by expanding its set of algorithms beyond those currently available in Thrust. For example, while support for segmented reductions via `thrust::reduce_by_key` exists, currently Thrust features no first class support for segmented sorts. A common use case is to sort individual subsequences within a larger sequence. Though a programmer can achieve the desired effect via two calls to `thrust::stable_sort_by_key`, this method's efficiency vanishes as the average segment size shrinks. First class support for segmented sorts through a named algorithm would better capture this use case and deliver high efficiency.

Another use case currently ill-served by Thrust is segmented reductions whose segment description is small compared to the size of the data. `thrust::reduce_by_key`'s interface consumes a sequence of keys and a sequence of values, both of the same length. While completely general, this dense association of segment identifier with each value implicitly assumes that the average segment size is small. When this is not the case, the memory storage and bandwidth associated with the key sequence is wasted. A complementary segmented reduction interface could accomodate data sets whose segmentation is sparse or whose segmentation follows a predictable pattern. For example, a common use case is to reduce each contiguous `K`-sized subsequence of an array. Not only is `thrust::reduce_by_key`'s interface overkill to describe such segmentation, its implementation is unlikely to be as efficient as a more specialized algorithm.

A final suite of useful functionality would be richer support for generating sequences from a sparse description. Currently, Thrust provides the algorithms `thrust::sequence` which is similar to `std::iota`, and `thrust::generate` and `thrust::fill` which are identical to their standard counterparts. However, we do not believe that these are sufficient to easily generate sequences from patterns frequently occuring in practice. For example, a common use case is to fill an array with a pattern of elements drawn from a source array. With each value of the source array, we associate the number of times it must be replicated in the output

sequence. The values are then output in the order in which they occur in the source. While it is possible to implement such a pattern by combining the non-standard `permutation_iterator` and `counting_iterator` with `thrust::inclusive_scan`, the resulting code is tedious. First class support for patterns like these would be helpful. An interface for this specific use case might look like:

```
template<typename InputIterator1,
         typename InputIterator2,
         typename OutputIterator>
  OutputIterator
    fill_by_count(InputIterator1 counts_first, InputIterator1 counts_last,
                  InputIterator2 values_first,
                  OutputIterator result);
```

It would be elegant if the interface to such sequence generation functions were symmetric with those of the segmented reductions. For example, `fill_by_count` could be complemented with the hypothetical segmented reduction `count_by_key`:

```
template<typename InputIterator,
         typename OutputIterator1,
         typename OutputIterator2>
  pair<OutputIterator1,OutputIterator2>
    count_by_key(InputIterator first, InputIterator last,
                 OutputIterator1 keys_result,
                 OutputIterator2 counts_result);
```

whose output is exactly the input to `fill_by_count` which would generate the original sequence. An investigation into a suite of additional parallel algorithms guided by category theory might be useful here.

8.4 Algorithmic Complexity Guarantees

Providing the parallel programmer with time and space guarantees allows her to make informed choices about the cost of her implementation and avoid surprising runtime behavior. Though Thrust algorithms do not currently provide the programmer with asymptotic complexity guarantees as the C++ standard algorithms do, this is a worthwhile goal. It is straightforward to prove asymptotic complexities for algorithms for an idealized parallel architecture, but practical architectures complicate the issue.

Ultimately we expect that the time and space complexity guarantees a parallel algorithms library could provide would be lax compared to the serial algorithms. In particular, the guarantees provided by most standard serial algorithms assume the existence a particular implementation. In fact, the C++ standard actually prescribes a specific implementation for many of them. But unlike serial algorithms executed within a single thread of execution, parallel algorithms are less likely to have an obviously optimal implementation.

Moreover, the widely varying feature sets of parallel architectures currently make it difficult to prescribe tight complexity bounds that could be satisfied universally. For example, the absence of processor-wide synchronization primitives on NVIDIA GPUs necessitate implementations which "ping-pong" through `O(N)` temporary buffers. The existence of such synchronization primitives could allow implementations of these kinds of algorithms to execute in-place. Such a feature would change their space complexity. Accordingly, it would be premature to provide strict guarantees for these relatively young parallel architectures.

8.5 Exception Propagation

Thrust currently makes no special effort to propagate exceptions arising from the invocation of its parallel algorithms. Instead, it simply relies on the behavior of the underlying parallel backend. Thrust's OpenMP

and TBB implementations spawn CPU threads, which mean that exceptions thrown from user-defined function objects are handled through normal C++ exception propagation. On the other hand, Thrust's CUDA backend need not handle exceptions from user-defined function objects, because they are currently prohibited from GPU code. Ultimately, we expect to define a precise protocol for exception propagation using `std::async`'s behavior as a model.

9 Conclusion

We have argued that a powerful way for C++ to target a broad class of modern architectures is by providing standard support for a suite of parallel algorithms. While this document is not a specification, it outlines a number of the qualities the design of such a library ought to have. Thrust provides evidence that such a library is immediately realizable and at once productive, featureful, and performance portable. We acknowledge the existence of remaining work to integrate such functionality into the C++ Standard Library and anticipate solving these problems armed with constructive feedback and experience gained through additional investigation.