Data Parallelism (2)

Lecture 13

Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4700/

Today's Theme (cont.)

- You are now accustomed to thinking about parallel programming in terms of "what workers do" and "assigning work to workers"
- Today I would like you to think about describing algorithms in terms of operations on sequences of data
  - `map`
  - `filter`
  - `fold / reduce`
  - `scan / segmented scan`
  - `sort`
  - `group_by`
  - `join`
  - `partition / flatten`
- Main idea: high-performance parallel implementations of these operations exist
  - So programs written in terms of these primitives can often run efficiently on a parallel machine (if you can avoid being bandwidth bound)

Key Data Type: Sequences

- Collection of elements
  - In C++: `vector<T>` (actually any C++ container)
  - Similar constructs exist in other programming languages:
    - Scala lists: `List[T]`
    - Python: Pandas Dataframes, Numpy arrays
    - In a functional language (like Haskell): `seq T`
    - Etc.

- Important: data-parallel programs only access elements of a sequence through specific operations, not direct element access
  - C++: Iterators

Other Key Data-parallel Operations

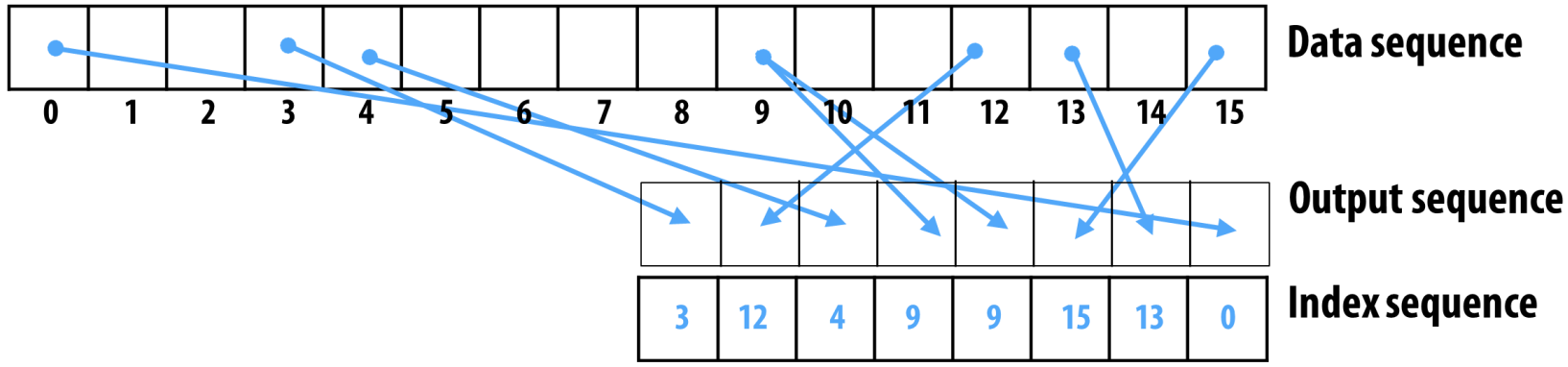
- gather(index, input, output)
  - `output[i] = input[index[i]]`

- scatter(index, input, output)
  - `output[index[i]] = input[i]`

Key Data-Parallel Operations

`output_seq = gather(index_seq, data_seq)`

- "Gather data from `data_seq` according to indices in `index_seq`"



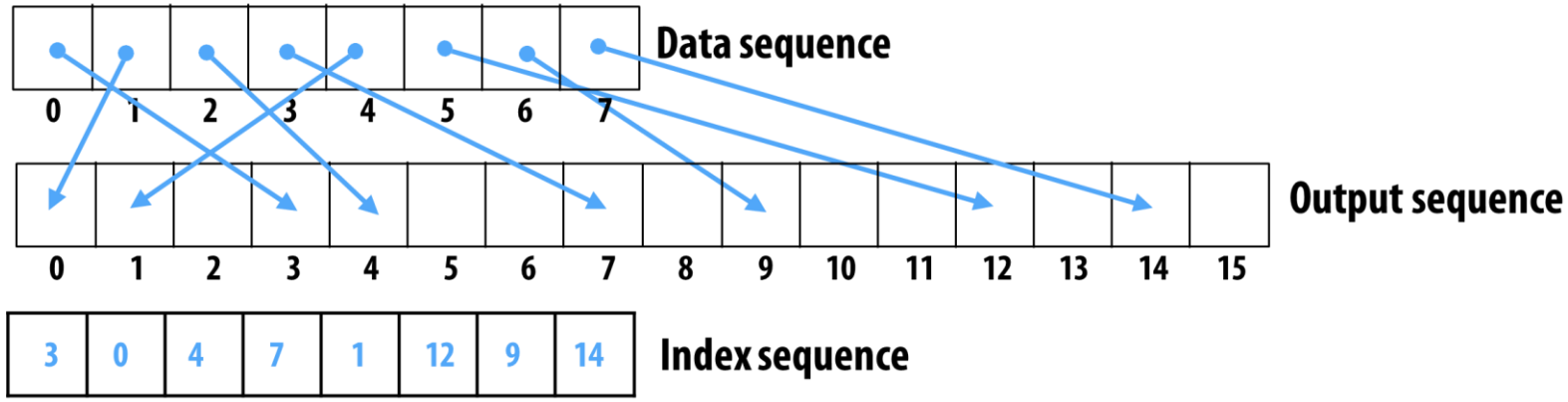
Gather Operation

```
void gather()
{
    std::vector<double> elements = {1., 2., 3., 4., 5., 6., 7., 8.};
    std::vector<int> indices = {1, 0, 3, 5, 4, 6};

    std::vector<double> gathered;
    std::ranges::transform(
        indices, std::back_inserter(gathered),
        [&](int idx) { return elements[idx]; });

    // gathered: {2., 1., 4., 6., 5., 7.}
    return 0;
}
```

Key Data-Parallel Operations

`output_seq = scatter(index_seq, data_seq)`

- "Scatter data from `data_seq` according to indices in `index_seq`"



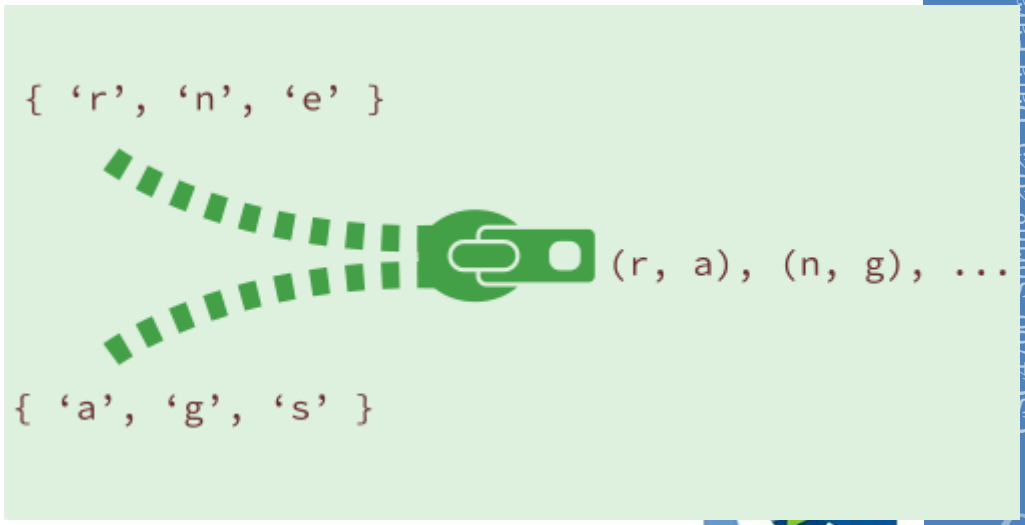
Aside: Zip Iterator

- The `zip_iterator` allows to iterate over more than one sequence at the same time
  - It is initialized from a list of iterators, one for each of the sequences to combine
  - When incremented, it increments all iterators it refers to
  - When decremented, it decrements all iterators it refers to
  - When compared with another `zip_iterator`, it compares all iterators it refers to pair-wise with the iterators of the other `zip_iterator`
  - When dereferenced, it dereferences all iterators it refers to an returns a tuple of those results.
- Iteration will stop once the first iterator reaches the end of its sequence

Aside: Zip Iterator

- A C++ zip iterator allows you to iterate over multiple ranges simultaneously:

```
template <typename R1, typename R2, typename... Rs>
auto zip(R1&& r1, R2&& r2, Rs&&... rs) {
    return iterator_range(
        zip_iterator(r1.begin(), r2.begin(), rs.begin()...),
        zip_iterator(r1.end(), r2.end(), rs.end()...));
}

int main() {
    std::vector<char> vec1{'r', 'n', 'e'};
    std::vector<char> vec2{'a', 'g', 's'};

    // Example usage of zip_iterator
    for (auto [ch1, ch2] : zip(vec1, vec2))
        std::println("({}, {})", ch1, ch2);
}
```



Aside: Zip Iterator (Simplified)

```
template <typename It1, typename It2>
struct zip_iterator
{
    zip_iterator(It1 iter1, It2 iter2) : it1(iter1), it2(iter2) {}

    friend bool operator==(zip_iterator const& lhs, zip_iterator const& rhs)
    {
        return lhs.it1 == rhs.it1 || lhs.it2 == rhs.it2;
    }

    auto operator*() { return std::make_tuple(*it1, *it2); }   // dereference

    zip_iterator& operator++() { ++it1; ++it2; return *this; } // pre-increment

    zip_iterator operator++(int) {                             // post-increment
        zip_iterator tmp = *this;
        ++it1; ++it2;
        return tmp;
    }

    It1 it1; It2 it2;
};
```

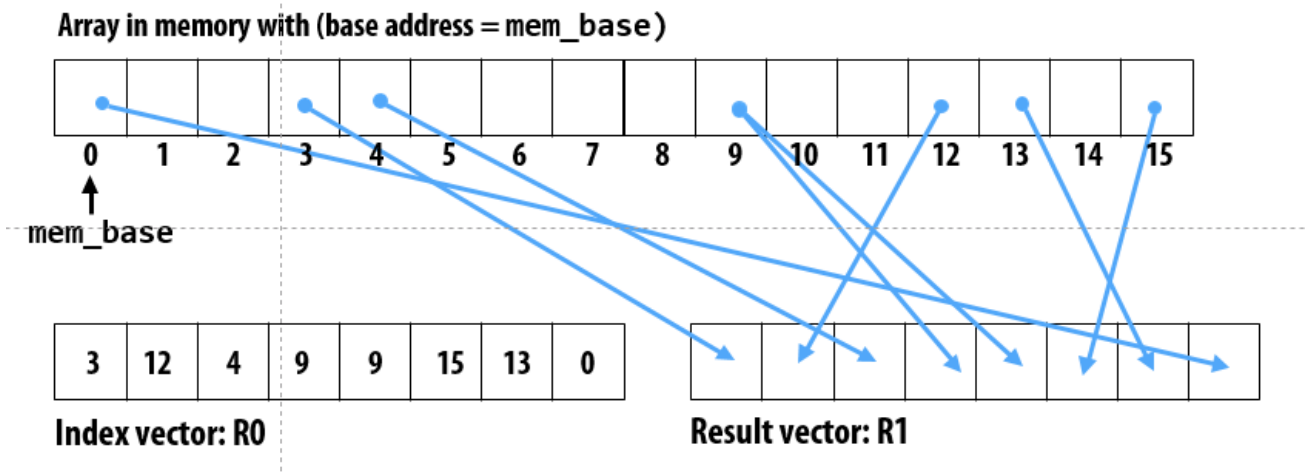
Scatter Operation

```
void scatter()
{
    std::vector<double> elements = {1., 2., 3., 4., 5., 6., 7., 8.};
    std::vector<int> indices = {1, 0, 7, 2};

    std::vector<double> scattered(elements.size(), 0.0);
    std::ranges::for_each(
        zip(indices, elements),
        [&](std::tuple<int, double> val) {
            scattered[std::get<0>(val)] = std::get<1>(val);
        });

    // scattered: {2., 1., 4., 0., 0., 0., 0., 3.}
    return 0;
}
```

Scatter/Gather Machine Instructions

`gather(R1, R0, mem_base);`

- "Gather from buffer `mem_base` into `R1` according to indices specified by `R0`."



- Gather/scatter SIMD instruction exist in AVX512
- Hardware supported gather/scatter exists on GPUs.

More Sequence Operations: Sort

- Sort
  - Sort elements in sequence based on given ordering criteria
  - In C++:

```
template <typename RandomIt, typename Compare>
void sort (RandomIt first, RandomIt last, Compare comp);  // defaults to std::less<>

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

- C++ also has parallel (stable-)sort:

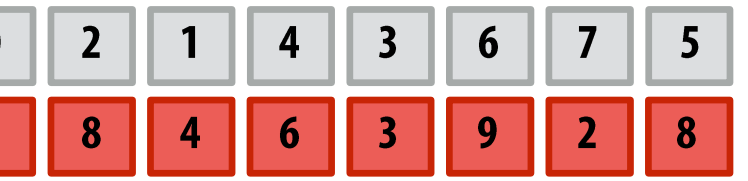
```
template <typename ExPolicy, typename RandomIt>
void sort (ExPolicy policy, RandomIt first, RandomIt last);
```

Turning a Scatter into Sort/Gather

- Special case:
  - assume elements of `index` are unique and
  - all elements are referenced in `index` (scatter is a permutation)

  For each index element: `output[index[i]] = input[i]`

```
scatter(index, input, output) {
    output = sort input sequence by values in index sequence
}
```



Turning a Scatter into Sort/Gather

```
template <typename Range1, typename Range2, typename Iter>
void scatter_sort(Range2 const& index, Range2 const& input, Iter output)
{
    std::ranges::copy(input, output);
    std::ranges::sort(
        zip(index, output),
        [](auto const& l, auto const& r) {
            return get<0>(l) < get<0>(r);
        });
}
```

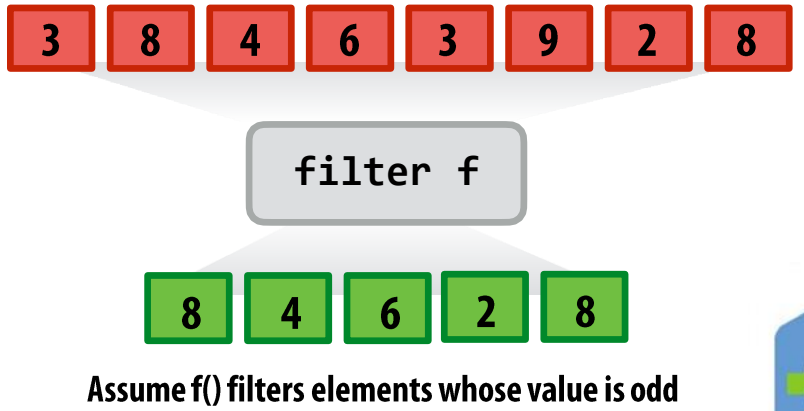
- May be faster than naïve implementation because of spatial locality

More Sequence Operations: Filter

- Filter
  - Produce a sequence of elements that match predicate
  - In C++:

```
template <typename ForwardIt, typename UnaryPred>
ForwardIt copy_if (ForwardIt first, ForwardIt last, UnaryPred p);
```

  - `remove_copy_if` does the opposite, otherwise equivalent



Filter

- Given an array input, produce an array output containing only elements such that `f(element)` is true
- Example:
  - `Input: [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]`
  - `f`: "element > 10"
  - `Output: [17, 11, 13, 19, 24]`
- Standard algorithm `copy_if`:

```
template <typename ForwardIt, typename OutIt, typename UnaryPred>
ForwardIt copy_if(ForwardIt first, ForwardIt last, OutIt dest, UnaryPred p) {
    for (ForwardIt i = first; i != last; ++i) {
        if (p(*i))          // retain elements that do match predicate
            *dest++ = *i;
    }
    return dest;
}
```

Parallel Filter

- Parallelizable?
  - Determining whether an element belongs in the output is easy
  - But determining where an element belongs in the output is hard; seems to depend on previous results….
- Solution:
  - Step 1: parallel map to compute a bit-vector for true elements:
    - `Input:  [17,  4,  6,  8, 11,  5, 13, 19,  0, 24]`
    - `Bits:   [ 1,  0,  0,  0,  1,  0,  1,  1,  0,  1]`
  - Step 2: parallel scan on the bit-vector (gives desired position in output):
    - `Bitsum: [ 1,  1,  1,  1,  2,  2,  3,  4,  4,  5]`
  - Step 3: parallel map to produce the output:
    - `Output: [17, 11, 13, 19, 24]`

Parallel Filter

```
void data_parallel_copy_if()
{
    std::vector<double> elements = {1., 2., 3., 4., 5., 6., 7., 8.};

    // map to compute a bit-vector for elements not matching predicate
    std::vector<char> flags(elements.size());
    std::transform(
        first, last, flags.begin(), [](double val) { return val > 3.; });

    // scan on the bit-vector (gives desired position in output)
    std::vector<char> bitsum(elements.size());
    std::inclusive_scan(flags.begin(), flags.end(), bitsum.begin());

    // map to produce the output (gather)
    std::vector<double> output(bitsum.back());
    std::ranges::for_each(zip(bitsum, elements),
        [&](auto curr) { output[std::get<0>(curr) - 1] = std::get<1>(curr); });
}
```

Parallel Filter

- First two steps can be combined into one pass
  - Just using a different base case for the scan
  - No effect on asymptotic complexity
- Can also combine third step into the down pass of the scan
  - Again no effect on asymptotic complexity
- Analysis: $O(N)$ work
  - 2 or 3 passes, but 3 is a constant

Implementing scatterOp with atomic sort/map/segmented-scan

- Now, assume elements in `index` are not unique, so synchronization is required for atomicity!

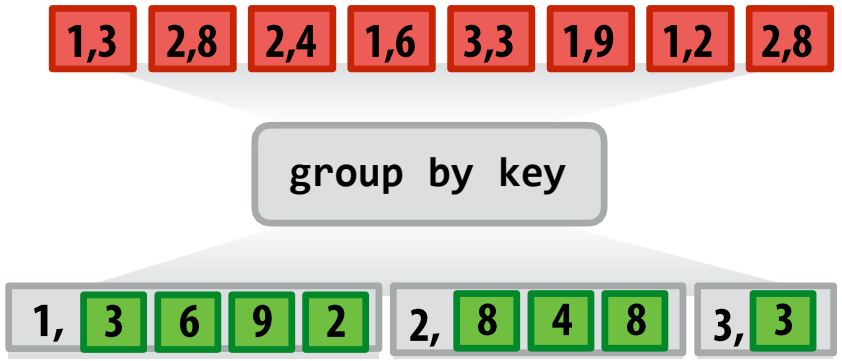
```
for all elements in sequence
    output[index[i]] = atomic_op(output[index[i]], input[i])
```

```
Example:  atomicAdd(output[index[i]], input[i])
```

- Example: $index = [1, 1, 0, 2, 0, 0]$
- Step 1: Sort input sequence according to values in `index` sequence
  - Sorted index: $[0, 0, 0, 1, 1, 2]$
  - Input sorted by index: $[input[2], input[4], input[5], input[0], input[1], input[3]]$
- Step 2: Compute starts of each range of values with the same `index` number
  - starts: `[1, 0, 0, 1, 0, 1]`
- Step 3: Segmented scan (using 'op') on each range
  - $[op(op(input[2], input[4]), input[5]), op(input[0], input[1]), input[3])]$

More Sequence Operations: Exercise

- Group by key
  - Creates a sequence of sequences containing elements with the same key



- Reduce by key
  - Creates pairs of values consisting of key and reduced result for corresponding values
- Sort by key
  - As group by key, but the sequences are sorted
- Etc.

Scan/Segmented Scan Summary

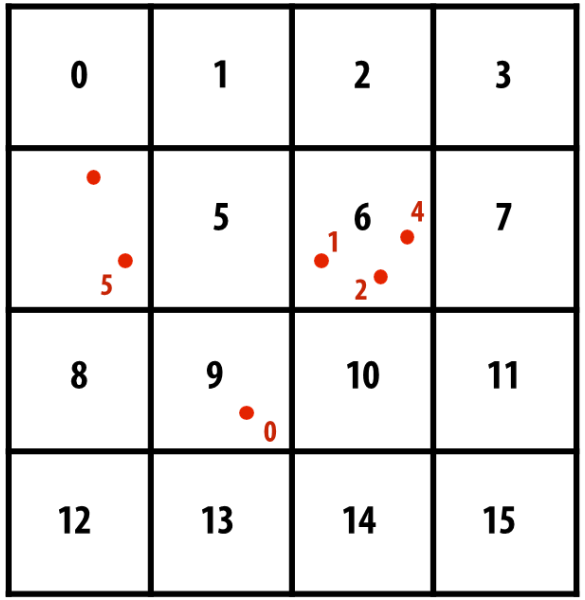
- Scan
  - Parallel implementation of (intuitively sequential application)
  - Theory: parallelism in problem is linear in number of elements
  - Practice: exploit locality, use only as much parallelism as necessary to fill the machine's execution resources
    - Great example of applying different strategies at different levels of the machine
- Segmented scan
  - Express computation and operate on irregular data structures (e.g., list of lists) in a regular, data parallel way

Examples

Grid of Particles

Example: Create Grid of Particles Data Structure

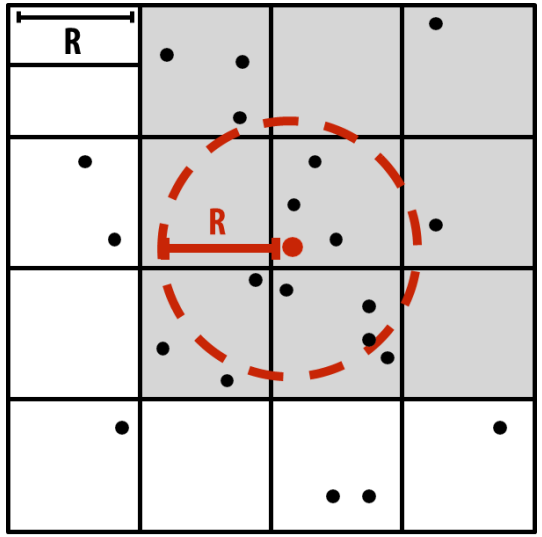
- Problem: place 1M point particles in a 16-cell uniform grid based on 2D position
  - Parallel data structure manipulation problem: build a 2D array of lists



Cell id	Count	Particle id
0	0	
1	0	
2	0	
3	0	
4	2	3, 5
5	0	
6	3	1, 2, 4
7	0	
8	0	
9	1	0
10	0	
11	0	
12	0	
13	0	
14	0	
15	0	

Common use of this Structure: N-Body Problems

- A common operation is to compute interactions with neighboring particles
  - Example: given a particle, find all particles within radius R
  - Organize particles by placing them in grid with cells of size R
  - Only need to inspect particles in surrounding grid cells
- N-Body problems require to compute pairwise forces between each pair of points
  - Creating a mesh of points allows to reduce amount of required coputations



Create Grid of Particles Data Structure

```
// one particle
struct point
{
    double x;
    double y;
};

// a cell containing particles
struct cell
{
    point upper_left;
    point lower_right;
    std::vector<point> points;
};
```

```
// the whole mesh (list of cells)
struct mesh
{
    size_t size_x = 0;
    size_t size_y = 0;
    std::vector<cell> cells;
};

// calculate the index of the cell for pt
size_t find_cell_index(
    point const& pt, mesh const& m)
{
    size_t pos_x = size_t(pt.x * m.size_x);
    size_t pos_y = size_t(pt.y * m.size_y);
    return pos_y * size_x + pos_x;
}
```

Sequential Implementation

```
double fill_mesh_sequentially(
    std::vector<point> const& points, size_t num_cells_x, size_t num_cells_y)
{
    mesh m(num_cells_x, num_cells_y);

    timer t;
    t.start();

    for (point const& p : points)
    {
        size_t idx = find_cell_index(p, m);
        m.cells[idx].points.push_back(p);
    }

    t.stop();
    return t.elapsed();
}
```

Solution 1: Parallelize over Particles

- One answer: assign one particle to each thread. Each thread computes cell containing particle, then atomically updates per cell list.
  - Massive contention: thousands of threads contending for access to update single shared data structure

```
list cell_list[16];    // 2D array of lists
lock cell_list_lock;

for each particle p           // in parallel
   c = compute cell containing p
   lock(cell_list_lock)
   append p to cell_list[c]
   unlock(cell_list_lock)
```

Solution 1: Parallelize over Particles

```
double fill_mesh_parallel_v1(
    std::vector<point> const& points, size_t num_cells_x, size_t num_cells_y)
{
    mesh m(num_cells_x, num_cells_x);
    timer t; t.start();

    hpx::mutex mtx;
    hpx::ranges::for_each(hpx::execution::par, points, [&](point const& p) {
        size_t idx = find_cell_index(p, m);
        std::lock_guard l(mtx);
        m.cells[idx].points.push_back(p);
    });

    t.stop(); return t.elapsed();
}
```

Solution 2: Use finer-granularity Locks

- Alleviate contention for single global lock by using per-cell locks
  - Assuming uniform distribution of particles in 2D space… ~16x less contention than previous solution

```
list cell_list[16];     // 2D array of lists
lock cell_list_lock[16];

for each particle p           // in parallel
   c = compute cell containing p
   lock(cell_list_lock[c])
   append p to cell_list[c]
   unlock(cell_list_lock[c])
```

Solution 2: Use finer-granularity Locks

- Alleviate contention for single global lock by using per-cell locks

```
// a cell containing particles, now with a mutex
struct cell
{
    std::mutex mtx;
    point upper_left;
    point lower_right;
    std::vector<point> points;
};
```

Solution 2: Use finer-granularity Locks

```
double fill_mesh_parallel_v2(
    std::vector<point> const& points, size_t num_cells_x, size_t num_cells_y)
{
    mesh m(num_cells_x, num_cells_y);
    timer t; t.start();

    std::ranges::for_each(std::execution::par, points,
       [&](point const& p) {
           size_t idx = find_cell_index(p, m);
           cell& c = m.cells[idx];
           std::lock_guard l(c.mtx);
           c.points.push_back(p);
       });

    t.stop(); return t.elapsed();
}
```

Solution 3: Parallelize over Cells

- Decompose work by cells: for each cell, independently compute what particles are within it
  - (eliminates contention because no synchronization is required)
  - Work inefficient: performs 16 times more particle-in-cell computations than sequential algorithm
  - Possibly insufficient parallelism: only 16 parallel tasks, but need thousands of independent tasks to efficiently utilize GPU)

```
list cell_lists[16];        // 2D array of lists

for each cell c             // in parallel
   for each particle p      // sequentially
       if (p is within c)
          append p to cell_lists[c]
```

Solution 3: Parallelize over Cells

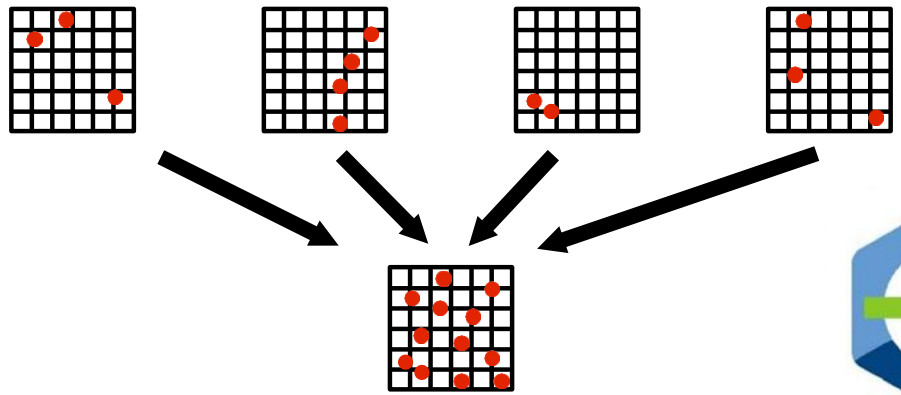
```
double fill_mesh_parallel_v3(
    std::vector<point> const& points, size_t num_cells_x, size_t num_cells_y)
{
    mesh m(num_cells_x, num_cells_y);
    timer t; t.start();

    std::ranges::for_each(std::execution::par, m.cells,
        [&](cell& c) {
            std::ranges::for_each(points,
                [&](point const& p) {
                    size_t idx = find_cell_index(p, m);
                    if (c.idx == idx) c.points.push_back(p);
                });
        });

    t.stop(); return t.elapsed();
}
```

Solution 3.1: Parallelize over Cells

```
double fill_mesh_parallel_v3(
    std::vector<point> const& points, size_t num_cells_x, size_t num_cells_y)
{
    mesh m(num_cells_x, num_cells_y);
    timer t; t.start();

    std::ranges::for_each(std::execution::par, m.cells,
        [&](cell& c) {
            std::ranges::copy_if(
                points, std::back_inserter(c.points),
                [idx = c.idx, &m](point const& p) {
                    return idx == find_cell_index(p, m);
                });
        });

    t.stop(); return t.elapsed();
}
```

Solution 4: Compute partial Results & Merge

- Yet another answer: generate N "partial" grids in parallel, then combine results
  - Example: create N threads per thread block (one thread block per core)
    - Every thread in thread block updates same grid
  - Enables faster synchronization: contention reduced by factor of N and cost of synchronization is lower because it is performed on block-local variables
  - Requires extra work: merging the N grids at the end of the computation
  - Requires extra memory footprint: stores N grids of lists, rather than just one



Solution 4: Compute partial Results & Merge

```
double fill_mesh_parallel_v4(
    std::vector<point> const& points, size_t num_cells_x, size_t num_cells_x)
{
    mesh m(num_cells_x, num_cells_y);
    timer t; t.start();

    hpx::experimental::for_loop(hpx::execution::par, size_t(0), points.size(),
        hpx::experimental::reduction(m, combine_meshes),
        [&](size_t i, mesh& local_m) {
            point const& p = points[i];
            size_t idx = find_cell_index(p, local_m);
            local_m.cells[idx].points.push_back(p);
        });

    t.stop(); return t.elapsed();
}
```

Solution 4: Compute partial Results & Merge

```cpp
// Merge partial Results
mesh& combine_meshes(mesh& acc, mesh const& curr)
{
    for (size_t i = 0; i != acc.cells.size(); ++i)
    {
        auto& dest_points = acc.cells[i].points;
        auto const& src_points = curr.cells[i].points;
        dest_points.insert(
            dest_points.end(), src_points.begin(), src_points.end());
    }
    return acc;
}
```

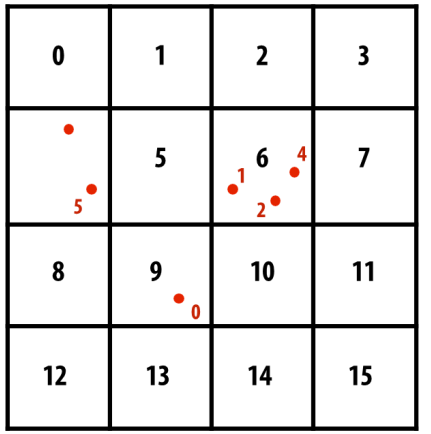
Solution 4: Compute partial Results & Merge

```
// Merge partial Results
mesh& combine_meshes_v2(mesh& acc, mesh const& curr)
{
    std::ranges::for_each(zip(acc.cells, curr.cells),
        [](auto& curr) {
            std::ranges::copy(std::get<1>(curr).points,
                std::back_inserter(std::get<0>(curr).points));
        });
    return acc;
}
```

Solution 5: Data-parallel Approach

- Step 1: map
  - Compute cell containing each particle (parallel over input particles)

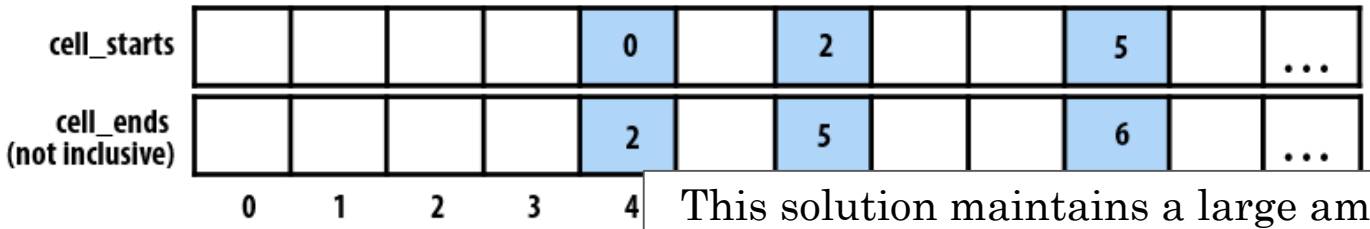
particle_index:	0	1	2	3	4	5
grid_cell:	9	6	6	4	6	4



- Step 2: sort results by cell (notice that the particle index array is permuted based on sort)

particle_index:	3	5	1	2	4	0
grid_cell:	4	4	6	6	6	9

- Step 3: find start/end of each cell (over `particle_index` elements)



This solution maintains a large amount of parallelism and removes the need for fine-grained synchronization…
At the cost of a sort and extra passes over the data (extra BW)!

Solution 5: Data-parallel Approach

```
// Compute cell containing each particle
std::ranges::transform(std::execution::par, points, indices.begin(),
    [&](point const& p) { return find_cell_index(p, m); });

// Sort results by cell index
std::ranges::sort(std::execution::par, zip(points, indices),
    [](auto const& l, auto const& r) {          // args: std::tuple<point&, size_t&>
        return hpx::get<1>(l) < hpx::get<1>(r);
    });

// find start/end of each cell, associate points with cells
std::ranges::for_each(std::execution::par, m.cells, [&](cell& c) {
    auto [b, e] = std::ranges::equal_range(indices, c.idx);
    c.points.insert(c.points.end(),
        points.begin() + (b - indices.begin()),
        points.begin() + (e - indices.begin()));
});
```

Create Grid of Particles Data Structure: Results

```
Mesh: 4x4

seq:    764 [ms]
par v1: 59593 [ms]
par v2: 6491 [ms]
par v3: 707 [ms]
par v4: 936 [ms]
par v5: 918 [ms]
```

```
Mesh: 8x8

seq:   1686 [ms]
par v1: 57192 [ms]
par v2: 2785 [ms]
par v3: 1229 [ms]
par v4: 1159 [ms]
par v5:  876 [ms]
```

```
Mesh: 16x16

seq:   1800 [ms]
par v1: 59531 [ms]
par v2: 1561 [ms]
par v3: 5366 [ms]
par v4: 3339 [ms]
par v5:  897 [ms]
```

```
Mesh: 32x32

seq:   2331 [ms]
par v1: 63897 [ms]
par v2: 1215 [ms]
par v3: 17834 [ms]
par v4: 14490 [ms]
par v5:  984 [ms]
```

Examples

Generate a Histogram

Another Example: Parallel Histogram

- Consider computing a histogram for a sequence of values

```
// maps array values to histogram bin id's for float input[N];
int find_bin_index(float value);

// bins are initialized to 0
std::vector<int> histogram_bins(NUM_BINS, 0);

for (int i = 0; i < N; ++i)
    ++histogram_bins[find_bin_index(input[i])];
```

- Challenge: Create a parallel implementation of histogram given only `map` and `sort` on sequences

Data-parallel Histogram Construction (Version 1)

```
// map find_bin_index onto input sequence to get bin ids of all elements
hpx::ranges::transform(hpx::execution::par, input, bin_ids.begin(),
    [](float val) { return find_bin_index(val); });

// find starting point of each bin in sorted list
hpx::ranges::sort(hpx::execution::par, bin_ids);

std::vector<int> bin_starts(NUM_BINS, -1);
bin_starts[bin_ids[0]] = 0;     // first bin starts with first element

hpx::experimental::for_loop(
    hpx::execution::par, 1, bin_ids.size(), [&](size_t idx) {
        // store first index of input for every bin
        if (bin_ids[idx] != bin_ids[idx - 1])
            bin_starts[bin_ids[idx]] = idx;
    });

// compute bin sizes sequentially (see next slide)
hpx::experimental::for_loop(0, NUM_BINS,
    [&](size_t idx) { adjust_bin_size(idx, bin_starts, histogram_bins); });
```

Data-parallel Histogram Construction (Version 1)

```
void adjust_bin_size(size_t idx, std::vector<int> const& bin_starts,
    std::vector<int>& histogram_bins)
{
    if (bin_starts[idx] == -1) {
        histogram_bins[idx] = 0;    // no items in this histogram bin
        return;
    }

    // find start of next bin in order to determine size of current bin
    auto next_idx = std::find_if(bin_starts.begin() + idx + 1, bin_starts.end(),
        [](size_t v) { return v != -1; });

    if (next_idx != bin_starts.end())
        histogram_bins[idx] = *next_idx - bin_starts[idx];
    else
        histogram_bins[idx] = NUM_BINS - bin_starts[idx];
}
```

Data-parallel Histogram Construction (Version 2)

```
std::vector<int>& combine_histograms(
    std::vector<int>& acc, std::vector<int> const& rhs)
{
    std::ranges::transform(acc, rhs, acc.begin(), std::plus{});
    return acc;
}

using namespace hpx::experimental;

std::vector<float> input = {...};
std::vector<int> histogram_bins(NUM_BINS, 0);
for_loop(hpx::execution::par, 0, input.size(),
    reduction(histogram_bins, combine_histograms),
    [&](size_t idx, std::vector<int>& local_histogram_bins) {
        ++local_histogram_bins[find_bin_index(input[idx])];
    });
```

Summary

- Data parallel thinking
  - Implementing algorithms in terms of simple (often widely parallelizable, efficiently implemented) operations on large data collections
- Turn irregular parallelism into regular parallelism
- Turn fine-grained synchronization into coarse synchronization
- But most solutions require multiple passes over data - bandwidth hungry!

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP