BERKELEY LAB
Lawrence Berkeley National Laboratory

U.S. DEPARTMENT OF ENERGY

UNIVERSITY OF CALIFORNIA

BERKELEY LAB

LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

C++17 Parallel Algorithms

Bryce Adelstein Lelbach

Computer Architecture Group, Computing Research Division

STE||AR GROUP

BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

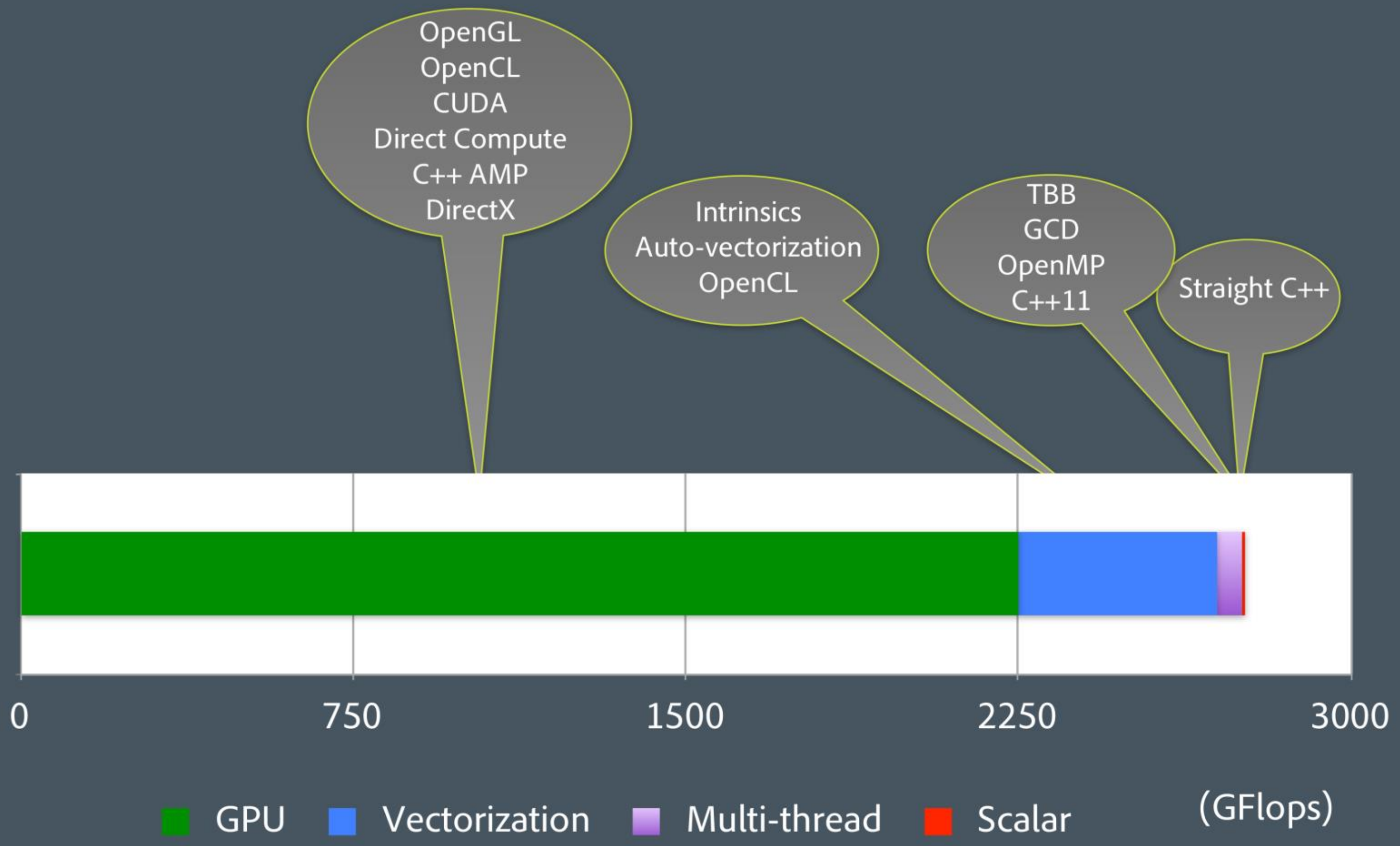
C++17 Parallel Algorithms

Bryce Adelstein Lelbach

Computer Architecture Group, Computing Research Division

STE||AR GROUP

Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)



18

Source: Sean Parent's C++Now 2012 Keynote

Hardware is increasingly parallel and increasingly diverse. Vendor-neutral parallel programming abstractions are desperately needed to avoid leaving performance on the table.

C++11/14 provide low-level concurrency primitives, but no real higher-level generic abstractions for parallel programming.

Q: What are standard algorithms?

The standard says the algorithms library

> *describes components that C++ programs may use to perform algorithmic operations on containers and other sequences*

(25.1 [algorithms.general] p1)

Q: What are standard algorithms?

A: Generic operations on sequences.

Q: What are standard algorithms?

A: Generic operations on sequences.

(except `min`, `max`, etc)

Three types of standard algorithms:

- Non-modifying sequence operations.
- Mutating sequence operations.
- Sorting and related operations.

The Committee Draft of C++17 includes a new standard parallel algorithms library which provides parallelized versions of these sequence operations.

Q: What does parallel mean?

Bit-Level Parallelism	implicit
Instruction-Level Parallelism	↑
Vector-Level Parallelism	↕
Task-Level Parallelism	↓
Process-Level Parallelism	explicit

Bit-Level Parallelism	implicit
Instruction-Level Parallelism	
Vector-Level Parallelism	
Task-Level Parallelism	
Process-Level Parallelism	explicit

Bit-Level Parallelism

Instruction-Level Parallelism

Vector-Level Parallelism

Task-Level Parallelism

Process-Level Parallelism

implicit

explicit

Parallel algorithms library components:

- `ExecutionPolicy` concept.
- Three standard execution policies.
- New `ExecutionPolicy` overloads for most existing standard algorithms.
- New “unordered” algorithms based on existing “ordered” algorithms.
- Fused algorithms.

An `ExecutionPolicy` describes how a generic algorithm may be parallelized.

They allow programmers to request parallelism and describe constraints.

Standard execution policies:

- `std::execution::par` – operations are <u>indeterminately sequenced</u> in the calling thread.
- `std::execution::par` – operations are <u>indeterminately sequenced</u> with respect to each other within the same thread.
- `std::execution::par_unseq` – operations are <u>unsequenced</u> with respect to each other and possibly interleaved.

Suppose we are using the following binary operation with `std::transform`:

```
double multiply(double x, double y)
{ return x * y; }

std::transform(
  // "Left" input sequence.
  x.begin(), x.end(),
  y.begin(), // "Right" input sequence.
  x.begin(), // Output sequence.
  multiply);
```

STE||AR GROUP

BERKELEY LAB

Suppose we are using the following <u>binary</u> operation with `std::transform`:

```
load x[i] to a scalar register
load y[i] to a scalar register
multiply x[i] and y[i]
store the result to x[i]
```

std::par

```
load x[i  ] to a scalar register
load y[i  ] to a scalar register
multiply x[i  ] and y[i  ]
store the result to x[I  ]
load x[i+1] to a scalar register
load y[i+1] to a scalar register
multiply x[i+1] and y[i+1]
store the result to x[i+1]
load x[i+2] to a scalar register
load y[i+2] to a scalar register
multiply x[i+2] and y[i+2]
store the result to x[i+2]
load x[i+3] to a scalar register
load y[i+3] to a scalar register
multiply x[i+3] and y[i+3]
store the result to x[i+3]
```

STE||AR GROUP

BERKELEY LAB

std::par

```
load x[i  ] to a scalar register
load y[i  ] to a scalar register
multiply x[i  ] and y[i  ]
store the result to x[I  ]
load x[i+1] to a scalar register
load y[i+1] to a scalar register
multiply x[i+1] and y[i+1]
store the result to x[i+1]
load x[i+2] to a scalar register
load y[i+2] to a scalar register
multiply x[i+2] and y[i+2]
store the result to x[i+2]
load x[i+3] to a scalar register
load y[i+3] to a scalar register
multiply x[i+3] and y[i+3]
store the result to x[i+3]
```

std::par_unseq

```
load x[i  ] to a scalar register
load x[i+1] to a scalar register
load x[i+2] to a scalar register
load x[i+3] to a scalar register
load y[i  ] to a scalar register
load y[i+1] to a scalar register
load y[i+2] to a scalar register
load y[i+3] to a scalar register
multiply x[i  ] and y[i  ]
multiply x[i+1] and y[i+1]
multiply x[i+2] and y[i+2]
multiply x[i+3] and y[i+3]
store the result to x[i  ]
store the result to x[i+1]
store the result to x[i+2]
store the result to x[i+3]
```

STE||AR GROUP

BERKELEY LAB

`std::par`

```
load x[i  ] to a scalar register
load y[i  ] to a scalar register
multiply x[i  ] and y[i  ]
store the result to x[I  ]
load x[i+1] to a scalar register
load y[i+1] to a scalar register
multiply x[i+1] and y[i+1]
store the result to x[i+1]
load x[i+2] to a scalar register
load y[i+2] to a scalar register
multiply x[i+2] and y[i+2]
store the result to x[i+2]
load x[i+3] to a scalar register
load y[i+3] to a scalar register
multiply x[i+3] and y[i+3]
store the result to x[i+3]
```

`std::par_unseq`

```
load x[i:i+3] to a vector register
load y[i:i+3] to a vector register
multiply x[i:i+3] and y[i:i+3]
store the results to x[i:i+3]
```

STE||AR GROUP

BERKELEY LAB

New ExecutionPolicy overloads for most existing standard algorithms.

adjacent_difference	is_sorted[_until]	rotate[_copy]
adjacent_find	lexicographical_compare	Search[_n]
all_of	max_element	set_difference
any_of	merge	set_intersection
copy[_if|_n]	min_element	set_symmetric_difference
count[_if]	minmax_element	set_union
equal	mismatch	sort
fill[_n]	move	stable_partition
find[_end|_first_of|_if|_if_not]	none_of	stable_sort
for_each	nth_element	swap_ranges
generate[_n]	partial_sort[_copy]	transform
includes	partition[_copy]	uninitialized_copy[_n]
inplace_merge	remove[_copy|_copy_if|_if]	uninitialized_fill[_n]
is_heap[_until]	replace[_copy|_copy_if|_if]	unique
is_partitioned	reverse[_copy]	unique_copy

```
std::vector<T> x = // ...

std::sort(std::execution::par,
          x.begin(), x.end());
```

```
std::vector<T> x = // ...

std::for_each(std::execution::par_unseq,
              x.begin(), x.end(), process);
```

```
std::vector<T> x = // ...

#pragma omp parallel for simd
for (std::size_t i = 0; i < x.size(); ++i)
  process(x[i]);
```

New “unordered” algorithms based on existing “ordered” algorithms.

- `std::reduce`
- `std::inclusive_scan`
- `std::exclusive_scan`
- `std::transform_reduce`

`std::reduce` - unordered `std::accumulate`

```
T v = std::reduce([ep,]
                  first, last,
                  [init,] [op])
```

`std::accumulate`:

first, `acc = init`

then for every `it` in `[first, last)` in <u>order</u>

`acc = binary_op(acc, *it)`

`std::reduce`:

`GSUM(binary_op, init, *first, ...)`

Commutativity: Changing the order of operations does not change the result.

- Integer Addition: `x + y == y + x`
- Integer Multiplication: `xy == yx`
- Integer Subtraction: `x – y != y - x`

Associativity: The grouping of operations does not change the result.

- Integer Addition: `(x + y) + z == x + (y + z)`
- Integer Multiplication: `(xy)z == x(yz)`
- Integer Subtraction: `(x – y) – z == x – (y – z)`

$$
\mathrm{GNSUM}(op, a^1, \ldots, a^N) =
\begin{cases}
a^1 & N == 1 \\
op(\mathrm{GNSUM}(op, a^1, \ldots, a^k), & \text{otherwise} \\
\quad \mathrm{GNSUM}(op, a^{k+1}, \ldots, a^N)) &
\end{cases}
$$

STE||AR GROUP

BERKELEY LAB

$\texttt{GSUM(op, } a^1\texttt{, …, } a^N\texttt{) == GNSUM(op, } b^1\texttt{, …, } b^N\texttt{)}$

where b^1, …, b^N may be any permutation of b^1, …, b^N

```
std::vector<double> x{1e-2, 1e-1, 1e0, 1e1, 1e2};

// sum ~= 111.111
double sum = std::accumulate(x.begin(), x.end(), 0.0);
```

```
std::vector<double> x{1e-2, 1e-1, 1e0, 1e1, 1e2};

// sum ~= 111.111
double sum = 0.0;
       sum = sum + x[0];
       sum = sum + x[1];
       sum = sum + x[2];
       sum = sum + x[3];
       sum = sum + x[4];
```

STE||AR GROUP

BERKELEY LAB

```
std::vector<double> x{1e-2, 1e-1, 1e0, 1e1, 1e2};

// sum ~= 111.111
double sum = std::reduce(x.begin(), x.end(), 0.0);
```

```
std::vector<double> x{1e-2, 1e-1, 1e0, 1e1, 1e2};

// sum ~= 111.111
double sum = 0.0;
       sum = sum + x[0];
       sum = sum + x[1];
       sum = sum + x[2];
       sum = sum + x[3];
       sum = sum + x[4];
```

```
std::vector<double> x{1e-2, 1e-1, 1e0, 1e1, 1e2};

// sum ~= 111.111
double sum = 0.0;
       sum = sum + x[1];
       sum = sum + x[0];
       sum = sum + x[2];
       sum = sum + x[4];
       sum = sum + x[3];
```

STE||AR GROUP

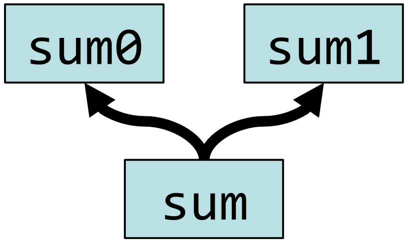
BERKELEY LAB

```cpp
std::vector<double> x{1e-2, 1e-1, 1e0, 1e1, 1e2};

// GNSUM(plus, x[2], x[3], x[4])
double sum0 = x[2] + x[3] + x[4];

// GNSUM(plus, 0.0,  x[0], x[1])
double sum1 = 0.0  + x[0] + x[1];

// GNSUM(plus, 0.0, x[0], ..., x[4])
// = plus(GNSUM(plus, x[2], x[3], x[4]),
//        GNSUM(plus, 0.0,  x[0], x[1]))
double sum = sum0 + sum1;
```



STE||AR GROUP

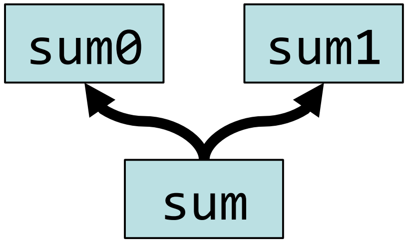
BERKELEY LAB

```
std::vector<double> x{1e-2, 1e-1, 1e0, 1e1, 1e2};

// GNSUM(plus, x[0], x[4], x[2])
double sum0 = x[0] + x[4] + x[2];

// GNSUM(plus, 0.0,  x[3], x[1])
double sum1 = 0.0  + x[3] + x[1];

// GNSUM(plus, 0.0, x[0], ..., x[4])
// = plus(GNSUM(plus, x[0], x[4], x[2]),
//        GNSUM(plus, 0.0,  x[3], x[1]))
double sum = sum0 + sum1;
```



STE||AR GROUP

BERKELEY LAB

`std::inclusive_scan` - unordered `std::partial_sum`.

```
OutputIt it =
  std::inclusive_scan([ep,]
                      first, last, output,
                      [op,] [init]);
```

STE||AR GROUP

BERKELEY LAB

`std::inclusive_scan` - unordered `std::partial_sum`.

```
OutputIt it =
  std::inclusive_scan([ep,]
                      first, last, output,
                      [op,] [init]);
```

```
*(output)   = *first;
*(output+1) = *first + *(first+1);
*(output+2) = *first + *(first+1) + *(first+2);
// ...
```

`std::exclusive_scan()` - exclusive prefix sum.

```
OutputIt it =
  std::exclusive_scan([ep,]
                      first, last, output,
                      [op,] [init]);
```

```
*(output)   = init;
*(output+1) = init + *first;
*(output+2) = init + *first + *(first+1);
// ...
```

Fused algorithms:

- `transform_reduce`
- `transform_inclusive_scan`
- `transform_exclusive_scan`

C++17 Parallel Algorithms

Bryce Adelstein Lelbach

Computer Architecture Group, Computing Research Division

BERKELEY LAB

LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

~~C++17 Parallel Algorithms~~

std::transform_reduce

Bryce Adelstein Lelbach

Computer Architecture Group, Computing Research Division

STE||AR GROUP

Execution policy (optional)

Input sequence

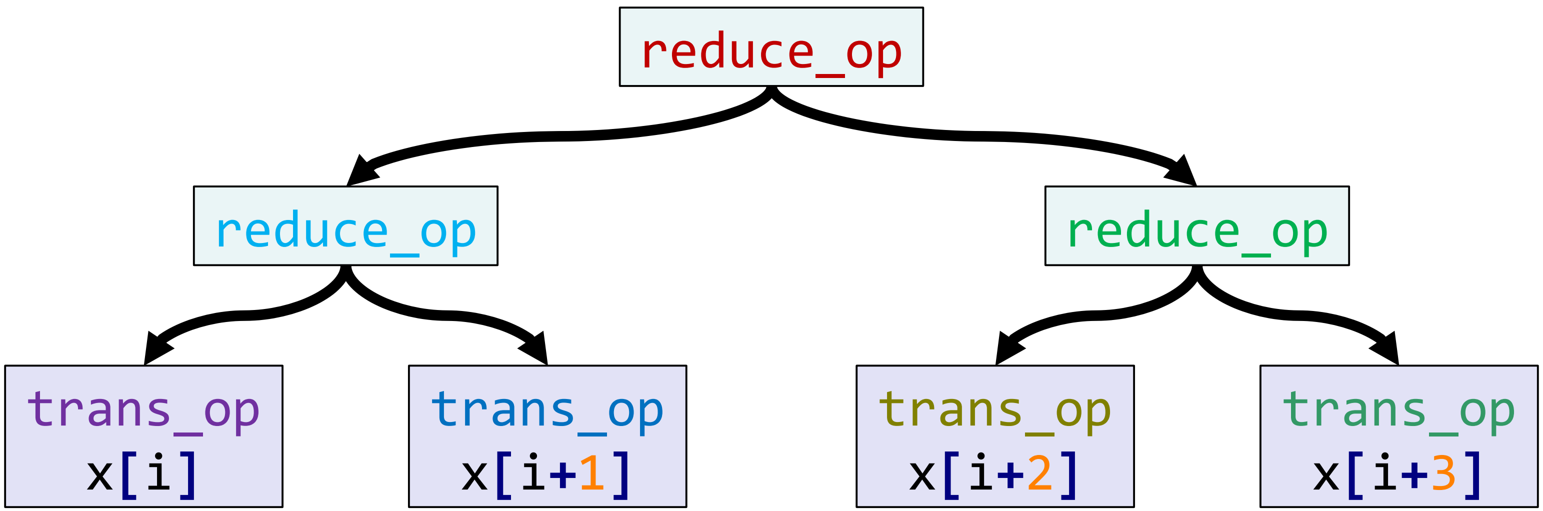
```
R v = std::transform_reduce([ep,]
                            first, last,
                            init, reduce_op, trans_op);
```

Initial value

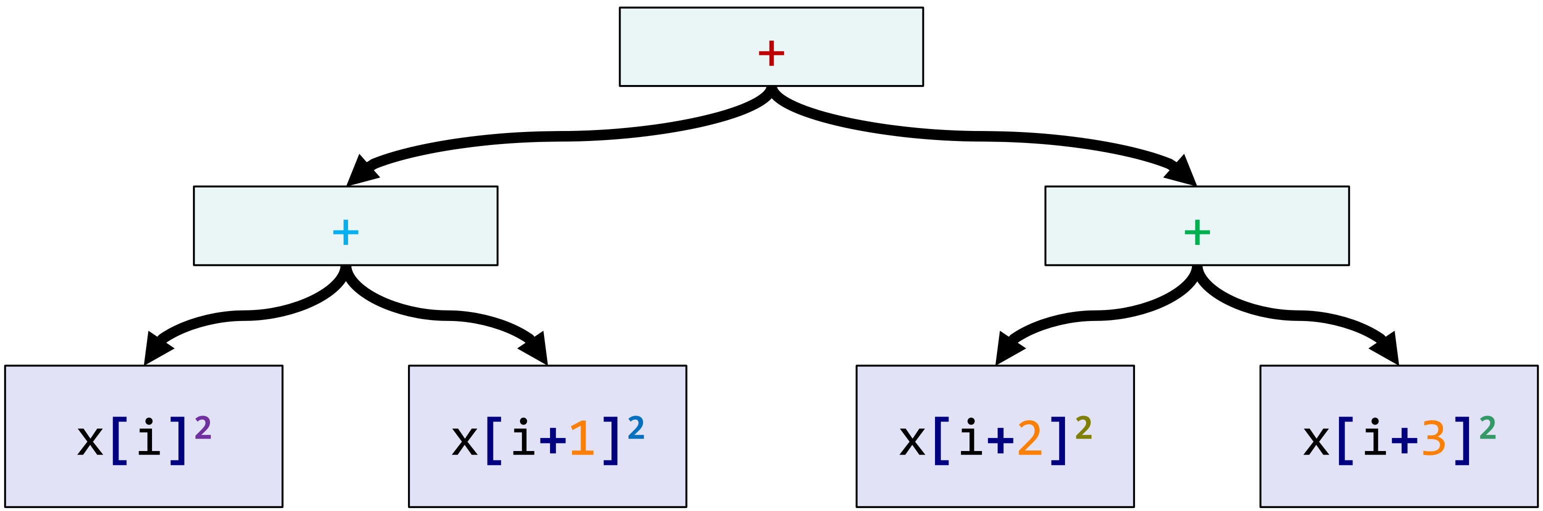
Binary reduction operation

Unary transformation operation

```
R v = reduce_op(
  reduce_op(trans_op(x[i]  ), trans_op(x[i+1])),
  reduce_op(trans_op(x[i+2]), trans_op(x[i+3])));
```



$$R\ v = (x[i]^2 + x[i+1]^2) + (x[i+2]^2 + x[i+3]^2) \ldots$$



```
std::vector<double> x = // ...

double norm =
  std::sqrt((x[0] * x[0]) + (x[1] * x[1]) + /* ... */);
```

```
std::vector<double> x = // ...

double norm =
  std::sqrt(
    std::transform_reduce(
      std::execution::par_unseq,

      // Input sequence.
      x.begin(), x.end(),

      // Initial reduction value.
      double(0.0),

      // Binary reduction op.
      [] (double xl, double xr) { return xl + xr; },

      // Unary transform op.
      [] (double x) { return x * x; }
    )
  );
```

Execution policy (optional)

Input sequence #1

Input sequence #2

```
R v = std::transform_reduce([ep,]
                  first1, last2, first2,
                  init, reduce_op, trans_op);
```

Initial value

Binary reduction operation

Binary transformation operation

```
std::vector<double> x = // ...
std::vector<double> y = // ...

double dot_product =
  (x[0] * y[0]) + (x[1] * y[1]) + // ...
```

```
std::vector<double> x = // ...
std::vector<double> y = // ...

double dot_product = std::transform_reduce(
  std::execution::par_unseq, x.begin(), x.end(), y.begin());
```

```
std::size_t word_count(std::string_view s) {
  // Goal: Count the number of word "beginnings" in the
  // input sequence.

  // ...
```

```cpp
bool is_word_beginning(char left, char right) {
  // If left is a space and right is not, we've hit a
  // new word.
  return std::isspace(left) && !std::isspace(right);
}
```

```
std::size_t word_count(std::string_view s) {
  if (s.empty()) return 0;

  // ...
```

```
std::size_t word_count(std::string_view s) {
  if (s.empty()) return 0;

  // If the first character is not a space, then it's the
  // beginning of a word.
  std::size_t wc = (!std::isspace(s.front()) ? 1 : 0);

  // ...
```

```
std::size_t word_count(std::string_view s) {
  // ...

  // Count the number of characters that start a new word
  wc +=
    std::transform_reduce(
      std::execution::par_unseq,

      // "Left" input:  s[0], s[1], ..., s[s.size() - 2]
      s.begin(), s.end() - 1,
      // "Right" input: s[1], s[2], ..., s[s.size() - 1]
      s.begin() + 1,

      // ...
```

STE||AR GROUP

BERKELEY LAB

```
std::size_t word_count(std::string_view s) {
  // ...

  // Count the number of characters that start a new word
  wc +=
    std::transform_reduce(
      std::execution::par_unseq,

      // "Left" input:  s[0], s[1], ..., s[s.size() - 2]
      s.begin(), s.end() - 1,
      // "Right" input: s[1], s[2], ..., s[s.size() - 1]
      s.begin() + 1,

      std::size_t(0), // Initial value for reduction.

      std::plus<std::size_t>(), // Binary reduction op.
      is_word_beginning         // Binary transform op: Return
                                // 1 when we hit a new word.
);

    // ...
```

STE||AR GROUP

BERKELEY LAB

```
Input sequence:
  "Whose woods these are I think I know.\n"
  "His house is in the village though;  \n"
  "He will not see me stopping here     \n"
  "To watch his woods fill up with snow.\n"
```

First Stanza of Stopping by Woods on a Snowy Evening, Robert Frost

```
Post-transform pseudo-sequence:
     0000010000010000010001010000010100000
    10001000001001001000100000001000000000
    10010000100010001001000000001000000000
    10010000010001000001000010010000100000
```

STE||AR GROUP

BERKELEY LAB

```
Input sequence:
  "Whose woods these are I think I know.\n"
  "His house is in the village though;  \n"
  "He will not see me stopping here     \n"
  "To watch his woods fill up with snow.\n"
```

First Stanza of Stopping by Woods on a Snowy Evening, Robert Frost

```
Post-transform pseudo-sequence:
         1  +  1  +  1 + 1+1  +  1+1 +
   1 + 1  +  1 +1 +1 + 1   +   1     +
   1 +1  + 1 + 1 + 1 +1   +    1     +
   1 +1  +  1 + 1  +  1  + 1 +1  + 1
```

```
bool is_word_beginning(char left, char right) {
  return std::isspace(left) && !std::isspace(right);
}

std::size_t word_count(std::string_view s) {
  if (s.empty()) return 0;

  std::size_t wc = (!std::isspace(s.front()) ? 1 : 0);

  wc +=
    std::transform_reduce(
      std::execution::par_unseq,
      s.begin(), s.end() - 1,
      s.begin() + 1,
      std::size_t(0),
      std::plus<std::size_t>(),
      is_word_beginning
    );

  return wc;
}
```

STE||AR GROUP

BERKELEY LAB

```
bool is_word_beginning(char left, char right) {
  return std::isspace(left) && !std::isspace(right);
}

std::size_t word_count(std::string_view s) {
  if (s.empty()) return 0;

  std::size_t wc =
    std::transform_reduce(
      std::execution::par_unseq,
      s.begin(), s.end() - 1,
      s.begin() + 1,
      std::size_t(!std::isspace(s.front()) ? 1 : 0),
      std::plus<std::size_t>(),
      is_word_beginning
    );

  return wc;
}
```

STE||AR GROUP

BERKELEY LAB

Sparse histogram:

- Goal: Find all the unique values in a sequence and count the number of times they occur.
- Example:
  - Input: `a, b, c, c, a, a, b, b, b, b, e`
  - Output keys: `[a, b, c, e]`
  - Output counts: `[2, 5, 3, 1]`

STE||AR GROUP

BERKELEY LAB

```
auto sparse_histogram(std::vector<T> const& x) {
  std::vector<T>           hist_keys;
  std::vector<std::size_t> hist_counts;

  // ...
```

```cpp
auto sparse_histogram(std::vector<T> const& x) {
  std::vector<T>              hist_keys;
  std::vector<std::size_t> hist_counts;

  if (x.empty())
    return std::make_tuple(std::move(hist_keys),
                           std::move(hist_counts));

  // ...
```

```
auto sparse_histogram(std::vector<T> const& x) {
  std::vector<T>              hist_keys;
  std::vector<std::size_t> hist_counts;

  if (x.empty())
    return std::make_tuple(std::move(hist_keys),
                           std::move(hist_counts));

  // Sort x to bring equal elements together.
  std::sort(std::execution::par_unseq, x.begin(), x.end());

  // ...
```

```
auto sparse_histogram(std::vector<T> const& x) {
  // ...

  // Count the number of unique elements.
  std::size_t num_unique_elements = // ...

  // ...
```

STE||AR GROUP

BERKELEY LAB

```
auto sparse_histogram(std::vector<T> const& x) {
  // ...

  // Count the number of unique elements.
  std::size_t num_unique_elements = std::transform_reduce(
    std::execution::par_unseq,

    x.begin(), x.end() - 1, // x[0], x[1], ..., x[x.size() - 2]
    x.begin() + 1,          // x[1], x[2], ..., x[x.size() - 1]

    std::size_t(1), // x isn't empty, so 1 unique key minimum.
    std::plus<std::size_t>() // Reduction operation.

    // Transform op: Return 1 if right is a new unique element.
    [] (auto&& left, auto&& right)
    // If the right is not equal to the left, then we've
    // hit the next unique element.
    { return left != right; },
  );

  // ...
```

STE||AR GROUP

BERKELEY LAB

```
auto sparse_histogram(std::vector<T> const& x) {
  // ...

  // Allocate storage.
  hist_keys.resize  (num_unique_elements);
  hist_counts.resize(num_unique_elements);

  // ...
```

```
auto sparse_histogram(std::vector<T> const& x) {
  // ...

  // Count the number of occurrences of each unique key.
  hpx::reduce_by_key(
    hpx::execution::par_unseq,

    // Input key sequence.
    x.begin(), x.end(),
    // Input value sequence.
    boost::constant_iterator<std::size_t>(1),

    // Output key sequence.
    hist_keys.begin(),
    // Output value sequence.
    hist_counts.begin()
  );

  // ...
```

STE||AR GROUP

BERKELEY LAB

```cpp
auto sparse_histogram(std::vector<T> const& x) {
  // ...

  // Count the number of occurrences of each unique key.
  hpx::reduce_by_key(
    hpx::execution::par_unseq,

    // Input key sequence.
    x.begin(), x.end(),
    // Input value sequence.
    boost::constant_iterator<std::size_t>(1),

    // Output key sequence.
    hist_keys.begin(),
    // Output value sequence.
    hist_counts.begin()
  );

  // ...
```

STE||AR GROUP

BERKELEY LAB

```
auto sparse_histogram(std::vector<T> const& x) {
  // ...

  // Count the number of occurrences of each unique key.
  hpx::reduce_by_key(
    hpx::execution::par_unseq,

    // Input key sequence.
    x.begin(), x.end(),
    // Input value sequence.
    boost::constant_iterator<std::size_t>(1),

    // Output key sequence.
    hist_keys.begin(),
    // Output value sequence.
    hist_counts.begin()
  );

  // ...
```

STE||AR GROUP

BERKELEY LAB

```cpp
auto sparse_histogram(std::vector<T> const& x) {
  // ...

  hpx::reduce_by_key(
    hpx::execution::par_unseq,

    // Input key sequence.
    x.begin(), x.end(),
    // Input value sequence.
    boost::constant_iterator<std::size_t>(1),

    // Output key sequence.
    hist_keys.begin(),
    // Output value sequence.
    hist_counts.begin()
  );

  return std::make_tuple(std::move(hist_keys),
                         std::move(hist_counts));
}
```

STE||AR GROUP

BERKELEY LAB

Parallel algorithms exception handling

- If an element access function exits via an uncaught exception, `std::terminate` is called.
- Parallel algorithms may also throw `std::bad_alloc` if temporary memory resources are needed for execution and none are available.

STE||AR GROUP

BERKELEY LAB

Acknowledgments

HPX: github.com/STEllAR-GROUP/hpx

Thrust: github.com/thrust/thrust

Boost.Compute: github.com/boostorg/compute

Jared Hoberock

Michael Garland

Grant Mercer

Hartmut Kaiser

John Biddiscombe