The Parallel-Container Template Library User's Guide

Deepsea Project

10 February 2015

Introduction

The parallel-container template library (pctl) is a C++ library that is currently under development by members of the Deepsea Project. The goal of the pctl is to provide a rich collection of parallel data structures, along with efficient data-parallel algorithms, such as merging, sorting, reduction, etc. We are designing pctl to target shared-memory, parallel (i.e., multicore) platforms.

Our design borrows to a large extent the conventions used by the Standard Template Library. However, when these conventions come into conflict with parallelism, we break with the STL convention.

The pctl provides a rigorous framework for reasoning about the cost of pctl operations. The cost model employed by pctl is the *work/span* model, which is provided by systems, such as Cilk Plus and pasl. Moreover, pctl implements a granularity-control technique that is helpful for taming overheads that relate to parallelization.

The sources of the pctl are made available by a github repository.

Preliminaries

We have implemented pctl to be compatible with the C++11 standard. Earlier versions of the C++ standard may be incompatible.

Downloading and building a pctl program

The pctl library itself consists of several C++ header files. As such, to use the pctl in a C++ project, one must first obtain the source code and then, in order to build with pctl support, pass to the compiler the paths to the required header

files. First, let us begin by downloading the loader script get.sh. Now, the same directory, run the script, specifying the folder in which to store pctl and its dependencies.

```
$ wget http://deepsea.inria.fr/pctl/get.sh
$ get.sh /home/foo/pctl-install/
```

The above command should clone the git repository of the pctl along with the repositories of its two package dependencies, namely `cmdline` and `chunkedseq`. In the example above, our script should have created directories `pctl`, `cmdline`, and `chunkedseq` in the path named `/home/foo/pctl-install/`. Any target path name will do.

Now, let us turn to building a small program, shown below, that specifies an array and computes the sum of the elements.

```
#include "datapar.hpp"

int main() {
  parray<int> xs = { 43, 3222, 11232, 30, 9, -3 };
  int r = pasl::pctl::sum(xs.begin(), xs.end());
  std::cout << "sum(xs) = " << r << std::endl;
  return 0;
}
```

Supposing that we put this program in a file named `sum.cpp`, we can use the following command sequence to build and run an executable.

```
$ g++ -std=c++11 `print-include-directives.sh /home/foo/pctl-install/`
sum.cpp -o sum.exe
$ sum.exe
sum(xs) = 14533
```

The above program is ***not*** compiled with support for parallel execution. Consequently, all calls to pctl functions are going to be performed in a sequential fashion. If we want parallel speedup, we need to pick the parallel library or language that we wish to target and use the appropriate configuration.

Parallelism in pctl can be realized by any library or language extension that brings fork-join parallelism to C++. The following table summarizes which of these systems are currently supported by pctl.

Table 1: Libraries and language extensions that are currently supported by pctl.

	Compiler flag	Description
Sequential elision (default)	`USE_SEQUENTIAL_ELISION_RUNTIME`	Sequentializes all opportunities for parallelism.

Cilk Plus	`USE_CILK_PLUS_RUNTIME`	Uses the Cilk Plus language extension to realize parallelism.
PASL	`USE_PASL_RUNTIME`	Uses the PASL system to realize parallelism.

To build the example shown above, but with support for Cilk Plus, first make sure that you are using `gcc >= 5.0` or a recent version of clang/llvm that supports Cilk Plus extensions. If so, the command shown below will build a parallel-ready binary.

```
$ g++ -std=c++11 `print-include-directives.sh /home/foo/pctl-install/`
-fcilkplus -lcilkrts -DUSE_CILK_PLUS_RUNTIME sum.cpp -o sum.exe
...
```

TODO document pasl support

The substrate for expressing parallelism

Binary fork join

```
namespace pasl {
namespace pctl {

template <class First_function, class Second_function>
void fork2(First_function f1, Second_function f2);

} }
```

In the pctl, an opportunity for parallelism can be expressed by an application of the binary fork-join primitive. The interface of this primitive is shown above. The call `fork2(f1, f2)` performs the calls `f1()` and `f2()`, allowing both calls to run in parallel. To see how we might use this primitive, consider the program below.

```
#include "datapar.hpp"

using namespace pasl::pctl;

template <class Iterator>
int my_sum(Iterator lo, Iterator hi) {
  int result = 0;
  int n = hi - lo;
  if (n == 0)
    result = 0;
  } else if (n == 1) {
```

```
    result = *lo;
  } else {
    Iterator mid = lo + (n / 2);
    int result1, result2;
    fork2([&] {
      result1 = sum(lo, mid);
    }, {
      result2 = sum(mid, hi);
    });
    result = result1 + result2;
  }
  return result;
}

int main() {
  parray<int> xs = { 43, 3222, 11232, 30, 9, -3 };
  std::cout << "my_sum(xs) = " << my_sum(xs.begin(), xs.end()) << std::endl;
  return 0;
}
```

In this program, we have a function named `my_sum` that returns the sum of a range of items in some container, realizing parallelism via the call to the pctl fork-join primitive. Unfortunately, this program is naive because of the very fine-grain use of the binary fork-join primitive. The good news is that, with a little extra annotation in the source code, one can easily bypass this problem and obtain a fast, clean, and concise solution.

Although parallel systems, such as Cilk Plus and PASL, are carefully engineered to control overheads, the reality is that, in general, these overheads may still be large enough to seriously harm performance. This problem is known as the problem of ***granularity control***. One way to characterize the granularity-control problem is as follows. On the one hand, if the program tries to exploit too many opportunities for parallelism, then the program will be slow due to overheads of the parallel primitives. On the other, if the program realizes too little parallelism, the program will be slow because of underutilized processors. For this reason, programmers often resort to manual granularity control, whereby a threshold is used to prune parallelism. In the `my_sum` function shown above, for example, one may address the problem by stopping recursion below some specified threshold.

Unfortunately, such a manual approach has some severe problems of its own. In brief, manual granularity control is not portable across different platforms and does not compose well in situations where the program uses nested parallelism or higher-order functions, such as map and reduce. Although well understood, there is currently no silver bullet to solve the granularity-control problem in general. However, there is one approach, named ***oracle-based granularity control***, that achieves mostly automatic granularity control. The original idea is described in

some recent research publications. The pctl implements a particular incarnation of oracle-guided scheduling.

Mostly automatic granularity control

In the pctl, oracle-guided scheduling is implemented by a mechanism called the ***controlled statement***. A controlled statement is (1) a block of code, expressed as a lambda function, that, when called, performs some specified parallel computation and (2) a ***complexity function*** that specifies the asymptotic complexity of the block of code.

```
namespace pasl {
namespace pctl {

// (1)
template <class Complexity_function, class Body>
void cstmt(Complexity_function cf, Body b);

// (2)
template <class Complexity_function, class Parallel_body, class Sequential_body>
void cstmt(Complexity_function cf, Parallel_body pb, Sequential_body sb);

} }
```

There are two variants of the controlled statement. Let us focus on the first one for now because the first one is simpler. The call `cstmt(cf, b)` performs the call `b()`, using the result of `cf()` to help determine whether to realize opportunities for parallelism or to execute the call `b()` serially. In the code shown below, we see how to extend our `my_sum` function to achieve granularity control by use of the controlled statement.

```
template <class Iterator>
int my_sum(Iterator lo, Iterator hi) {
  int result = 0;
  int n = hi - lo;
  cstmt([&] { return n; }, [&] {
    if (n == 0)
      result = 0;
    } else if (n == 1) {
      result = *lo;
    } else {
      Iterator mid = lo + (n / 2);
      int result1, result2;
      fork2([&] {
        result1 = sum(lo, mid);
      }, {
```

```
      result2 = sum(mid, hi);
    });
    result = result1 + result2;
  }
 });
 return result;
}
```

Now, with the above code, we have a much more efficient program than we had before because we are using automatic granularity control to deal with parallelism-related overheads. But we can do better in this case because we have a `my_sum` function for which the serial portion of the computation can be expressed more efficiently as a serial loop.

For this reason, the pctl provides the second variant of the controlled statement. The call `cstmt(cf, pb, sb)` performs the call `pb()` if the system chooses to execute the given computation in a parallel fashion and the call `sb()` if the system chooses to sequentialize the call. As before, the result `cf()` of the complexity function is used by the system to determine whether to use parallel or serial execution. We can modify our `my_sum` example as shown below to use a serial body.

```
template <class Iterator>
int my_sum(Iterator lo, Iterator hi) {
  int result = 0;
  int n = hi - lo;
  cstmt([&] { return n; }, [&] {
    // parallel body
    if (n == 0)
      result = 0;
    } else if (n == 1) {
      result = *lo;
    } else {
      Iterator mid = lo + (n / 2);
      int result1, result2;
      fork2([&] {
        result1 = sum(lo, mid);
      }, {
        result2 = sum(mid, hi);
      });
      result = result1 + result2;
    }
  }, [&] {
    // serial body
    for (; lo != hi; lo++) {
      result += *lo;
    }
```

```
  });
  return result;
}
```

Containers

Our definition of a conainer is the same as that of STL: a *container* is a holder object that stores a collection of other objects (its elements). Elements are implemented as class templates, which allows flexibility to specify at compile time the type of the elements to be stored in any given container.

The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

The particular containers defined by pctl fill what we believe are important gaps in the current state-of-the-art C++ container libraries. In specific, the gaps relate to data structures that can always be resized and accessed efficiently in parallel. Note that, although we designed the pctl containers to useful for *parallel programming*, we have not designed them to be useful for *concurrent programming*. A discussion of the difference between parallel and concurrent programming is out of the scope of this manual, but can be found here. What this property means for clients of pctl is that the container operations that modify the container (e.g., `push` or `pop`) generally are not thread safe. Parallelism is achieved instead by operating in parallel on disjoint ranges of, say, the same sequence, or by splitting and merging sequences in a fork-join fashion.

Sequence containers

Table 2: Sequence containers that are provided by pctl.

Class name	Description
`parray`	Contiguous array class
`pchunkedseq`	Discontiguous, efficiently resizable sequence class

Associative containers

Table 3: Associative containers that are provided by pctl.

Class name	Description
`pset`	Set class
`pmap`	Associative-map class

Class name	Description

String containers

Table 4: String containers that are provided by pctl.

Class name	Description
`pstring`	Array-based string class

Parallel array

```
namespace pasl {
namespace pctl {

template <class Item, class Alloc = std::allocator<Item>>
class parray;

} }
```

Parallel arrays are containers representing arrays that are populated with their elements in parallel. Although they can be resized, the work cost of resizing a parallel array is linear. In other words, parallel arrays are good for bulk-parallel processing, but not so good for incremental updates.

Just like traditional C++ arrays (i.e., raw memory allocated by, say, `malloc`), parallel arrays use contiguous storage to hold their elements. As such, elements can be accessed as efficiently as C arrays by using offsets or by using pointer values. Unlike C arrays, storage is managed automatically by the container. Moreover, unlike C arrays, parallel arrays initialize and de-initialize their cells in parallel.

Template parameters

Table 5: Template parameters for the `parray` class.

Template parameter	Description
`Item`	Type of the objects to be stored in the container
`Alloc`	Allocator to be used by the container to construct and destruct objects of type `Item`

Template parameter	Description

Item type

```
class Item;
```

Type of the elements. Only if `Item` is guaranteed to not throw while moving, implementations can optimize to move elements instead of copying them during reallocations. Aliased as member type `parray::value_type`.

Allocator

```
class Alloc;
```

Type of the allocator object used to define the storage allocation model. By default, the allocator class template is used, which defines the simplest memory allocation model and is value-independent. Aliased as member type `parray::allocator_type`.

Member types

Table 6: Parallel-array member types.

Type	Description
`value_type`	Alias for template parameter `Item`
`reference`	Alias for `value_type&`
`const_reference`	Alias for `const value_type&`
`pointer`	Alias for `value_type*`
`const_pointer`	Alias for `const value_type*`
`iterator`	Iterator
`const_iterator`	Const iterator

Iterator

The type `iterator` and `const_iterator` are instances of the random-access iterator concept.

Constructors and destructors

Table 7: Parallel-array constructors and destructors.

Constructor	Description
empty container constructor (default constructor)	constructs an empty container with no items
fill constructor	constructs a container with a specified number of copies of a given item
populate constructor	constructs a container with a specified number of values that are computed by a specified function
copy constructor	constructs a container with a copy of each of the items in the given container, in the same order
initializer list	constructs a container with the items specified in a given initializer list
move constructor	constructs a container that acquires the items of a given parallel array
destructor	destructs a container

Empty container constructor

```
parray();
```

Complexity. Constant time.

Constructs an empty container with no items;

Fill container

```
parray(long n, const value_type& val);
```

Constructs a container with `n` copies of `val`.

Complexity. Work and span are linear and logarithmic in the size of the resulting container, respectively.

Populate constructor

```
// (1) Constant-time body
parray(long n, std::function<Item(long)> body);
// (2) Non-constant-time body
parray(long n,
       std::function<long(long)> body_comp,
       std::function<Item(long)> body);
```

```
// (3) Non-constant-time body along with range-based complexity function
parray(long n,
       std::function<long(long,long)> body_comp_rng,
       std::function<Item(long)> body);
```

Constructs a container with `n` cells, populating those cells with values returned by the `n` calls, `body(0)`, `body(1)`, ..., `body(n-1)`, in that order.

In the second version, the value returned by `body_comp(i)` is used by the constructor as the complexity estimate for the call `body(i)`.

In the third version, the value returned by `body_comp(lo, hi)` is used by the constructor as the complexity estimate for the calls `body(lo)`, `body(lo+1)`, ... `body(hi-1)`.

Complexity. The work and span cost are $(\sum_{0 \leq i < n} w(i))$ and $(\log n + \max_{0 \leq i < n} s(i))$, respectively, where $w(i)$ represents the work cost of computing `body`(i) and $s(i)$ the corresponding span cost.

Copy constructor

```
parray(const parray& other);
```

Constructs a container with a copy of each of the items in `other`, in the same order.

Complexity. Work and span are linear and logarithmic in the size of the resulting container, respectively.

Initializer-list constructor

```
parray(initializer_list<value_type> il);
```

Constructs a container with the items in `il`.

Complexity. Work and span are linear in the size of the resulting container.

Move constructor

```
parray(parray&& x);
```

Constructs a container that acquires the items of `other`.

Complexity. Constant time.

Destructor

```
~parray();
```

Destructs the container.

Complexity. Work and span are linear and logarithmic in the size of the container, respectively.

Member functions

Table 8: Parallel-array member functions.

Operation	Description
`operator[]`	Access member item
`size`	Return size
`clear`	Empty container contents
`resize`	Change size
`swap`	Exchange contents
`begin cbegin`	Returns an iterator to the beginning
`end cend`	Returns an iterator to the end

Indexing operator

```
reference operator[](long i);
const_reference operator[](long i) const;
```

Returns a reference at the specified location `i`. No bounds check is performed.

Complexity. Constant time.

Size operator

```
long size() const;
```

Returns the size of the container.

Complexity. Constant time.

Clear

```
void clear();
```

Erases the contents of the container, which becomes an empty container.

Complexity. Work and span are linear and logarithmic in the size of the container, respectively.

Iterator validity. Invalidates all iterators, if the size before the operation differs from the size after.

Resize

```
void resize(long n, const value_type& val); // (1)
void resize(long n) {                       // (2)
  value_type val;
  resize(n, val);
}
```

Resizes the container to contain `n` items.

If the current size is greater than `n`, the container is reduced to its first `n` elements.

If the current size is less than `n`,

1. additional copies of `val` are appended
2. additional default-inserted elements are appended

Complexity. Let m be the size of the container just before and n just after the resize operation. Then, the work and span are linear and logarithmic in $\max(m, n)$, respectively.

Iterator validity. Invalidates all iterators, if the size before the operation differs from the size after.

Exchange operation

```
void swap(parray& other);
```

Exchanges the contents of the container with those of `other`. Does not invoke any move, copy, or swap operations on individual items.

Complexity. Constant time.

Iterator begin

```
iterator begin() const;
const_iterator cbegin() const;
```

Returns an iterator to the first item of the container.

If the container is empty, the returned iterator will be equal to end().

Complexity. Constant time.

Iterator end

```
iterator end() const;
const_iterator cend() const;
```

Returns an iterator to the element following the last item of the container.

This element acts as a placeholder; attempting to access it results in undefined behavior.

Complexity. Constant time.

Parallel chunked sequence

```
namespace pasl {
namespace pctl {

template <class Item, class Alloc = std::allocator<Item>>
class pchunkedseq;

} }
```

Parallel chunked sequences are containers representing sequences. Elements of the container are populated with their elements in parallel. The container itself can be resized in an incremental fashion in amortized constant time and can be split at a specified position and concatenated in logarithmic time.

Unlike parallel arrays, parallel chunked sequences use small chunks, usually sized to contain between 512 and 1024 items. Internally, the chunks are stored at the leaves of a self-balancing tree. The tree is, for all practical purposes, no more than eight levels deep. The space usage of the sequence is, in practice, never more than 20% greater than the space of a corresponding array.

The parallel chunked sequence class is a wrapper class whose sole member object is a sequential chunked sequence. The wrapper function ensures that, when accessed in bulk fashion, the sequential chunked sequence is processed in parallel. The parallel chunked sequence class defines wrapper methods for many of the methods of the sequential chunked sequence class. However, additional methods can be accessed by directly calling methods of the sequential chunked sequence object. The complete interface of the sequential chunked sequence class is documented here.

Template parameters

Table 9: Template parameters for the `pchunkedseq` class.

Template parameter	Description
`Item`	Type of the objects to be stored in the container
`Alloc`	Allocator to be used by the container to construct and destruct objects of type `Item`

Item type

```
class Item;
```

Type of the elements. Only if `Item` is guaranteed to not throw while moving, implementations can optimize to move elements instead of copying them during reallocations. Aliased as member type `parray::value_type`.

Allocator

```
class Alloc;
```

Type of the allocator object used to define the storage allocation model. By default, the allocator class template is used, which defines the simplest memory allocation model and is value-independent. Aliased as member type `parray::allocator_type`.

Member types

Table 10: Parallel chunked sequence member types.

Type	Description
`value_type`	Alias for template parameter `Item`
`reference`	Alias for `value_type&`
`const_reference`	Alias for `const value_type&`
`pointer`	Alias for `value_type*`
`const_pointer`	Alias for `const value_type*`
`iterator`	Iterator
`const_iterator`	Const iterator
`seq_type`	Alias for `data::chunkedseq::bootstrapped::deque<Item>`
`segment_type`	Alias for `typename seq_type::segment_type`

Type	Description
`const_segment_type`	Alias for `typename seq_type::const_segment_type`

Iterator

The type `iterator` and `const_iterator` are instances of the random-access iterator concept.

Additionally, the iterator supports *segment-wise access*, which allows the client of the iterator to query an iterator value to discover the contiguous region of memory in the same chunk that stores nearby items. The documentation on chunked sequences describes this interface in detail.

Sequential chunked sequence type

The class which implements this type is described in detail here.

Segment type

The segment type, when instantiated by the paralle chunked sequence class, looks like the following class.

```
class segment {
public:

  pointer begin;
  pointer middle;
  pointer end;

  segment()
  : begin(nullptr), middle(nullptr), end(nullptr) { }

  segment(pointer begin, pointer middle, pointer end)
  : begin(begin), middle(middle), end(end) { }

};
```

The value `begin` points to the first item in the segment, the value `middle` to the item of interest in the segment, and `end` to the address one past the end of the sequence.

Segments are described in detail here.

Const segment type

The const segment type looks the same as the segment class above, except that `pointer` types are replaced by `const_pointer` types.

Member objects

Table 11: Parallel chunked sequence member objects.

Member objects	Description
`seq`	Sequential chunked sequence

Sequential chunked sequence

```
seq_type seq;
```

This object provides the storage for all the elements in the container.

Constructors and destructors

Table 12: Parallel chunked sequence constructors and destructors.

Constructor	Description
empty container constructor (default constructor)	constructs an empty container with no items
fill constructor	constructs a container with a specified number of copies of a given item
populate constructor	constructs a container with a specified number of values that are computed by a specified function
copy constructor	constructs a container with a copy of each of the items in the given container, in the same order
initializer list	constructs a container with the items specified in a given initializer list
move constructor	constructs a container that acquires the items of a given parallel array
destructor	destructs a container

Empty container constructor

```
pchunkedseq();
```

Complexity. Constant time.

Constructs an empty container with no items;

Fill container

```
pchunkedseq(long n, const value_type& val);
```

Constructs a container with `n` copies of `val`.

Complexity. Work and span are linear and logarithmic in the size of the resulting container, respectively.

Populate constructor

```
// (1) Constant-time body
pchunkedseq(long n, std::function<Item(long)> body);
// (2) Non-constant-time body
pchunkedseq(long n,
            std::function<long(long)> body_comp,
            std::function<Item(long)> body);
// (3) Non-constant-time body along with range-based complexity function
pchunkedseq(long n,
            std::function<long(long,long)> body_comp_rng,
            std::function<Item(long)> body);
```

Constructs a container with `n` cells, populating those cells with values returned by the `n` calls, `body(0)`, `body(1)`, ..., `body(n-1)`, in that order.

In the second version, the value returned by `body_comp(i)` is used by the constructor as the complexity estimate for the call `body(i)`.

In the third version, the value returned by `body_comp(lo, hi)` is used by the constructor as the complexity estimate for the calls `body(lo)`, `body(lo+1)`, ... `body(hi-1)`.

Complexity. The work and span cost are $(\sum_{0 \leq i < n} w(i))$ and $(\log n + \max_{0 \leq i < n} s(i))$, respectively, where $w(i)$ represents the work cost of computing `body`(i) and $s(i)$ the corresponding span cost.

Copy constructor

```
pchunkedseq(const pchunkedseq& other);
```

Constructs a container with a copy of each of the items in `other`, in the same order.

Complexity. Work and span are linear and logarithmic in the size of the resulting container, respectively.

Initializer-list constructor

```
pchunkedseq(initializer_list<value_type> il);
```

Constructs a container with the items in `il`.

Complexity. Work and span are linear in the size of the resulting container.

Move constructor

```
pchunkedseq(pchunkedseq&& x);
```

Constructs a container that acquires the items of `other`.

Complexity. Constant time.

Destructor

```
~pchunkedseq();
```

Destructs the container.

Complexity. Work and span are linear and logarithmic in the size of the container, respectively.

Member functions

Table 13: Operations of the parallel chunked sequence.

Operation	Description
`operator[]`	Access member item
`size`	Return size
`swap`	Exchange contents
`begin cbegin`	Returns an iterator to the beginning
`end cend`	Returns an iterator the end
`clear`	Erases contents of the container
`tabulate`	Repopulate container changing size
`resize`	Change size

Indexing operator

```
reference operator[](long i);
const_reference operator[](long i) const;
```

Returns a reference at the specified location `i`. No bounds check is performed.

Complexity. Logarithmic time. The complexity is, however, for practical purposes, constant time, because thanks to the high branching factor the height of the tree is, in practice, never more than eight.

Size operator

```
long size() const;
```

Returns the size of the container.

Complexity. Constant time.

Exchange operation

```
void swap(pchunkedseq& other);
```

Exchanges the contents of the container with those of `other`. Does not invoke any move, copy, or swap operations on individual items.

Complexity. Constant time.

Iterator begin

```
iterator begin() const;
const_iterator cbegin() const;
```

Returns an iterator to the first item of the container.

If the container is empty, the returned iterator will be equal to end().

Complexity. Constant time.

Iterator end

```
iterator end() const;
const_iterator cend() const;
```

Returns an iterator to the element following the last item of the container.

This element acts as a placeholder; attempting to access it results in undefined behavior.

Complexity. Constant time.

Clear

```
void clear();
```

Erases the contents of the container, which becomes an empty container.

Complexity. Work and span are linear and logarithmic in the size of the container, respectively.

Iterator validity. Invalidates all iterators, if the size before the operation differs from the size after.

Tabulate

```
void tabulate(long n, std::function<value_type(long)> body);
void tabulate(long n,
              std::function<long(long)> body_comp_rng,
              std::function<value_type(long)> body);
```

Resizes the container so that it contains `n` items.

The contents of the current container are removed and replaced by the `n` items returned by the `n` calls, `body(0)`, `body(1)`, ..., `body(n-1)`, in that order.

Complexity. The work and span cost are $(\sum_{0 \leq i < n} w(i))$ and $(\log n + \max_{0 \leq i < n} s(i))$, respectively, where $w(i)$ represents the work cost of computing `body`(i) and $s(i)$ the corresponding span cost.

Iterator validity. Invalidates all iterators, if the size before the operation differs from the size after.

Resize

```
void resize(long n, const value_type& val);  // (1)
void resize(long n) {                         // (2)
  value_type val;
  resize(n, val);
}
```

Resizes the container to contain `n` items.

If the current size is greater than `n`, the container is reduced to its first `n` elements.

If the current size is less than `n`,

1. additional copies of `val` are appended
2. additional default-inserted elements are appended

Complexity. Let m be the size of the container just before and n just after the resize operation. Then, the work and span are linear and logarithmic in $\max(m, n)$, respectively.

Iterator validity. Invalidates all iterators, if the size before the operation differs from the size after.

Non-member functions

Table 14: Functions relating to the parallel chunked sequence.

Function	Description
`for_each`	Applies a given function to each item in the container
`for_each_segment`	Applies a given function to each segment in the container

For each

```
namespace pasl {
namespace pctl {
namespace segmented {

template <class Body>
void for_each(Body body);

} } }
```

Performs the calls `body(x1)`, ..., `body(xn)` for each item `xi` in the container.

The class `Body` should provide a call operator of the following type. The reference `xi` points to the i^{th} item in the container.

```
void operator()(Item& xi);
```

Complexity. The work and span cost are $(\sum_{0 \leq i < n} w(i))$ and $(\log n + \max_{0 \leq i < n} s(i))$, respectively, where $w(i)$ represents the work cost of computing `body`(i) and $s(i)$ the corresponding span cost.

For each segment

```
namespace pasl {
namespace pctl {
namespace segmented {
```

```
template <class Body_seg>
void for_each_segment(Body_seg body_seg);

} } }
```

Performs the calls `body_seg(lo1, hi1)`, ..., `body(lon, hin)` for each segment of items (`loi`, `hii`) in the container.

The class `Body_seg` should provide a call operator of the following type. The `lo` value points to the first item in the segment and `hi` one past the end of the segment.

```
void operator()(pointer lo, pointer hi);
```

Complexity. The work and span cost are $(\sum_{0 \leq i < n} w(i))$ and $(\log n + \max_{0 \leq i < n} s(i))$, respectively, where $w(i)$ represents the work cost of computing `body_seg`$(i, i+1)$ and $s(i)$ the corresponding span cost.

Parallel set

```
namespace pasl {
namespace pctl {

template <
  class Item,
  class Compare = std::less<Item>,
  class Alloc = std::allocator<Item>,
  int chunk_capacity = 8
>
class pset;

} }
```

The set container stores unique elements in ascending order. Order among the container elements is maintained by a comparison operator, which is provided by the template parameter `Compare`. The value of the elements in the set container cannot be modified once in the container, but they can be inserted or removed from the container.

Just like the STL `set` container, our `pset` container provides `insert` and `erase` methods. Both methods take logarithmic time in the size of the container.

```
pset<int> s;
s.insert(5);
s.insert(432);
s.insert(5);
s.insert(89);
```

```
s.erase(5);
std::cout << "s = " << s << std::endl;
```

This program prints the following.

```
s = { 89, 432 }
```

Unlike an STL `set`, our `pset` provides bulk set operations, including union (i.e., merge), intersection, and difference. Moreover, these methods are highly parallel: they take linear work and logarithmic span in the total size of the two containers being combined. The `merge` method computes the set union with a given container, leaving the result in the targeted container and leaving the given container empty.

```
pset<int> s1 = { 3, 1, 5, 8, 12 };
pset<int> s2 = { 3, 54, 8, 9 };
s1.merge(s2);
std::cout << "s1 = " << s1 << std::endl;
std::cout << "s2 = " << s2 << std::endl;
```

The output:

```
s1 = { 1, 3, 5, 8, 9, 12, 54 }
s2 = {  }
```

Set intersection is handled in a similar fashion, by the `intersection` method.

```
pset<int> s1 = { 3, 1, 5, 8, 12 };
pset<int> s2 = { 3, 54, 8, 9 };
s1.intersect(s2);
std::cout << "s1 = " << s1 << std::endl;
std::cout << "s2 = " << s2 << std::endl;
```

The output:

```
s1 = { 3, 8 }
s2 = {  }
```

Set difference is handled similarly by the `diff` method.

```
pset<int> s1 = { 3, 1, 5, 8, 12 };
pset<int> s2 = { 3, 54, 8, 9 };
s1.diff(s2);
std::cout << "s1 = " << s1 << std::endl;
std::cout << "s2 = " << s2 << std::endl;
```

The output:

```
s1 = { 1, 5, 12 }
s2 = {  }
```

Template parameters

Table 15: Template parameters for the `pset` class.

Template parameter	Description
`Item`	Type of the objects to be stored in the container
`Compare`	Type of the comparison function
`Alloc`	Allocator to be used by the container to construct and destruct objects of type `Item`
`chunk_capacity`	Capacity of the contiguous chunks of memory that are used by the container to store items

Item type

```
class Item;
```

Type of the elements. Only if `Item` is guaranteed to not throw while moving, implementations can optimize to move elements instead of copying them during reallocations. Aliased as member type `pset::value_type`.

Compare function

```
class Compare;
```

```
bool operator()(const Item& lhs, const Item& rhs);
```

Allocator

```
class Alloc;
```

Type of the allocator object used to define the storage allocation model. By default, the allocator class template is used, which defines the simplest memory allocation model and is value-independent. Aliased as member type `pset::allocator_type`.

Chunk capacity

```
int chunk_capacity;
```

An integer which determines the maximum size of a contiguous chunk of memory in the underlying container. The chunks are used by the container to store the items. The minimum size of a chunk is guaranteed by the container to be at least `chunk_size/2`.

Member types

Table 16: Parallel-set member types.

Type	Description
`value_type`	Alias for template parameter `Item`
`key_type`	Alias for template parameter `Item`
`key_compare`	Alias for template parameter `Compare`
`reference`	Alias for `value_type&`
`const_reference`	Alias for `const value_type&`
`pointer`	Alias for `value_type*`
`const_pointer`	Alias for `const value_type*`
`iterator`	Iterator
`const_iterator`	Const iterator

Iterator

The type `iterator` and `const_iterator` are instances of the random-access iterator concept.

Constructors and destructors

Table 17: Parallel set constructors and destructors.

Constructor	Description
empty container constructor (default constructor)	constructs an empty container with no items
fill constructor	constructs a container with a specified number of copies of a given item
populate constructor	constructs a container with a specified number of values that are computed by a specified function
copy constructor	constructs a container with a copy of each of the items in the given container, in the same order

Constructor	Description
initializer list	constructs a container with the items specified in a given initializer list
move constructor	constructs a container that acquires the items of a given parallel array
destructor	destructs a container

Empty container constructor

```
pset();
```

Complexity. Constant time.

Constructs an empty container with no items;

Fill container

```
pset(long n, const value_type& val);
```

Constructs a container with `n` copies of `val`.

Complexity. Work and span are linear and logarithmic in the size of the resulting container, respectively.

Populate constructor

```
// (1) Constant-time body
pset(long n, std::function<Item(long)> body);
// (2) Non-constant-time body
pset(long n,
     std::function<long(long)> body_comp,
     std::function<Item(long)> body);
// (3) Non-constant-time body along with range-based complexity function
pset(long n,
     std::function<long(long,long)> body_comp_rng,
     std::function<Item(long)> body);
```

Constructs a container with `n` cells, populating those cells with values returned by the `n` calls, `body(0)`, `body(1)`, ..., `body(n-1)`, in that order.

In the second version, the value returned by `body_comp(i)` is used by the constructor as the complexity estimate for the call `body(i)`.

In the third version, the value returned by `body_comp(lo, hi)` is used by the constructor as the complexity estimate for the calls `body(lo)`, `body(lo+1)`, ... `body(hi-1)`.

Complexity. The work and span cost are $(\sum_{0 \leq i < n} w(i))$ and $(\log n + \max_{0 \leq i < n} s(i))$, respectively, where $w(i)$ represents the work cost of computing `body`(i) and $s(i)$ the corresponding span cost.

Copy constructor

```
pset(const pset& other);
```

Constructs a container with a copy of each of the items in `other`, in the same order.

Complexity. Work and span are linear and logarithmic in the size of the resulting container, respectively.

Initializer-list constructor

```
pset(initializer_list<value_type> il);
```

Constructs a container with the items in `il`.

Complexity. Work and span are linear in the size of the resulting container.

Move constructor

```
pset(pset&& x);
```

Constructs a container that acquires the items of `other`.

Complexity. Constant time.

Destructor

```
~pset();
```

Destructs the container.

Complexity. Work and span are linear and logarithmic in the size of the container, respectively.

Member functions

Table 18: Operations of the parallel set.

Operation	Description
`size`	Return size

Operation	Description
`swap`	Exchange contents
`begin cbegin`	Returns an iterator to the beginning
`end cend`	Returns an iterator the end
`clear`	Erases contents of the container
`find`	Get iterator to element
`insert`	Insert element
`erase`	Remove element
`merge`	Take set union with given pset
`intersect`	Take set intersection with given pset
`diff`	Take set difference with give pset

Size operator

```
long size();
```

Returns the size of the container.

Complexity. Constant time.

Exchange oparation

```
void swap(pset& other);
```

Exchanges the contents of the container with those of other. Does not invoke any move, copy, or swap operations on individual items.

Complexity. Constant time.

Iterator begin

```
iterator begin() const;
const_iterator cbegin() const;
```

Returns an iterator to the first item of the container.

If the container is empty, the returned iterator will be equal to end().

Complexity. Constant time.

Iterator end

```
iterator end() const;
const_iterator cend() const;
```

Returns an iterator to the element following the last item of the container.

This element acts as a placeholder; attempting to access it results in undefined behavior.

Complexity. Constant time.

Clear

```
void clear();
```

Erases the contents of the container, which becomes an empty container.

Complexity. Work and span are linear and logarithmic in the size of the container, respectively.

Iterator validity. Invalidates all iterators, if the size before the operation differs from the size after.

Find

```
itereator find(const value_type& val);
```

Searches the container for an element equivalent to `val` and returns an iterator to it if found, otherwise it returns an iterator to `pset::end`.

Two elements of a set are considered equivalent if the container's comparison object returns false reflexively (i.e., no matter the order in which the elements are passed as arguments).

Complexity. Logarithmic time.

Insert

```
pair<iterator,bool> insert (const value_type& val);
```

Extends the container by inserting new elements, effectively increasing the container size by the number of elements inserted.

Because elements in a set are unique, the insertion operation checks whether each inserted element is equivalent to an element already in the container, and if so, the element is not inserted, returning an iterator to this existing element (if the function returns a value).

Complexity. Logarithmic time.

Iterator validity. Invalidates all iterators, if the size before the operation differs from the size after.

Erase

```
size_type erase (const value_type& val);
```

Removes from the set the value `val`, if it is present.

This effectively reduces the container size by the number of elements removed, which are destroyed.

Complexity. Logarithmic time.

Iterator validity. Invalidates all iterators, if the size before the operation differs from the size after.

Set union

```
void merge(pset& other);
```

Leaves in the current container the result of taking the set union of the original container and the `other` container and leaves the `other` container empty.

Complexity. Work and span are linear and logarithmic in the combined sizes of the two containers, respectively.

Iterator validity. Invalidates all iterators.

Set intersection

```
void intersect(pset& other);
```

Leaves in the current container the result of taking the set intersection of the original container and the `other` container and leaves the `other` container empty.

Complexity. Work and span are linear and logarithmic in the combined sizes of the two containers, respectively.

Iterator validity. Invalidates all iterators.

Set difference

```
void diff(pset& other);
```

Leaves in the current container the result of taking the difference intersection between the original container and the `other` container and leaves the `other` container empty.

Complexity. Work and span are linear and logarithmic in the combined sizes of the two containers, respectively.

Iterator validity. Invalidates all iterators.

Parallel map

Parallel string

```
namespace pasl {
namespace pctl {

template <class Alloc = std::allocator<Item>>
class pstring;

} }
```

Member types

Table 19: Parallel string member types.

Type	Description
`value_type`	Alias for type `char`
`reference`	Alias for `value_type&`
`const_reference`	Alias for `const value_type&`
`pointer`	Alias for `value_type*`
`const_pointer`	Alias for `const value_type*`
`iterator`	Iterator
`const_iterator`	Const iterator

Iterator

The type `iterator` and `const_iterator` are instances of the random-access iterator concept.

Constructors and destructors

Table 20: Parallel-string constructors and destructors.

Constructor	Description
empty container constructor (default constructor)	constructs an empty container with no items
fill constructor	constructs a container with a specified number of copies of a given item

Constructor	Description
populate constructor	constructs a container with a specified number of values that are computed by a specified function
copy constructor	constructs a container with a copy of each of the items in the given container, in the same order
initializer list	constructs a container with the items specified in a given initializer list
move constructor	constructs a container that acquires the items of a given parallelstring
destructor	destructs a container

Empty container constructor

```
pstring();
```

Complexity. Constant time.

Constructs an empty container with no items;

Fill container

```
pstring(long n, char val);
```

Constructs a container with `n` copies of `val`.

Complexity. Work and span are linear and logarithmic in the size of the resulting container, respectively.

Populate constructor

```
// (1) Constant-time body
pstring(long n, std::function<char(long)> body);
// (2) Non-constant-time body
pstring(long n,
        std::function<long(long)> body_comp,
        std::function<char(long)> body);
// (3) Non-constant-time body along with range-based complexity function
pstring(long n,
        std::function<long(long,long)> body_comp_rng,
        std::function<char(long)> body);
```

Constructs a container with `n` cells, populating those cells with values returned by the `n` calls, `body(0)`, `body(1)`, ..., `body(n-1)`, in that order.

In the second version, the value returned by `body_comp(i)` is used by the constructor as the complexity estimate for the call `body(i)`.

In the third version, the value returned by `body_comp(lo, hi)` is used by the constructor as the complexity estimate for the calls `body(lo)`, `body(lo+1)`, ... `body(hi-1)`.

Complexity. The work and span cost are $(\sum_{0 \leq i < n} w(i))$ and $(\log n + \max_{0 \leq i < n} s(i))$, respectively, where $w(i)$ represents the work cost of computing `body`(i) and $s(i)$ the corresponding span cost.

Copy constructor

```
pstring(const pstring& other);
```

Constructs a container with a copy of each of the items in `other`, in the same order.

Complexity. Work and span are linear and logarithmic in the size of the resulting container, respectively.

Initializer-list constructor

```
pstring(initializer_list<value_type> il);
```

Constructs a container with the items in `il`.

Complexity. Work and span are linear in the size of the resulting container.

Move constructor

```
pstring(pstring&& x);
```

Constructs a container that acquires the items of `other`.

Complexity. Constant time.

Destructor

```
~pstring();
```

Destructs the container.

Complexity. Work and span are linear and logarithmic in the size of the container, respectively.

Operations

Table 21: Parallel-array member functions.

Operation	Description
`operator[]`	Access member item
`size`	Return size
`clear`	Erase contents
`resize`	Change size
`swap`	Exchange contents
`begin cbegin`	Returns an iterator to the beginning
`end cend`	Returns an iterator to the end
`operator+=`	Append to string
`operator+`	Concatenates strings
`c_str`	Get C string equivalent

Indexing operator

```
reference operator[](long i);
const_reference operator[](long i) const;
```

Returns a reference at the specified location `i`. No bounds check is performed.

Complexity. Constant time.

Size operator

```
long size() const;
```

Returns the size of the container.

Complexity. Constant time.

Clear

```
void clear();
```

Erases the contents of the container, which becomes an empty container.

Complexity. Linear and logarithmic work and span, respectively.

Resize

```
void resize(long n, char val); // (1)
void resize(long n) {                       // (2)
```

```
  char val;
  resize(n, val);
}
```

Resizes the container to contain `n` items.

If the current size is greater than `n`, the container is reduced to its first `n` elements.

If the current size is less than `n`,

1. additional copies of `val` are appended
2. additional default-inserted elements are appended

Complexity. Let m be the size of the container just before and n just after the resize operation. Then, the work and span are linear and logarithmic in $\max(m, n)$, respectively.

Exchange operation

```
void swap(pstring& other);
```

Exchanges the contents of the container with those of `other`. Does not invoke any move, copy, or swap operations on individual items.

Complexity. Constant time.

Iterator begin

```
iterator begin() const;
const_iterator cbegin() const;
```

Returns an iterator to the first item of the container.

If the container is empty, the returned iterator will be equal to end().

Complexity. Constant time.

Iterator end

```
iterator end() const;
const_iterator cend() const;
```

Returns an iterator to the element following the last item of the container.

This element acts as a placeholder; attempting to access it results in undefined behavior.

Complexity. Constant time.

Append to string

```
pstring& operator+=(const pstring& str);
```

Extends the string by appending additional characters at the end of its current value.

Complexity. Linear and logarithmic in the size of the resulting string, respectively.

Iterator validity. Invalidates all iterators, if the size before the operation differs from the size after.

Concatenate strings

```
pstring operator+(const pstring& rhs);
```

Returns a newly constructed string object with its value being the concatenation of the characters in `this` followed by those of `rhs`.

Complexity. Linear and logarithmic in the size of the resulting string, respectively.

Get C string equivalent

```
const char* c_str();
```

Returns a pointer to an array that contains a null-terminated sequence of characters (i.e., a C-string) representing the current value of the string object.

This array includes the same sequence of characters that make up the value of the string object plus an additional terminating null-character (`'\0'`) at the end.

Complexity. Constant time.

Data-parallel operations

This section introduces a collection of data-parallel operations that are useful for processing over pctl containers.

Parallel-for loop

```
namespace pasl {
namespace pctl {

template <class Iter, class Body>
```

```
void parallel_for(Iter lo, Iter hi, Body body);

} }
```

The first of data-parallel operations we are going to consider is a looping construct that is useful for making changes to memory. For now, we are going to assume that each iterate of our loops take constant time. In the next section, we show how we can handle in an effective manner loop bodies that are not constant time.

Let us begin by considering example code. The following program uses a parallel-for loop to assign each position `i` in `xs` to the value `i+1`.

```
parray<long> xs = { 0, 0, 0, 0 };
parallel_for((long)0, xs.size(), [&] (long i) {
  xs[i] = i+1;
});

std::cout << "xs = " << xs << std::endl;
```

It's output is the following.

```
xs = { 1, 2, 3, 4 }
```

We can just as easily configure our parallel-for loop to iterate with respect to a specified range of iterator values. In this case, we are instead just incrementing the value in each cell of our parallel array.

```
parray<long> xs = { 0, 1, 2, 3 };
parallel_for(xs.begin(), xs.end(), [&] (long* p) {
  long& xs_i = *p;
  xs_i = xs_i + 1;
});

std::cout << "xs = " << xs << std::endl;
```

The output of this program is the same as that of the one just above.

By exchanging the type `parray` for `pchunkedseq` and the iterator type `long*` for `typename pchunkedseq<long>::iterator`, we can iterate over a parallel chunked sequence in a similar manner.

```
pchunkedseq<long> xs = { 0, 1, 2, 3 };
parallel_for(xs.begin(), xs.end(), [&] (typename pchunkedseq<long>::iterator p) {
  long& xs_i = *p;
  xs_i = xs_i + 1;
});

std::cout << "xs = " << xs << std::endl;
```

The output of this program is the same as that of the one just above. All of the examples presented here can be found in the source file parallelfor.cpp.

Non-constant-time loop bodies

```
namespace pasl {
namespace pctl {

template <class Iter, class Body, class Comp>
void parallel_for(Iter lo, Iter hi, Comp comp, Body body);

} }
```

When the loop body does not take constant time, that is, the loop body takes time in proportion to a known quantity, our code needs to provide a runction of that quantity to the looping construct in order for the loop to find an efficient schedule for the loop iterations. To see how this reporting is done, let us consider a program that computes the product of a dense matrix by a dense vector. Our matrix is `n` $\times$ `n` and is laid out in memory in row major format. In the following example, we call `dmdmult1` to compute the vector resulting from the multiplication of matrix `mtx` and vector `vec`.

```
parray<double> mtx = { 1.1, 2.1, 0.3, 5.8,
                       8.1, 9.3, 3.1, 3.2,
                       5.3, 3.5, 7.9, 2.3,
                       4.5, 5.5, 3.4, 4.5 };

parray<double> vec = { 4.3, 0.3, 2.1, 3.3 };

parray<double> result = dmdvmult1(mtx, vec);
std::cout << "result = " << result << std::endl;
```

The output of the program is the following.

```
result = { 25.13, 54.69, 48.02, 42.99 }
```

In the body of `dmdvmult1`, we need to compute the dot product of a given row by our vector `vec`. Let us just assume that we have such a function already. In particular, the `ddotprod` function takes the dimension value, a pointer to the start of a row in the matrix, a pointer to the vector. The function returns the corresponding dot product, taking time $O(\mathtt{n})$.

```
double ddotprod(long n, const double* row, const double* vec);
```

Now, we see that the body of our `dmdvmult1` function uses a parallel-for loop to fill each cell in the result vector. The complexity function `comp` describes the approximate cost of performing one loop iterate, which is the same as performing one dot product operation on a specified row.

```
parray<double> dmdvmult1(const parray<double>& mtx, const parray<double>& vec) {
  long n = vec.size();
  parray<double> result(n);
  auto comp = [&] (long i) {
    return n;
  };
  parallel_for(0L, n, comp, [&] (long i) {
    result[i] = ddotprod(n, mtx.cbegin()+(i*n), vec.begin());
  });
  return result;
}
```

The above code is perfectly fine for most purposes, because the granularity-control algorithm of pctl should be able to use the complexity function to schedule loop iterations efficiently. However, this particular type of parallel-for loop imposes scheduling-related overhead that is important to be aware of. In particular, inside the implementation of this particular parallel-for loop, there is an operation that is being called to precompute the cost of computing any subrange of the parallel-for loop iterations.

The `weights` operation

This operation is a function which takes a number `n` and a weight function `w` and returns a *weight table*.

```
template <class Weight>
parray<long> weights(long n, Weight weight);
```

The result returned by the call `weights(n, w)` is the sequence `[0, w(0), w(0)+w(1), w(0)+w(1)+w(2), ..., w(0)+...+w(n-1)]`. Notice that the size of the value returned by the `weights` function is always `n+1`. The work and span cost of this operation are linear and logarithmic in `n`, respectively.

Let us consider how the range-based parallel-for loop in `dmdvmult2` would call the weights function. Suppose that $\texttt{n} = 4$. Now, internally, the parallel-for loop is going to compute a weight table that looks the same as `w` in the code below.

```
parray<long> w = weights(4, comp);

std::cout << "w = " << w << std::endl;
```

The output is going to be as shown below.

```
w = { 0, 4, 8, 12, 16 }
```

For one more example, let us consider an application of the `weights` function where the given weight function is one that returns the value of its current position.

```
parray<long> w = weights(4, [&] (long i) {
  return i;
});

std::cout << "w = " << w << std::endl;
```

The output is the following.

```
w = { 0, 0, 1, 3, 6 }
```

Returning to our `dmdvmult2` example, we can now see that the work and span cost of calculating the weights table are linear and logarithmic in `n`, respectively. Since the total cost of the multiplication is $O(\mathtt{n}^2)$, the cost of calculating the weights table is relatively negligible and can therefore be disregarded in this case. Nevertheless, in other cases, we may want to avoid the cost of calculating the weights table. To do so, we need to consider the *range-based* parallel-for loop.

Range-based complexity functions for non-constant-time loop bodies

```
namespace pasl {
namespace pctl {
namespace range {

template <
  class Iter,
  class Body,
  class Comp_rng
>
void parallel_for(Iter lo,
                  Iter hi, Comp_rng comp_rng,
                  Body body);

} } }
```

The idea here is that, instead of reporting the cost of computing an individual iterate, we report the cost of a given range. The function `comp_rng` is going to calculate and return exactly this value. The code below shows one way that we can make this modification to our `dmdvmult2` function.

```
parray<double> dmdvmult2(const parray<double>& mtx, const parray<double>& vec) {
  long n = vec.size();
  parray<double> result(n);
  auto comp_rng = [&] (long lo, long hi) {
    return (hi - lo) * n;
  };
  range::parallel_for(0L, n, comp_rng, [&] (long i) {
    result[i] = ddotprod(n, mtx.cbegin()+(i*n), vec.begin());
```

```
  });
  return result;
}
```

The idea is that instead of reporting the cost of computing the dot product on an individual row, we instead report the cost of performing a sequence of dot products on a specified range. In particular, the `comp_rng` function calculates the cost of performing `hi-lo` dot products. In this way, we have bypassed having to calculate the weights table by providing such a range-based complexity function.

Sequential-alternative loop bodies

```
namespace pasl {
namespace pctl {
namespace range {

template <
  class Iter,
  class Body,
  class Comp_rng,
  class Seq_body_rng
>
void parallel_for(Iter lo,
                  Iter hi,
                  Comp_rng comp_rng,
                  Body body,
                  Seq_body_rng seq_body_rng);

} } }
```

Let us suppose that we have some highly optimized sequential algorithm that we wish to use along with our parallel-for loop. How can we inject such code into a parallel-for computation? For this purpose, we have another form of parallel-for loop that takes a sequential-body function as its last argument.

Now, we can see from the following example code how we can use the new parallel-for loop. Just after the parallel body, we see a sequential body, which consists of two ordinary, sequential for loops. When the scheduler determines that a loop range is too small to benefit from parallelism, the body that is applied is the sequential body.

```
parray<double> dmdvmult3(const parray<double>& mtx, const parray<double>& vec) {
  long n = vec.size();
  parray<double> result(n);
  auto comp_rng = [&] (long lo, long hi) {
    return (hi - lo) * n;
  };
```

```
  range::parallel_for(0L, n, comp_rng, [&] (long i) {
    result[i] = ddotprod(n, mtx.cbegin()+(i*n), vec.begin());
  }, [&] (long lo, long hi) {
    for (long i = lo; i < hi; i++) {
      double dotp = 0.0;
      for (long j = 0; j < n; j++) {
        dotp += mtx[i*n+j] * vec[j];
      }
      result[i] = dotp;
    }
  });
  return result;
}
```

At first glance, our optimization of injecting a sequential body might seem artificial, given that, for a large enough matrix, the sequential body may never get applied. The reason is that the work involved in processing just a single, large row may be plenty to warrant parallel execution. However, if we consider rectangular matrices, or sparce matrices, it should be clear that the sequential body can improve performance in certain cases.

To summarize, we started with a basic parallel-for loop, showing that the basic version is flexible enough to handle various indexing formats (e.g., integers, iterators of various kinds). We then saw that, when the body of the loop is not constant time, we need to define a complexity function and pass the complexity function to the parallel-for function. We then saw that we can potentially reduce the scheduling overhead by instead using a range-based version of parallel for, where we pass a range-based complexity function. Then, we saw that we can potentially optimize further by providing a sequential alternative body to the parallel for. Finally, all of the code examples presented in this section can be found in the file dmdvmult.cpp.

Template parameters

The following table describes the template parameters that are used by the different version of our parallel-for function.

Table 22: All template parameters used by various instance of the parallel-for loop.

Template parameter	Description
`Iter`	Type of the iterator to be used by the loop
`Body`	Loop body
`Seq_body_rng`	Sequentialized version of the body

Template parameter	Description
`Comp`	Complexity function for a specified iteration
`Comp_rng`	Complexity function for a specified range of iterations

Loop iterator

```
class Iter;
```

At a minimum, any value of type `Iter` must support the following operations. Let `a` and `b` denote values of type `Iter` and `n` a value of type `long`. Then, we need the subtraction operation `a-b`, the comparison operation `a!=b`, the addition-by-a-number-operation `a+n`, and the increment operation `a++`.

As such, the concept of the `Iter` class bears resemblance to the concept of the random-access iterator. The main difference between the two is that, with the random-access iterator, an iterable value necessarily has the ability to dereference, whereas with our `Iter` class this feature is not used by the parallel-for loop and therefore not required.

Loop body

```
class Body;
```

The loop body class must provide a call operator of the following type. The iterator parameter is to receive the value of the current iterate.

```
void operator()(Iter it);
```

Sequentialized loop body

```
class Seq_body_rng;
```

The sequential loop body class must provide a call operator of the following type.

```
void operator()(Iter lo, Iter hi);
```

This method is called by the scheduler when a given range of loop iterates is determined to be too small to benefit from parallel execution. This range to be executed sequentially is the right-open range [`lo`, `hi`).

Complexity function

```
class Comp;
```

The complexity-function class must provide a call operator of the following type.

```
long operator()(Iter it);
```

This method is called by the scheduler to determine the cost of executing a given iterate, specified by `it`. The return value should be a positive number that represents an approximate (i.e., asymptotic) cost required to execute the loop body at the iterate `it`.

Range-based compelxity function

```
class Comp_rng;
```

The range-based complexity-function class must provide a call operator of the following type.

```
long operator()(Iter lo, Iter hi);
```

This method is called by the scheduler to determine the cost of a *range* of loop iterates. The range is specified by the right-open range [`lo`, `hi`). Like the above function, the return value should be a positive number that represents an approximate (i.e., asymptotic) cost required to execute specified range.

Reductions and scans

Parallel-for loops give us the ability to process in parallel over a specified range of iterates. However, often, we want to, say, take the sum of a given sequence of numbers, or perhaps, find the lowest point in a given set of points in two-dimensional space. The problem is that the parallel-for loop is the wrong tool for this job, because it is not safe to use a parallel-for loop to accumulate a value in a shared memory cell. We need another mechanism if we want to program such accumulation patterns. In general, when we have a collection of values that we want to combine in a certain fashion, we can often obtain an efficient and clean solution by using a *reduction*, which is an operation that combines the items in a specified range in a certain fashion.

The mechanism that a reduction uses to combine a given collection of items is a mathematical object called a *monoid*. The reason that monoids are useful is because, if the problem can be stated in terms of a monoid, then a monoid provides a simple, yet sufficient condition to find a corresponding reduction that delivers an efficient parallel solution. In other words, a monoid is a bit like a design pattern for programming parallel algorithms. Recall that a monoid is an algebraic structure that consists of a set T, an associative binary operation $\oplus$ and an identity element $\mathbf{I}$. That is, $(T, \oplus, \mathbf{I})$ is a monoid if:

- $\oplus$ is associative: for every x, y and z in T, $x \oplus (y \oplus z) = (x \oplus y) \oplus z$.
- $\mathbf{I}$ is the identity for $\oplus$: for every x in T, $x \oplus \mathbf{I} = \mathbf{I} \oplus x$.

Examples of monoids include the following:

- T = the set of all integers; $\oplus$ = addition; $\mathbf{I} = 0$

- T = the set of 32-bit unsigned integers; $\oplus$ = addition modulo 2^{32}; $\mathbf{I} = 0$
- T = the set of all strings; $\oplus$ = concatenation; $\mathbf{I}$ = the empty string
- T = the set of 32-bit signed integers; $\oplus$ = `std::max`; $\mathbf{I} = -2^{32} + 1$

Let us now see how we can encode a reduction in C++.

Basic reduction

```
namespace pasl {
namespace pctl {

template <class Iter, class Item, class Combine>
Item reduce(Iter lo, Iter hi, Item id, Combine combine);

} }
```

The most basic reduction pattern that is provided by pctl is the one above. This template function applies a monoid, encoded by the identity element `id` and the associative combining operator `combine`, to return the value computed by combining together the items contained by the given right-open range [`lo`, `hi`). For example, suppose that we are given as input an array of numbers and our task is to find the largest number using linear work and logarithmic span. We can solve this problem by using our basic reduction, as shown below.

```
int max(const parray<int>& xs) {
  int id = std::numeric_limits<int>::lowest();
  return reduce(xs.cbegin(), xs.cend(), id, [&] (int x, int y) {
    return std::max(x, y);
  });
}
```

Let us pause for a moment to consider what is the value returned by the `max` function when it is passed the empty array (i.e., `max({ })`. In this case, the value is going to be the smallest

As we are going to see later, the work and span cost of this operation are linear and logarithmic in the size of the input array, implying a good parallel solution to this particular problem.

It is worth noting that reduction can be applied to any container type that provides a random-access iterator. For example, we could obtain a similarly valid and efficient parallel solution to find the max of the items in a chunked sequence, for example, by simply replacing the type `parray` in the code above by the type `pchunkedseq`.

Basic reduction with non-constant-time combining operators

```
namespace pasl {
namespace pctl {

template <
  class Iter,
  class Item,
  class Weight,
  class Combine
>
Item reduce(Iter lo,
            Iter hi,
            Item id,
            Weight weight,
            Combine combine);

} }
```

Recall that, for parallel-for loops, we had to take special measures in order to properly encode loops with non-constant-time loop bodies. The situation is similar for reductions: if the associative combining operator is not constant time, then we need to provide an extra function to report the costs. However, the way that we are going to report is a little different that the way we reported to the parallel-for loop.

What we want to report is the *weight* of each item in the input sequence. The idea is that the cost of combining any two items in the input sequence using the given associative combining operator is the sum of the weights of the two items. Let us consider an example of this pattern. The code below takes as input an array of arrays of numbers and returns the largest number contained by the subarrays. Now, observe that the cost of performing our `combine` operator to any two subarrays `xs1` and `xs2` is the value `xs1.size() + xs2.size()`. It should be clear that our reduce operation can calculate these intermediate costs from the `weight` function, which we pass to our reduction. Just like the weight-based parallel-for loop, the reduction operation calculates a table containing the prefix sums of the weights of the items.

```
int max0(const parray<parray<int>>& xss) {
  parray<int> id = { std::numeric_limits<int>::lowest() };
  auto weight = [&] (const parray<int>& xs) {
    return xs.size();
  };
  auto combine = [&] (const parray<int>& xs1,
                      const parray<int>& xs2) {
    parray<int> r = { std::max(max(xs1), max(xs2)) };
    return r;
```

```
  };
  parray<int> a =
    reduce(xss.cbegin(), xss.cend(), id, weight, combine);
  return a[0];
}
```

When the `max0` function returns, the result is just one number that is our maximum value. It is therefore unfortunate that our combining operator has to pay the cost to package the current maximum value in the array `r`. The abstraction boundaries, in particular, the type of the `reduce` function here leaves us no choice, however. Later, we are going to see that, by generalizing our `reduce` function a little, we can sidestep this issue.

The example codes shown in this section can be found in max.hpp and max.cpp.

Basic scan

```
namespace pasl {
namespace pctl {

template <
  class Iter,
  class Item,
  class Combine
>
parray<Item> scan(Iter lo,
                  Iter hi,
                  Item id,
                  Combine combine,
                  scan_type st);

} }
```

A *scan* is an operation that computes the running totals of a given sequence of values of a specified type. Just like with a reduction, a scan computes its running totals with respect to a given monoid. So, accordingliy, the signature of our scan function shown above looks a lot like the signature of our reduce function.

However, there are a couple differences. First, instead of returning a single result value, the scan function returns an array (of length `hi - lo`) of the running totals of the input sequence. Second, the function takes the argument `st` to determine whether the scan is inclusive or exclusive and whether the results are to be calculated from the front to the back of the input sequence or from back to front.

```
namespace pasl {
```

```
namespace pctl {

using scan_type = enum {
  forward_inclusive_scan,
  forward_exclusive_scan,
  backward_inclusive_scan,
  backward_exclusive_scan
};

} }
```

Before we define scan formally, let us first consider some example programs. For our running example, we are going to pick the same monoid that we used in our running example for the reduce function: the set of 32-bit signed integers, along with the `std::max` combining operator and the identity $-2^{32} + 1$.

```
auto combine = [&] (int x, int y) {
  return std::max(x, y);
};
int id = std::numeric_limits<int>::lowest();  // == -2^32+1 == -2147483648
```

We are going to use the following array as the input sequence.

```
parray<int> xs = { 1, 3, 9, 0, 33, 1, 1 };
```

The following code prints the result of the forward-exclusive scan of the above sequence.

```
parray<int> fe =
  scan(xs.cbegin(), xs.cend(), id, combine, forward_exclusive_scan);
std::cout << "fe\t= " << fe << std::endl;
```

The output is shown below. Notice that the first value is the identity element, `id` and that the running totals are computed starting from the beginning to the end of the input sequence. In general, the first value of the result of any exclusive scan is the identity element. Moreover, the total of the entire input sequence is not included in the result array.

```
fe  = { -2147483648, 1, 3, 9, 9, 33, 33 }
```

In general, the result of a forward-exclusive reduction is the sequence $[(\mathbf{I}), (\mathbf{I} \oplus xs_0), (\mathbf{I} \oplus xs_0 \oplus xs_1), \ldots, (\mathbf{I} \oplus xs_0 \oplus \ldots \oplus xs_{n-2})]$, where the input sequence is $xs = [xs_0, \ldots, xs_{n-1}]$. Now, let us consider the forward-inclusive scan.

```
parray<int> fi =
  scan(xs.cbegin(), xs.cend(), id, combine, forward_inclusive_scan);
std::cout << "fi\t= " << fi << std::endl;
```

In the forward-inclusive scan, the first value in the result array is always the first value in the input sequence and the last value is the total of the entire input sequence.

```
fi  = { 1, 3, 9, 9, 33, 33, 33 }
```

The below example shows the backward-oriented versions of the scan operator.

```
parray<int> be =
  scan(xs.cbegin(), xs.cend(), id, combine, backward_exclusive_scan);
std::cout << "be\t= " << be << std::endl;

parray<int> bi =
  scan(xs.cbegin(), xs.cend(), id, combine, backward_inclusive_scan);
std::cout << "bi\t= " << bi << std::endl;
```

Output:

```
be  = { 33, 33, 33, 33, 1, 1, -2147483648 }
bi  = { 33, 33, 33, 33, 33, 1, 1 }
```

The uses of the scan function that we have considered thus far use a constant-time combining operator. For such applications, the work and span complexity of an application is the same as the work and span complexity of the corresponding reduce operation. In specific, our applications of scan shown above take linear work and logarithmic span in the size of the input sequence, and so does our `max` function, which uses reduce in a similar way. It turns out that, in general, given the same inputs (disragarding, of course, the scan-type argument, which has no effect with respect to asymptotic running time), reduce and scan have the same work and span complexity.

Basic scan with a non-constant-time combining operator

```
namespace pasl {
namespace pctl {

template <
  class Iter,
  class Item,
  class Weight,
  class Combine
>
parray<Item> scan(Iter lo,
                  Iter hi,
                  Item id,
                  Weight weight,
                  Combine combine,
                  scan_type st)

} }
```

Just as was the case with reduce, our scan operator has to handle non-constant-time associative combining operators. To this end, pctl provides the above scan function, which now takes the corresponding weight function. An application of this scan function looks just like with the corresponding reduce function.

Template parameters

Table 23: Template parameters for basic reduce and scan operations.

Template parameter	Description
`Iter`	Type of the iterator to be used to access items in the input container
`Item`	Type of the items in the input container
`Combine`	Associative combining operator
`Weight`	Weight function (optional)

Item iterator

```
class Iter;
```

An instance of this class must be an implementation of the random-access iterator.

An iterator value of this type points to a value from the input stream (i.e., a value of type `Item`).

Item

```
class Item;
```

Type of the items to be processed by the reduction.

Associative combining operator

```
class Combine;
```

The combining operator is a C++ functor that takes two items and returns a single item. The call operator for the `Combine` class should have the following type.

```
Item operator()(const Item& x, const Item& y);
```

The behavior of the reduction is well defined only if the combining operator is *associative*.

Associativity. Let `f` be an object of type `Combine`. The operator `f` is associative if, for any `x`, `y`, and `z` that are values of type `Item`, the following equality holds:

```
f(x, f(y, z)) == f(f(x, y), z)
```

Weight function

```
class Weight;
```

The weight function is a C++ functor that takes a single item and returns a non-negative "weight value" describing the size of the item. The call operator for the weight function should have the following type.

```
long operator()(const Item& x);
```

Complexity

There are two cases to consider for any reduction `reduce`(lo, hi, id, f) (or, correspondingly, any scan `scan`(lo, hi, id, f, st) for any scan type st):

1. ***Constant-time associative combining operator.*** The amount of work performed by the reduction is $O(hi - lo)$ and the span is $O(\log(hi - lo))$.
2. ***Non-constant-time associative combining operator.*** We define $\mathcal{R}$ to be the set of all function applications $f(x, y)$ that are performed in the reduction tree. Then,
   - The work performed by the reduction is $O(n + \sum_{f(x,y) \in \mathcal{R}(f,id,lo,hi)} W(f(x, y)))$.
   - The span of the reduction is $O(\log n \max_{f(x,y) \in \mathcal{R}(f,id,lo,hi)} S(f(x, y)))$.

Under certain conditions, we can use the following lemma to deduce a more precise bound on the amount of work performed by the reduction.

Lemma (Work efficiency). For any associative combining operator f and weight function w, if for any x, y,

- $w(f(x, y)) \leq w(x) + w(y)$, and
- $W \leq c(w(x) + w(y))$, for some constant c,

where W denotes the amount of work performed by the call $f(x, y)$, then the amount of work performed by the reduction is $O(\log(hi - lo) \sum_{lo \leq it < hi} (1 + w(*it)))$.

Example: analysis of the complexity of a non-constant time combining operator. For this example, let us analyze the `max0` function, which is defined in a previous section. We will begin by analyzing the work. To start, we need to determine whether the combining operator of the reduction over `xss` is constant-time or not. This combining operator is not because the combining operator calls the `max` function (twice, in fact). The first call is applied to the array `xs` and the second to `ys`. The total work performed by these two calls is

linear in $|\texttt{xs}| + |\texttt{ys}|$. Therefore, by applying the work-lemma shown above, we get that the total work performed by this reduction is $O(\log |\texttt{xss}| \max_{\texttt{xs} \in \texttt{xss}} |xs||)$. The span is simpler to analyze. By applying our span rule for reduce, we get that the span for the reduction is $O(\log |xss| \max_{\texttt{xs} \in \texttt{xss}} \log |xs|)$.

Advanced reductions and scans

Although quite general already, the reduce and scan functions that we have considered thus far are not always sufficiently general. For example, the basic forms of reduce and scan require that the type of the result value (or values in the case of scan) is the same as the type of the values in the input sequence. As we saw in a previous section, this requirement made the code of the `max0` function look rather awkward and perhaps unnecessarily inefficient. Fortunately, in pctl, such issues can be readily addressed by using more advanced forms of reduce and scan.

In this section, we are going to examine in detail which problems (or, more precisely, problem patterns) that can be solved more efficiently and concisely by using more general forms of reduce and scan. Because the number of possible variations of reduce and scan is quite large, we are going to use a layered design, whereby, for each generalization we introduce, we introduce one new abstraction layer on top of the previous one. The following table summarizes the layers that we define and what each successive layer adds on top of its predecessor layer. The lowest level, namely level 0, is the level corresponding to our basic reduce and scan functions from the beginning of this section.

Table 24: Abstraction layers for reduce and scan that are provided by pctl.

Abstraction layer	Description
Level 0 (basic reduce and scan)	Apply a specified monoid to combine the items of a range in memory that is specified by a pair of iterator pointer values
Level 1	Introduces a lift operator that allows the client to inline into the reduce a specified map operation
Level 2	Introduces an operator that provides a sequentialized alternative for the lift operator
Level 3	Introduces a “mergeable output” type that enables destination-passing style reduction

Abstraction layer	Description
Level 4	Introduces a “splittlable input” type that replaces the iterator pointer values

Level 1

```
namespace pasl {
namespace pctl {
namespace level1 {

template <
  class Iter,
  class Result,
  class Combine,
  class Lift
>
Result reduce(Iter lo,
              Iter hi,
              Result id,
              Combine combine,
              Lift lift);

template <
  class Iter,
  class Result,
  class Combine,
  class Lift_comp,
  class Lift
>
Result reduce(Iter lo,
              Iter hi,
              Result id,
              Combine combine,
              Lift_comp lift_comp,
              Lift lift);


template <
  class Iter,
  class Result,
  class Combine,
  class Lift
>
```

```
parray<Result> scan(Iter lo,
                    Iter hi,
                    Result id,
                    Combine combine,
                    Lift lift,
                    scan_type st);

template <
  class Iter,
  class Result,
  class Combine,
  class Lift_comp,
  class Lift
>
parray<Result> scan(Iter lo,
                    Iter hi,
                    Result id,
                    Combine combine,
                    Lift_comp lift_comp,
                    Lift lift,
                    scan_type st);

} } }
```

To motivate level 1, let us return to our running example and, in particular, the `max0` function from a previous section. When we examine `max0`, we see that our identity value is an array, our combine operator takes two arrays and returns an array, and our result value is itself an array. In other words, the monoid that we use for `max0` is one where the input sequence is the set of all arrays of numbers, and the identity and combining function follow accordingly. As such, to get the final result value, at the end of the function, we have to extract the first value from the result array. Why do we need to use these intermediate arrays? The only reason is that the basic reduce function requires that the type of the input value (an array of numbers) be the same as the type of the result value (an array of numbers).

The level-1 reduce function allows us to bypass this limitation, provided that we define and pass to the level-1 reduce function a *lift* function of our choice. To see what it looks like, let us consider how we solve the same problem as before, but this time using level-1 reduce. In the `max1` function below, we see first that our identity element and associative combining operator are specified for base values (i.e., `int`s) rather than arrays of `int`s. Furthermore, we see that our lift function is a function that takes (a reference to) an array and returns the maximum value of the given array, using the same `max` function that we defined in a previous section. Now, it should be clear what is the purpose of the lift function: it describes how to solve the same problem, but specifically for an individual item in the input.

```
int max1(const parray<parray<int>>& xss) {
  auto lo = xss.cbegin();
  auto hi = xss.cend();
  int id = std::numeric_limits<int>::lowest();
  auto combine = [&] (int x, int y) {
    return std::max(x, y);
  };
  auto lift_comp = [&] (const parray<int>& xs) {
    return xs.size();
  };
  auto lift = [&] (const parray<int>& xs) {
    return max(xs);
  };
  return level1::reduce(lo, hi, id, combine, lift_comp, lift);
}
```

As usual, because our lift function is not a constant-time function, we need to report the cost of the lift function. To this end, in the code above, we use the `lift_comp` function which returns the cost of the corresponding application of the `lift` function. The cost that we defined is, in particular, equal to the size of the array, since the `max` function takes linear time.

In general, our level 1 reduce (and scan) now recognize two types of values:

1. the type `Item`, which is the type of the items stored in the input sequence (i.e., the type of values pointed at by an iterator of type `Iter`)
2. the type `Result`, which is the type of the result value (or values in the case of scan) that are returned by the reduce (or scan).

In terms of types, the reduce function converts a value of type `Item` to a corresponding value of type `Result`. In the example above, our `lift` function converts from a value of type `Item = parray<int>` to a value of type `int`.

Index-passing reduce and scan

```
namespace pasl {
namespace pctl {
namespace level1 {

template <
  class Iter,
  class Result,
  class Combine,
  class Lift_idx
>
Result reducei(Iter lo,
               Iter hi,
```

```
              Result id,
              Combine combine,
              Lift_idx lift_idx);

template <
  class Iter,
  class Result,
  class Combine,
  class Lift_comp_idx,
  class Lift_idx
>
Result reducei(Iter lo,
              Iter hi,
              Result id,
              Combine combine,
              Lift_comp_idx lift_comp_idx,
              Lift_idx lift_idx);

template <
  class Iter,
  class Result,
  class Combine,
  class Lift_idx
>
parray<Result> scani(Iter lo,
                     Iter hi,
                     Result id,
                     Combine combine,
                     Lift_idx lift_idx,
                     scan_type st);

template <
  class Iter,
  class Result,
  class Combine,
  class Lift_comp_idx,
  class Lift_idx
>
parray<Result> scani(Iter lo,
                     Iter hi,
                     Result id,
                     Combine combine,
                     Lift_comp_idx lift_comp_idx,
                     Lift_idx lift_idx,
                     scan_type st);
```

```
} } }
```

Sometimes, it is useful for reduce and scan to pass to their lift function an extra argument: the corresponding position of the item in the input sequence. For this reason, pctl provides for each of the level 1 functions a corresponding *index-passing* version. For instance, the `reducei` function below now takes a `lift_idx` function instead of the usual a `lift` function. This new, index-passing version of the lift function itself takes an additional position argument: for item xs_i, the lift function is now applied by `reducei` to each element as before, but along with the position of the item (i.e., `lift_idx`(i, xs_i)).

Template parameters

Table 25: Template parameters that are introduced in level 1.

Template parameter	Description
`Result`	Type of the result value to be returned by the reduction
`Lift`	Lifting operator
`Lift_idx`	Index-passing lifting operator
`Combine`	Associative combining operator
`Lift_comp`	Complexity function associated with the lift funciton
`Lift_comp_idx`	Index-passing lift complexity function

Result

```
class Result
```

Type of the result value to be returned by the reduction.

This class must provide a default (i.e., zero-arity) constructor.

Lift

```
class Lift;
```

The lift operator is a C++ functor that takes an iterator and returns a value of type `Result`. The call operator for the `Lift` class should have the following type.

```
Result operator()(const Item& x);
```

Index-passing lift

```
class Lift_idx;
```

The lift operator is a C++ functor that takes an index and a corresponding iterator and returns a value of type `Result`. The call operator for the `Lift` class should have the following type.

```
Result operator()(long pos, const Item& x);
```

The value passed in the `pos` parameter is the index corresponding to the position of item `x`.

Associative combining operator

```
class Combine;
```

Now, the type of our associative combining operator has changed from what it is in level 0. In particular, the values that are being passed and returned are values of type `Result`.

```
Result operator()(const Result& x, const Result& y);
```

Complexity function for lift

```
class Lift_comp;
```

The lift-complexity function is a C++ functor that takes a reference on an item and returns a non-negative number of type `long`. The `Lift_comp` class should provide a call operator of the following type.

```
long operator()(const Item& x);
```

Index-passing lift-complexity function

```
class Lift_comp_idx;
```

The lift-complexity function is a C++ functor that takes an index and an reference on an item and returns a non-negative number of type `long`. The `Lift_comp_idx` class should provide a call operator of the following type.

```
long operator()(long pos, const Item& x);
```

Complexity

Let us now consider the work and span complexity of a level-1 reduction `reduce`(lo, hi, id, f, l) (or, correspondingly, any scan `scan`(lo, hi, id, f, l, st) for any scan type st). Compared to the previous level, in level 1, we now need to account for the costs associated with applications of the lift operator. We define $\mathcal{L}$ to be the set of all function applications of $l(a)$ that are performed in the leaves of the level-1 reduction tree and $\mathcal{R}$ to be the set of all function applications $f(x, y)$ in the interior nodes of the level-1 reduction tree. There are two cases with respect to the associative combining operator f:

1. ***Constant-time associative combining operator.*** There are two cases with respect to the lifting operator l.

a. ***Constant-time lift operator.*** The amount of work performed by the reduction is $O(hi - lo)$ and the span is $O(\log(hi - lo))$.

b. ***Non-constant-time lift operator.*** The total amount of work performed by the reduction is $O((hi - lo) + \sum_{l(a) \in \mathcal{L}(lo,hi,id,f,l)} W(l(a)))$ and the span is $O(\log(hi - lo) + \max_{l(a) \in \mathcal{L}(lo,hi,id,f,l)} W(l(a)))$.

2. ***Non-constant-time associative combining operator.*** Then,

   - The work performed by the level-1 reduction is $O(n + \sum_{f(x,y) \in \mathcal{R}(lo,hi,id,f,l)} W(f(x,y)) + \sum_{l(a) \in \mathcal{L}(lo,hi,id,f,l)} W(l(a)))$.
   - The span of the reduction is $O(\log n \max_{f(x,y) \in \mathcal{R}(lo,hi,id,f,l)} S(f(x,y)) + \max_{l(a) \in \mathcal{L}(lo,hi,id,f,l)} S(l(a)))$.

Under certain conditions, we can use the following lemma to deduce a more precise bound on the amount of work performed by the level-1 reduction.

Lemma (Work efficiency). For any associative combining operator f and weight function w, if for any a

- $w(l(a)) \leq w(a)$, and
- $W_l \leq c(\log w(a) \cdot w(a))$, for some constant c.

where W_l denotes the amount of work performed by the call $l(a)$, and for any x, y,

- $w(f(x,y)) \leq w(x) + w(y)$, and
- $W_f \leq c(w(x) + w(y))$, for some constant c,

where W_f denotes the amount of work performed by the call $f(x,y)$, then the amount of work performed by the reduction is $O(\log(hi - lo) \sum_{lo \leq it < hi} (1 + w(*it)))$.

Level 2

```
namespace pasl {
namespace pctl {
namespace level2 {

template <
  class Iter,
  class Result,
  class Combine,
  class Lift_comp_rng,
  class Lift_idx,
  class Seq_reduce_rng
>
Result reduce(Iter lo,
              Iter hi,
```

```
              Result id,
              Combine combine,
              Lift_comp_rng lift_comp_rng,
              Lift_idx lift_idx,
              Seq_reduce_rng seq_reduce_rng);

template <
  class Iter,
  class Result,
  class Combine,
  class Lift_comp_rng,
  class Lift_idx,
  class Seq_reduce_rng
>
parray<Result> scan(Iter lo,
                    Iter hi,
                    Result id,
                    Combine combine,
                    Lift_comp_rng lift_comp_rng,
                    Lift_idx lift_idx,
                    Seq_reduce_rng seq_reduce_rng,
                    scan_type st);

} } }
```

In the section on parallel for loops, we saw that we can combine two algorithms, one parallel and one sequential, to make a hybrid algorithm that combines the best features of both algorithms. We rely on the underlying granularity-control algorithm of pctl to switch intelligently between the parallel and sequential algorithms as the program runs.

Now, we are going to see how we can apply a similar technique to reduction. Let us return to the problem of finding the maximum value in an array of arrays of numbers. The solution we are going to consider is the one given by the `max2` function below.

```
int max2(const parray<parray<int>>& xss) {
  using const_iterator = typename parray<parray<int>>::const_iterator;
  const_iterator lo = xss.cbegin();
  const_iterator hi = xss.cend();
  int id = std::numeric_limits<int>::lowest();
  parray<long> w = weights(xss.size(), [&] (const parray<int>& xs) {
    return xs.size();
  });
  auto combine = [&] (int x, int y) {
    return std::max(x, y);
  };
```

```
  const_iterator b = lo;
  auto lift_comp_rng = [&] (const_iterator lo, const_iterator hi) {
    return w[hi - b] - w[lo - b];
  };
  auto lift_idx = [&] (int, const parray<int>& xs) {
    return max(xs);
  };
  auto seq_reduce_rng = [&] (const_iterator lo, const_iterator hi) {
    int m = id;
    for (const_iterator i = lo; i != hi; i++) {
      for (auto j = i->cbegin(); j != i->cend(); j++) {
        m = std::max(m, *j);
      }
    }
    return m;
  };
  return level2::reduce(lo, hi, id, combine, lift_comp_rng,
                                     lift_idx, seq_reduce_rng);
}
```

Compared to our `max1` function, we can see two differences:

1. The complexity function for lift, namely `lift_comp_rng`, calculates the cost for a given range of input values rather than for a given individual value. The cost is precomputed by the `weights` function, which is described in a previous section.
2. The last argument to the reduce function is the sequential alternative body, namely `seq_reduce_rng`.

To summarize, our level-1 solution, namely `max1`, had a parallel solution that was expressed by nested instances of our reduce functions. In this section, we saw how to use the level-2 reduce function to combine parallel and sequential solutions to make a hybrid solution.

Template parameters

Table 26: Template parameters that are introduced in level 2.

Template parameter	Description
`Seq_reduce_rng`	Sequential alternative body for the reduce operation
`Lift_comp_rng`	Range-based lift complexity function
`Seq_scan_rng_dst`	Sequential alternative body for the scan operation

Sequential alternative body for the lifting operator

```
class Seq_reduce_rng;
```

The sequential-reduce function is a C++ functor that takes a pair of iterators and returns a result value. The `Seq_reduce_rng` class should provide a call operator with the following type.

```
Result operator()(Iter lo, Iter hi);
```

Range-based lift-complexity function

```
class Lift_comp_rng;
```

The range-based lift-complexity function is a C++ functor that takes a pair of iterators and returns a non-negative number. The value returned is a value to account for the amount of work to be performed to apply the lift function to the items in the right-open range `[lo, hi)` of the input sequence.

```
long operator()(Iter lo, Iter hi);
```

Sequential alternative body for the scan operation

```
class Seq_scan_rng_dst;
```

The sequential-scan function is a C++ functor that takes a pair of iterators, namely `lo` and `hi`, and writes its result to a range in memory that is pointed to by `dst_lo`. The `Seq_scan_rng_dst` class should provide a call operator with the following type.

```
Result operator()(Iter lo, Iter hi, typename parray<Result>::iterator dst_lo);
```

Level 3

```
namespace pasl {
namespace pctl {
namespace level3 {

template <
  class Input_iter,
  class Output,
  class Result,
  class Lift_comp_rng,
  class Lift_idx_dst,
  class Seq_reduce_rng_dst
>
void reduce(Input_iter lo,
            Input_iter hi,
            Output out,
            Result id,
            Result& dst,
```

```
            Lift_comp_rng lift_comp_rng,
            Lift_idx_dst lift_idx_dst,
            Seq_reduce_rng_dst seq_reduce_rng_dst);

template <
  class Input_iter,
  class Output,
  class Result,
  class Output_iter,
  class Lift_comp_rng,
  class Lift_idx_dst,
  class Seq_scan_rng_dst
>
void scan(Input_iter lo,
          Input_iter hi,
          Output out,
          Result& id,
          Output_iter outs_lo,
          Lift_comp_rng lift_comp_rng,
          Lift_idx_dst lift_idx_dst,
          Seq_scan_rng_dst seq_scan_rng_dst,
          scan_type st);

} } }
```

In this section, we are going to introduce the destination-passing-style interface for reduction and scan. Relative to the previous one, this level introduces one new concept. An *output descriptor* is a class that describes how objects of type `Result` are to be initialized, combined, and copied. Let us see how to use an output descriptor by continuing with our running example.

```
int max3(const parray<parray<int>>& xss) {
  using const_iterator = typename parray<parray<int>>::const_iterator;
  const_iterator lo = xss.cbegin();
  const_iterator hi = xss.cend();
  int id = std::numeric_limits<int>::lowest();
  parray<long> w = weights(xss.size(), [&] (const parray<int>& xs) {
    return xs.size();
  });
  auto combine = [&] (int x, int y) {
    return std::max(x, y);
  };
  using output_type = level3::cell_output<int, decltype(combine)>;
  output_type out(id, combine);
  const_iterator b = lo;
  auto lift_comp_rng = [&] (const_iterator lo, const_iterator hi) {
    return w[hi - b] - w[lo - b];
```

```
  };
  auto lift_idx_dst = [&] (int, const parray<int>& xs, int& result) {
    result = max(xs);
  };
  auto seq_reduce_rng = [&] (const_iterator lo, const_iterator hi, int& result) {
    int m = id;
    for (const_iterator i = lo; i != hi; i++) {
      for (auto j = i->cbegin(); j != i->cend(); j++) {
        m = std::max(m, *j);
      }
    }
    result = m;
  };
  int result;
  level3::reduce(lo, hi, out, id, result, lift_comp_rng,
                 lift_idx_dst, seq_reduce_rng);
  return result;
}
```

The only change in `max3` relative to `max2` is the use of the output descriptor `cell_output`. This output descriptor is the simplest output descriptor: all it does is apply the given combining operator directly to its arguments, writing the result into a destination cell. The implementation of the cell output is shown below.

```
namespace pasl {
namespace pctl {
namespace level3 {

template <class Result, class Combine>
class cell_output {
public:

  using result_type = Result;
  using array_type = parray<result_type>;
  using const_iterator = typename array_type::const_iterator;

  result_type id;
  Combine combine;

  cell_output(result_type id, Combine combine)
  : id(id), combine(combine) { }

  cell_output(const cell_output& other)
  : id(other.id), combine(other.combine) { }

  void init(result_type& dst) const {
```

```
    dst = id;
  }

  void copy(const result_type& src, result_type& dst) const {
    dst = src;
  }

  void merge(const result_type& src, result_type& dst) const {
    dst = combine(dst, src);
  }

  void merge(const_iterator lo, const_iterator hi, result_type& dst) const {
    dst = id;
    for (const_iterator it = lo; it != hi; it++) {
      dst = combine(*it, dst);
    }
  }

};

} } }
```

The constructor takes the identity element and associative combining operator and stores them in member fields. The `init` method writes the identity element into the given destination cell. The `copy` method simply copies the value in `src` to the `dst` cell. There are two `merge` methods: the first one applies the combining operator, leaving the result in the `dst` cell. The second uses the combining operator to accumulate a result value from a given range of input items, leaving the result in the `dst` cell.

Because it assumes that objects of type `Result` can be copied efficiently, the cell output descriptor does not demonstrate the benefit of the output-descriptor mechanism. However, the benefit is clear when the `Result` objects are container objects, such as `pchunkedseq` objects, which require linear work to be copied.

```
template <class Chunked_sequence>
class chunkedseq_output {
public:

  using result_type = Chunked_sequence;
  using const_iterator = const result_type*;

  result_type id;

  chunkedseq_output() { }

  void init(result_type& dst) const {
```

```
  }

  void copy(const result_type& src, result_type& dst) const {
    dst = src;
  }

  void merge(result_type& src, result_type& dst) const {
    dst.concat(src);
  }

  void merge(const_iterator lo, const_iterator hi, result_type& dst) const {
    dst = id;
    for (const_iterator it = lo; it != hi; it++) {
      merge(*it, dst);
    }
  }

};
```

The first `merge` method concatenates the `src` and `dst` containers, leaving the result in `dst` and taking logarithmic work to complete. As such, the chunkedseq output descriptor can merge results taking logarithmic work in the size of the chunked sequence, thereby avoiding costly linear-work copy merges that would be imposed if we were to instead use the cell output descriptor for the same task. The second `merge` method simply combines all items in the range using the first `merge` method.

Template parameters

Table 27: Template parameters that are introduced in level 3.

Template parameter	Description
`Input_iter`	Type of an iterator for input values
`Output_iter`	Type of an iterator for output values
`Output`	Type of the object to manage the output of the reduction
`Lift_idx_dst`	Lift function in destination-passing style
`Seq_reduce_rng_dst`	Sequential reduce function in destination-passing style

Input iterator

```
class Input_iter;
```

An instance of this class must be an implementation of the random-access iterator.

An iterator value of this type points to a value from the input stream.

Output iterator

```
class Output_iter;
```

An instance of this class must be an implementation of the random-access iterator.

An iterator value of this type points to a value from the output stream.

Output

```
class Output;
```

Type of the object to receive the output of the reduction.

Table 28: Constructors that are required for the `Output` class.

Constructor	Description
copy constructor	Copy constructor

Table 29: Public methods that are required for the `Output` class.

Public method	Description
`init`	Initialize given result object
`copy`	Copy the contents of a specified object to a specified cell
`merge`	Merge result objects

Copy constructor

```
Output(const Output& other);
```

Copy constructor.

Result initializer

```
void init(Result& dst) const;
```

Initialize the contents of the result object referenced by `dst`.

Copy

```
void copy(const Result& src, Result& dst) const;
```

Copy the contents of `src` to `dst`.

Merge

```
void merge(Result& src, Result& dst) const;                    // (1)
void merge(Output_iter lo, Output_iter hi, Result& dst) const; // (2)
```

(1) Merge the contents of `src` and `dst`, leaving the result in `dst`.

(2) Merge the contents of the cells in the right-open range `[lo, hi)`, leaving the result in `dst`.

Destination-passing-style lift

```
class Lift_idx_dst;
```

The destination-passing-style lift function is a C++ functor that takes an index, an iterator, and a reference on result object. The call operator for the `Lift_idx_dst` class should have the following type.

```
void operator()(long pos, const Item& xs, Result& dst);
```

The value that is passed in for `pos` is the index in the input sequence of the item `x`. The object referenced by `dst` is the object to receive the result of the lift function.

Destination-passing-style sequential lift

```
class Seq_reduce_rng_dst;
```

The destination-passing-style sequential lift function is a C++ functor that takes a pair of iterators and a reference on an output object. The call operator for the `Seq_reduce_dst` class should have the following type.

```
void operator()(Input_iter lo, Input_iter hi, Result& dst);
```

The purpose of this function is provide an alternative sequential algorithm that is to be used to process ranges of items from the input. The range is specified by the right-open range `[lo, hi)`. The object referenced by `dst` is the object to receive the result of the sequential lift function.

Level 4

```
namespace pasl {
namespace pctl {
namespace level4 {

template <
  class Input,
  class Output,
  class Result,
```

```
  class Convert_reduce_comp,
  class Convert_reduce,
  class Seq_convert_reduce
>
void reduce(Input& in,
            Output out,
            Result id,
            Result& dst,
            Convert_reduce_comp convert_comp,
            Convert_reduce convert,
            Seq_convert_reduce seq_convert);

template <
  class Input,
  class Output,
  class Result,
  class Output_iter,
  class Merge_comp,
  class Convert_reduce_comp,
  class Convert_reduce,
  class Convert_scan,
  class Seq_convert_scan
>
void scan(Input& in,
          Output out,
          Result id,
          Output_iter outs_lo,
          Merge_comp merge_comp,
          Convert_reduce_comp convert_reduce_comp,
          Convert_reduce convert_reduce,
          Convert_scan convert_scan,
          Seq_convert_scan seq_convert_scan,
          scan_type st);

} } }
```

In the last level of our scan and reduce operators, we are going to introduce one more concept. At this level, reduction and scan no longer take as input a pair of iterator values. Instead, the input is described in an even more generic form. An *input descriptor* is an object that describes a process for dividing an input container into two or more pieces. The following input descriptor is the one we use to represent a range encoded by a pair or random-access iterator values.

```
namespace pasl {
namespace pctl {
namespace level4 {
```

```
template <class Input_iter>
class random_access_iterator_input {
public:

  using self_type = random_access_iterator_input;
  using array_type = parray<self_type>;

  Input_iter lo;
  Input_iter hi;

  random_access_iterator_input() { }

  random_access_iterator_input(Input_iter lo, Input_iter hi)
  : lo(lo), hi(hi) { }

  bool can_split() const {
    return size() >= 2;
  }

  long size() const {
    return hi - lo;
  }

  void split(random_access_iterator_input& dst) {
    dst = *this;
    long n = size();
    assert(n >= 2);
    Input_iter mid = lo + (n / 2);
    hi = mid;
    dst.lo = mid;
  }

  array_type split(long) {
    array_type tmp;
    return tmp;
  }

  self_type slice(const array_type&, long _lo, long _hi) {
    self_type tmp(lo + _lo, lo + _hi);
    return tmp;
  }

};

} } }
```

As usual, the pair (`lo`, `hi`) represents the right-open index [`lo`, `hi`). The `can_split` method returns `true` when the range contains at least two elements and `false` otherwise. The `size` method returns the number of items in the range. The `split` method divides the range in two subranges of approximately the same size, leaving the result in `dst`. The `slice` method creates a subrange starting at `lo + _lo` and ending at `lo + _hi`.

Now, we are going to see one feature that we have at level 4 that we lacked in previous levels. Because we can provide a custom input descriptor, we can provide one that destroys its input as the reduction operation proceeds. It turns out that this feature is just what we need to encode a useful algorithmic pattern: a pattern that updates a chunked sequence container in place and all in parallel. First, let us see the input descriptor, and then we will see it put to work. The chunkedseq input descriptor removes items from a chunked sequence structure as the reduction repeatedly divides the input.

```
template <class Chunkedseq>
class chunkedseq_input {
public:

  using self_type = chunkedseq_input<Chunkedseq>;
  using array_type = parray<self_type>;

  Chunkedseq seq;

  chunkedseq_input(Chunkedseq& _seq) {
    _seq.swap(seq);
  }

  chunkedseq_input(const chunkedseq_input& other) { }

  bool can_split() const {
    return seq.size() >= 2;
  }

  void split(chunkedseq_input& dst) {
    long n = seq.size() / 2;
    seq.split(seq.begin() + n, dst.seq);
  }

  array_type split(long) {
    array_type tmp;
    assert(false);
    return tmp;
  }

  self_type slice(const array_type&, long _lo, long _hi) {
```

```
    self_type tmp;
    assert(false);
    return tmp;
  }

};
```

Let us see this input descriptor at work. Recall that our `pchunkedseq` class provides a `keep_if` method, which removes items from the container according to a given predicate function. The following example shows how we can remove all odd values from a chunked sequence of `int`s.

```
pchunkedseq<int> xs = { 3, 10, 35, 2, 2, 100 };
xs.keep_if([&] (int x) { return x%2 == 0; });
std::cout << "xs = " << xs << std::endl;
```

The output is the following. All the odd values were deleted from the container.

```
{ 10, 2, 2, 100 }
```

The body of this method simply calls to a generic function in the `chunked` module, passing references to the chunked sequence object, `seq`.

```
template <class Pred>
void keep_if(const Pred& p) {
  chunked::keep_if(p, seq, seq);
}
```

The body of this function is shown below. Here, we see that, before the reduction, we feed the input sequence `xs` to our input descriptor, `in`. After the `reduce` operation completes, `xs` is going to be empty. All the items that survived the filtering process go into the `dst` chunked sequence.

```
namespace chunked {

template <class Pred, class Chunkedseq>
void keep_if(const Pred& p, Chunkedseq& xs, Chunkedseq& dst) {
  using input_type = level4::chunkedseq_input<Chunkedseq>;
  using output_type = level3::chunkedseq_output<Chunkedseq>;
  using value_type = typename Chunkedseq::value_type;
  input_type in(xs);
  output_type out;
  Chunkedseq id;
  auto convert_reduce_comp = [&] (input_type& in) {
    return in.seq.size();
  };
  auto convert_reduce = [&] (input_type& in, Chunkedseq& dst) {
    while (! in.seq.empty()) {
      value_type v = in.seq.pop_back();
      if (p(v)) {
```

```
        dst.push_front(v);
      }
    }
  };
  auto seq_convert_reduce = convert_reduce;
  level4::reduce(in, out, id, dst, convert_reduce_comp,
                 convert_reduce, seq_convert_reduce);
}

}
```

To summarize, we saw that input descriptors are general enough to describe read-only inputs as well as inputs that are to be consumed destructively. The `keep_if` example shows the usefulness of the latter input type. Using `keep_if` we can process our input sequence destructively and in parallel. As a bonus, during the processing, chunks can be reused on the fly by the memory allocator to construct the output chunked sequence, thanks to the property that chunks are popped off the input sequence as the `reduce` is working.

Template parameters

Table 30: Template parameters that are introduced in level 4.

Template parameter	Description
`Input`	Type of input to the reduction
`Convert_reduce`	Function to convert the items of a given input and then produce a specified reduction on the converted items
`Convert_scan`	Function to convert the items of a given input and then produce a specified scan on the converted items
`Seq_convert_scan`	Alternative sequentialized version of the `Convert_scan` function
`Convert_reduce_comp`	Complexity function associated with a convert function
`Seq_convert_reduce`	Alternative sequentialized version of the `Convert_reduce` function

Input

```
class Input;
```

Table 31: Constructors that are required for the `Input` class.

Constructor	Description
copy constructor	Copy constructor

Table 32: Public methods that are required for the `Input` class.

Public method	Description
`can_split`	Return value to indicate whether split is possible
`size`	Return the size of the input
`slice`	Return a specified slice of the input
`split`	Divide the input into two pieces

Copy constructor

```
Input(const Input& other);
```

Copy constructor.

Can split

```
bool can_split() const;
```

Return a boolean value to indicate whether a split is possible.

Size

```
long size() const;
```

Returns the size of the input.

Slice

```
Input slice(parray<Input>& ins, long lo, long hi);
```

Returns a slice of the input that occurs logically in the right-open range `[lo, hi)`, optionally using `ins`, the results of a precomputed application of the `split` function.

Split

```
void split(Input& dst);          // (1)
parray<Input> split(long n));    // (2)
```

(1) Transfer a fraction of the contents of the current input object to the input object referenced by `dst`.

The behavior of this method may be undefined when the `can_split` function would return `false`.

(2) Divide the contents of the current input object into at most `n` pieces, returning an array which stores the new pieces.

Convert-reduce complexity function

```
class Convert_reduce_comp;
```

The convert-complexity function is a C++ functor which returns a positive number that associates a weight value to a given input object. The `Convert_reduce_comp` class should provide the following call operator.

```
long operator()(const Input& in);
```

Convert-reduce

```
class Convert_reduce;
```

The convert function is a C++ functor which takes a reference on an input value and computes a result value, leaving the result value in an output cell. The `Convert_reduce` class should provide a call operator with the following type.

```
void operator()(Input& in, Result& dst);
```

Sequential convert-reduce

```
class Seq_convert_reduce;
```

The sequential convert function is a C++ functor whose purpose is to substitute for the ordinary convert function when input size is small enough to sequentialize. The `Seq_convert_reduce` class should provide a call operator with the following type.

```
void operator()(Input& in, Result& dst);
```

The sequential convert function should always compute the same result as the ordinary convert function given the same input.

Convert-scan

```
class Convert_scan;
```

The convert function is a C++ functor which takes a reference on an input value and computes a result value, leaving the result value in an output cell. The `Convert_scan` class should provide a call operator with the following type.

```
void operator()(Input& in, Output_iter outs_lo);
```

Sequential convert-scan

```
class Seq_convert_scan;
```

The sequential convert function is a C++ functor whose purpose is to substitute for the ordinary convert function when input size is small enough to sequentialize. The `Seq_convert_scan` class should provide a call operator with the following type.

```
void operator()(Input& in, Output_iter outs_lo);
```

The sequential convert function should always compute the same result as the ordinary convert function given the same input.

Derived operations

Type-level operators Sometimes, as we will see in this section, the pctl defines a function that both takes as template parameter an iterator class and extracts from that iterator class the type of the items that are referenced by the iterator. For this purpose, the pctl defines a few type-level functions whose purpose is to extract from the iterator the corresponding value, reference and pointer types.

```
namespace pasl {
namespace pctl {

template <class Iter>
using value_type_of = typename std::iterator_traits<Iter>::value_type;

template <class Iter>
using reference_of = typename std::iterator_traits<Iter>::reference;

template <class Iter>
using pointer_of = typename std::iterator_traits<Iter>::pointer;

} }
```

Pack

```
namespace pasl {
namespace pctl {

template <class Item_iter, class Flags_iter>
parray<value_type_of<Item_iter>> pack(Item_iter lo,
                                      Item_iter hi,
                                      Flags_iter flags_lo);

} }
```

Filter

```
namespace pasl {
namespace pctl {

template <class Iter, class Pred_idx>
parray<value_type_of<Iter>> filteri(Iter lo, Iter hi, Pred_idx pred_idx);

template <class Iter, class Pred>
parray<value_type_of<Iter>> filter(Iter lo, Iter hi, Pred pred);

} }
```

Predicate

```
class Pred;     // (1)
class Pred_idx; // (2)

bool operator()(const Item& x);

bool operator()(long pos, const Item& x);
```

Max index

```
namespace pasl {
namespace pctl {

template <
  class Iter,
  class Item,
  class Compare,
  class Lift
>
long max_index(Iter lo, Iter hi, Item id, Compare compare, Lift lift);

template <
  class Iter,
  class Item,
  class Compare
>
long max_index(Iter lo, Iter hi, Item id, Compare compare);

} }
```

Lift function

```
class Lift;
Item operator()(const Item& x);
```

Comparison function

```
class Compare;
bool operator()(Item x, Item y);
```

In-place operations

Here, document the functions which are exported by `dpsdatapar.hpp`.

Merging and sorting

```
class Compare;
bool operator()(const Item& x, const Item& y);
class Weight;
long operator()(const Item& x);
class Weight_rng;
long operator()(const Item& lo, const Item* hi);
```

Comparison-based merge

Iterator based

```
namespace pasl {
namespace pctl {

template <
  class Input_iter,
  class Output_iter,
  class Compare
>
void merge(Input_iter first1, Input_iter last1,
           Input_iter first2, Input_iter last2,
           Output_iter d_first, Compare compare);

template <
  class Input_iter,
```

```
  class Output_iter,
  class Weight,
  class Compare
>
void merge(Input_iter first1, Input_iter last1,
           Input_iter first2, Input_iter last2,
           Output_iter d_first, Weight weight,
           Compare compare);

} }

namespace pasl {
namespace pctl {
namespace range {

template <
  class Input_iter,
  class Output_iter,
  class Weight_rng,
  class Compare
>
void merge(Input_iter first1, Input_iter last1,
           Input_iter first2, Input_iter last2,
           Output_iter d_first, Weight_rng weight_rng,
           Compare compare);

} } }
```

Parallel chunked sequence

```
namespace pasl {
namespace pctl {
namespace chunked {

template <class Item, class Compare>
pchunkedseq<Item> merge(pchunkedseq<Item>& xs, pchunkedseq<Item>& ys,
                        Compare compare);

template <class Item, class Weight, class Compare>
pchunkedseq<Item> merge(pchunkedseq<Item>& xs, pchunkedseq<Item>& ys,
                        Weight weight, Compare compare);

} } }

namespace pasl {
namespace pctl {
```

```
namespace range {

template <class Item, class Weight_rng, class Compare>
pchunkedseq<Item> pcmerge(pchunkedseq<Item>& xs, pchunkedseq<Item>& ys,
                          Weight_rng weight_rng, Compare compare);

} } }
```

Comparison-based sort

Iterator based

```
namespace pasl {
namespace pctl {

template <class Iter, class Compare>
void sort(Iter lo, Iter hi, Compare compare);

template <class Iter, class Weight, class Compare>
void sort(Iter lo, Iter hi, Weight weight, Compare compare);

} }
```

```
namespace pasl {
namespace pctl {
namespace range {

template <class Iter, class Weight_rng, class Compare>
void sort(Iter lo, Iter hi, Weight_rng weight_rng, Compare compare);

} } }
```

Parallel chunked sequence

```
namespace pasl {
namespace pctl {
namespace pchunked {

template <class Item, class Compare>
pchunkedseq<Item> sort(pchunkedseq<Item>& xs, Compare compare);

template <class Item, class Weight, class Compare>
pchunkedseq<Item> sort(pchunkedseq<Item>& xs,
                       Weight weight,
```

```
                        Compare compare);

} } }

namespace pasl {
namespace pctl {
namespace pchunked {
namespace range {

template <class Item, class Weight_rng, class Compare>
pchunkedseq<Item> sort(pchunkedseq<Item>& xs,
                       Weight_rng weight_rng,
                       Compare compare);

} } } }
```

Integer sort

Iterator based

```
namespace pasl {
namespace pctl {

template <class Iter>
void integersort(Iter lo, Iter hi);

} }
```

Parallel chunked sequence

```
namespace pasl {
namespace pctl {
namespace pchunked {

template <class Integer>
pchunkedseq<Integer> integersort(pchunekdseq<Integer>& xs);

} } }
```