Stories from a parallel universe

Jana Machutová

Do you feel sudden tension in your brain?

SEMAPHORE

JOIN

JTHREAD

BARRIER

WAIT

DEADLOCK

FALSE SHARING

ATOMICS

LOCK

CRITICAL SECTION

MEMORY ORDER

MUTEX

DETACH

NOTIFY ALL

RACE CONDITION

LOCK-FREE

FETCH

FENCES

We have a painkiller for you!

SEMAPHORE

JTHREAD

JOIN

DEADLOCKS

BARRIER

ATOMICS

WAIT

FALSE SHARING

STANDARD LIBRARY

CRITICAL SECTION

PARALLEL ALGORITHMS

LOCK

MEMORY ORDER

MUTEX

NOTIFY ALL

RACE CONDITION

DETACH

FETCH

FENCES

LOCK-FREE

Standard library parallel algorithm

Provides parallel version of most standard library algorithms

Introduces execution policy parameter

Easy switch between sequential and several types of parallel executions

Well-optimized solutions to common problems

No need to reinvent parallel wheels

Create new worlds

```
array<animal_species, 100> species{...};

vector<jthread> sky_residents;
for_each(species.begin(), species.end(),
    [&](){
      sky_residents.push_back(jthread([]{
        fly_high();
    }));
});
```

```
for_each(execution::par_unseq, species.begin(), species.end(),
    [](){
      swim_deep();
});
```

Create new worlds

```
array<animal_species, 100> species{...};

vector<jthread> sky_residents;
for_each(species.begin(), species.end(),
    [&](){
      sky_residents.push_back(jthread([]{
        fly_high();
    }));
});
```

```
for_each(execution::par_unseq, species.begin(), species.end(),
    [](){
      swim_deep();
});
```

Create new worlds

```
array<animal_species, 1000000> species{...};

vector<jthread> sky_residents;
for_each(species.begin(), species.end(),
    [&](){
      sky_residents.push_back(jthread([]{
        fly_high();
    }));
});
```

```
for_each(execution::par_unseq, species.begin(), species.end(),
    [](){
      swim_deep();
});
```

Create new worlds

```
array<animal_species, 1000000> species{...};

vector<jthread> sky_residents;
for_each(species.begin(), species.end(),
    [&](){
      sky_residents.push_back(jthread([]{
        fly_high();
    }));
});
```

Thread #0 species [0]

Thread #1 species [1]

Thread #999 999 species [999 999]

```
for_each(execution::par_unseq, species.begin(), species.end(),
    [](){
      swim_deep();
});
```

Create new worlds

```
array<animal_species, 1000000> species{...};

vector<jthread> sky_residents;
for_each(species.begin(), species.end(),
    [&](){
      sky_residents.push_back(jthread([]{
        fly_high();
    }));
});
```

Thread #0	species [0]
Thread #1	species [1]
Thread #999 999	species [999 999]

```
for_each(execution::par_unseq, species.begin(), species.end(),
    [](){
      swim_deep();
});
```

Thread #0	species [0 – 9 999]
Thread #1	species [10 000 – 19 999]
Thread #100	species [90 000 – 99 999]

Create new worlds

```
array<animal_species, 1000000> species{...};

vector<jthread> sky_residents;
for_each(species.begin(), species.end(),
    [&](){
      sky_residents.push_back(jthread([]{
        fly_high();
    }));
});
```

Thread #0 species [0]

Thread #1 species [1]

RESOURCE UNAVAILABLE

Thread #999 999 species [999 999]

```
for_each(execution::par_unseq, species.begin(), species.end(),
    [](){
      swim_deep();
});
```

Thread #0 species [0 – 9 999]

Thread #1 species [10 000 – 19 999]

Thread #100 species [90 000 – 99 999]

Standard library parallel algorithm

In standard since C++17 with a few updates in C++20

Clever thread management is already part of the package

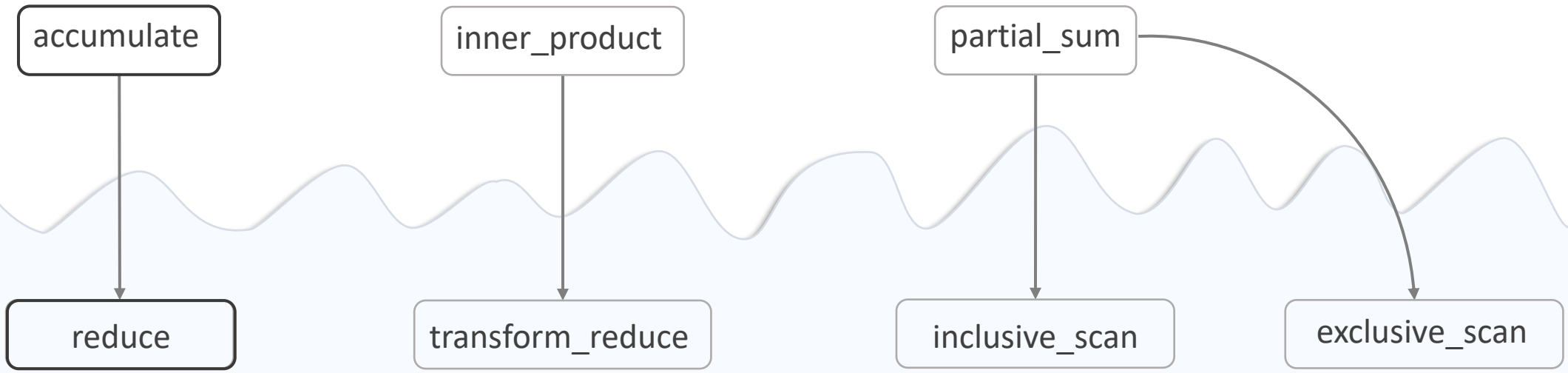
Several new algorithms introduced

Some old algorithms don’t have any parallel version

Not available for ranges so far

New algorithms introduced

All of them are here as an alternative to an existing, sequential only solution



accumulate vs reduce

```
vector<int> counts {68, 15, 4, 45, 18, 3, 2, 11};
```

```
accumulate(counts.begin(), counts.end(), 0);
```

output: **166**

```
reduce(execution::par_unseq, counts.begin(), counts.end(), 0);
```

output: **166**

accumulate vs reduce

```
vector<int> counts {68, 15, 4, 45, 18, 3, 2, 11};
```

```
accumulate(counts.begin(), counts.end(), 0 , plus{});
```

output: **166**

```
reduce(execution::par_unseq, counts.begin(), counts.end(), 0, plus{});
```

output: **166**

accumulate vs reduce

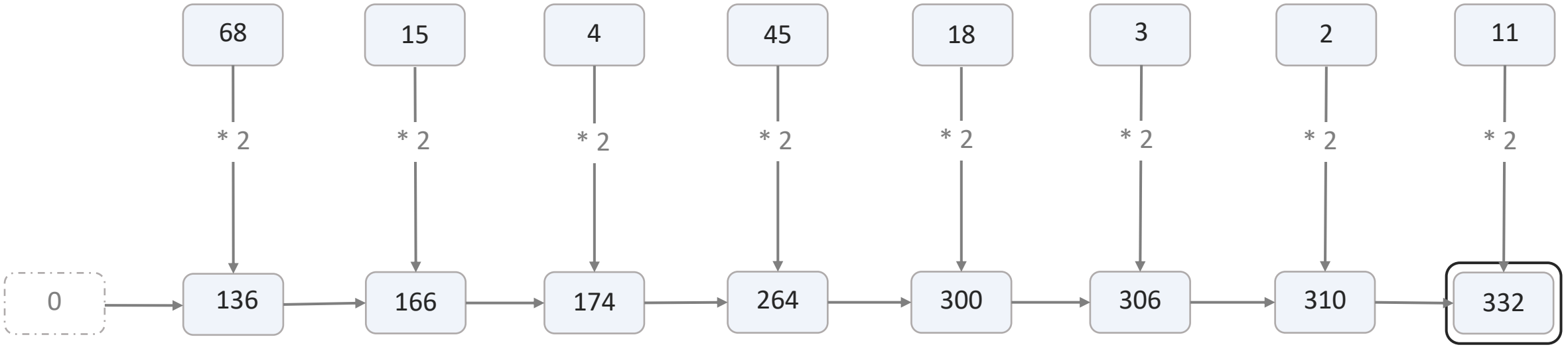
```
vector<int> counts {68, 15, 4, 45, 18, 3, 2, 11};
int         growth_factor = 2;

accumulate(counts.begin(), counts.end(), 0,
    [growth_factor](auto first, auto second){
        return first + second * growth_factor;
});
```

output: **332**

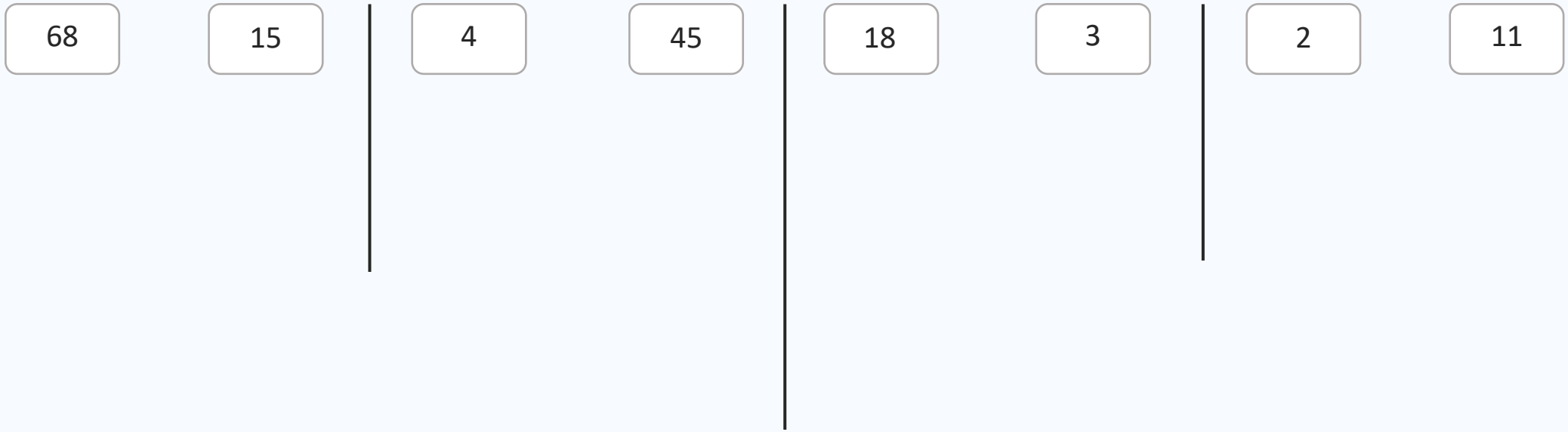
```
reduce(execution::par_unseq, counts.begin(), counts.end(), 0,
    [growth_factor](auto first, auto second){
        return first + second * growth_factor;
});
```

output: **716**

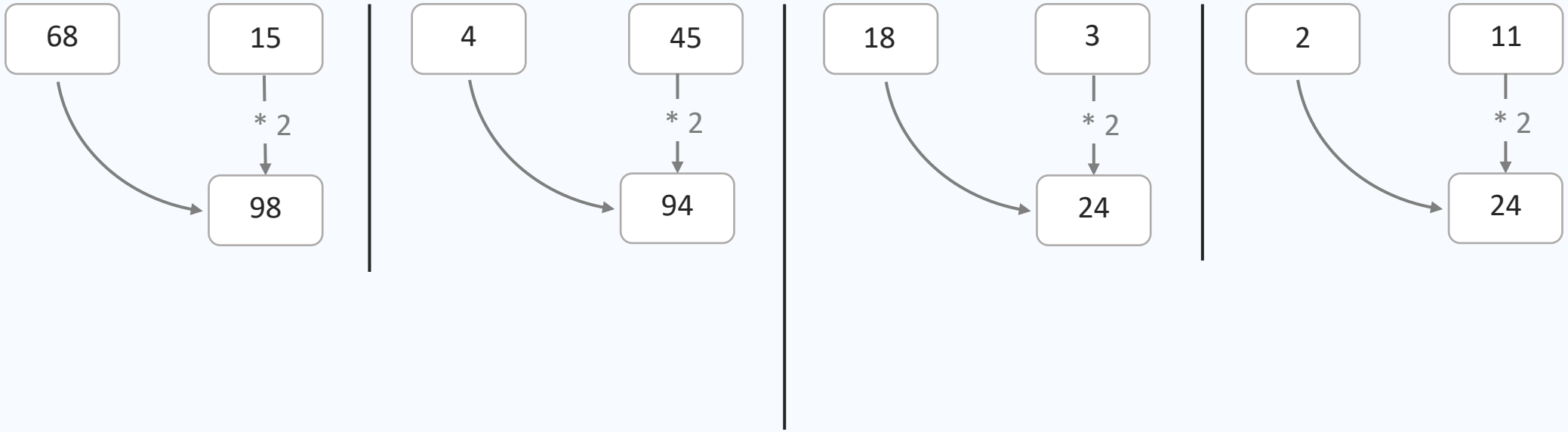
accumulate walktrough



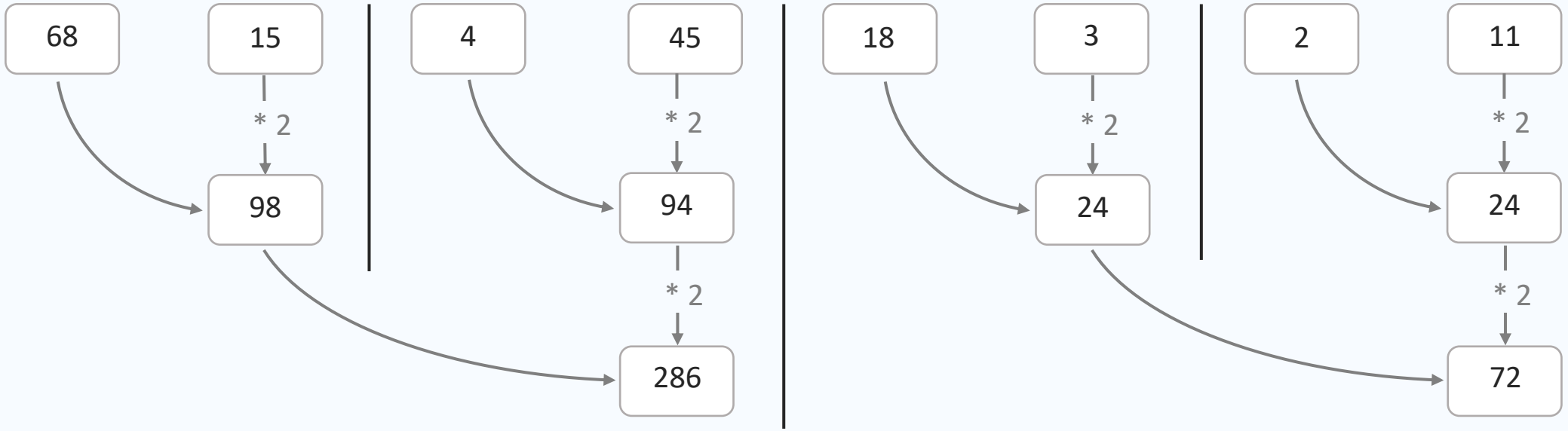
reduce walktrough



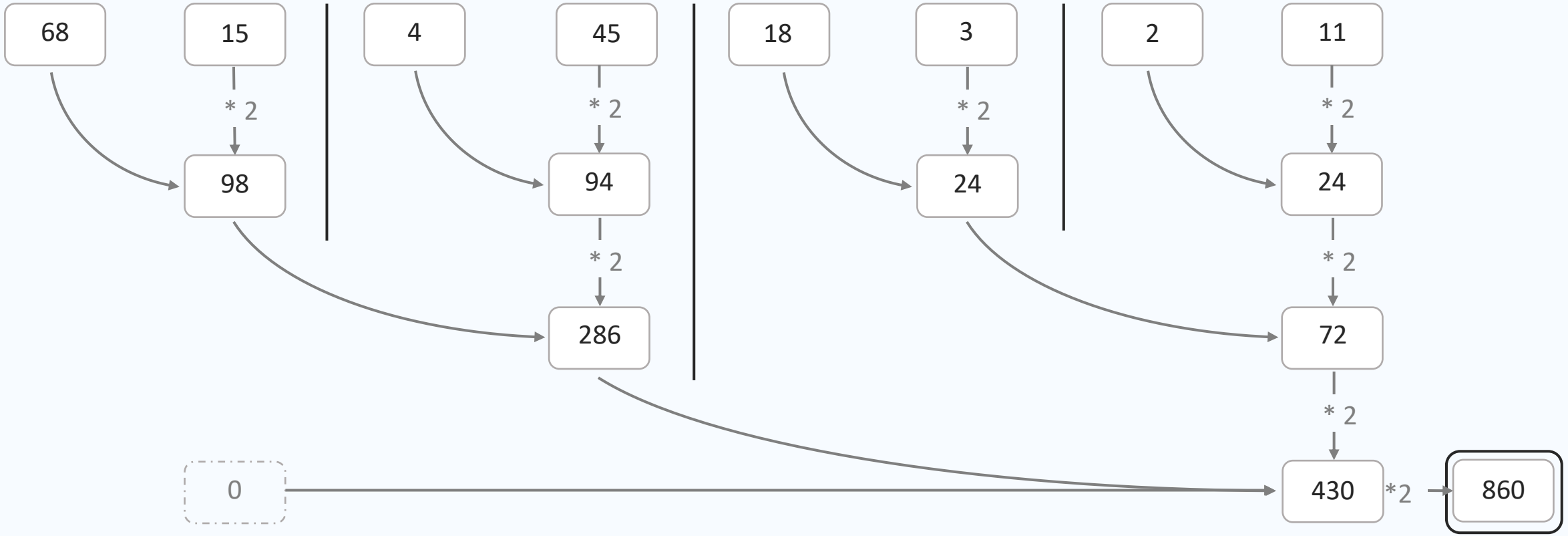
reduce walktrough



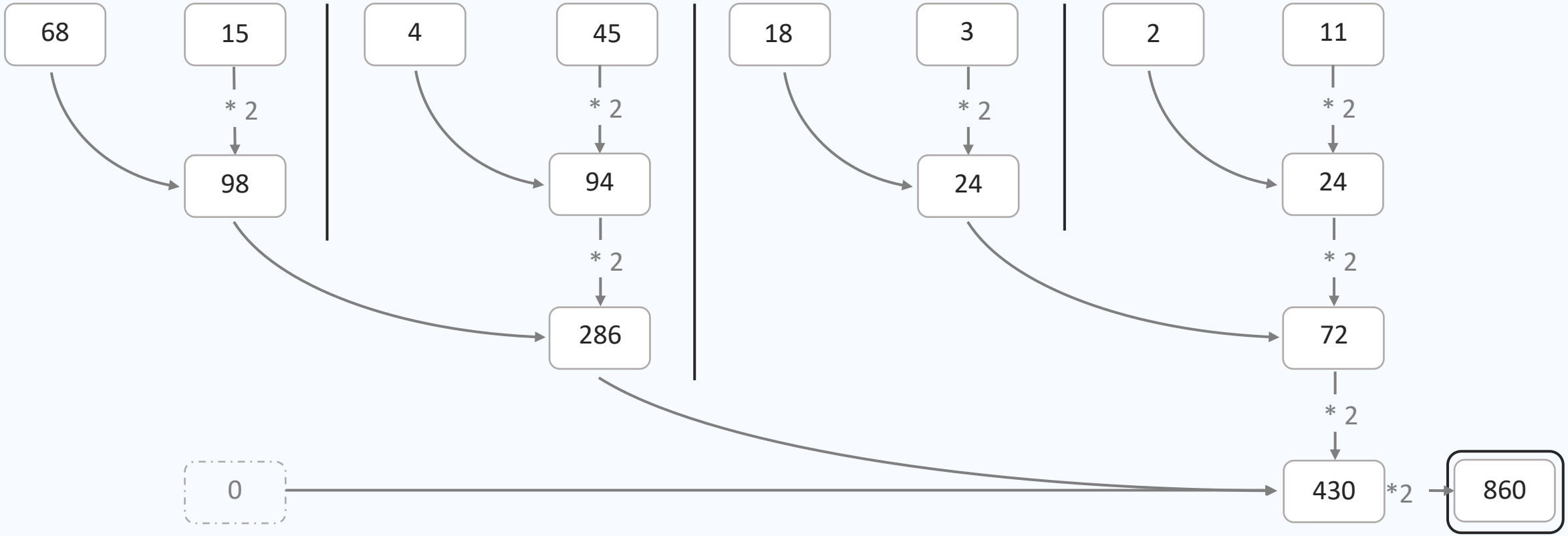
reduce walktrough



reduce walktrough



reduce walktrough



0 + 2*(68 + 2*15 + 2*(4 + 2*45) + 2*(18 + 2*3 + 2*(2 + 2* 11))) != 0 + ((((((2*68 + 2*15) + 2*4) + 2*45) + 2*18) + 2*3) + 2*2) + 2* 11

accumulate vs reduce

```
vector<int> counts {68, 15, 4, 45, 18, 3, 2, 11};
int         growth_factor = 2;

accumulate(counts.begin(), counts.end(), 0,
    [growth_factor](const auto first, const auto second){
        return first + second * growth_factor;
});
```

output: **322**

```
reduce(execution::par_unseq, counts.begin(), counts.end(), 0,
    [growth_factor](const auto first, const auto second){
        return first + second * growth_factor;
});
```

output: **716**

accumulate vs reduce - fixed

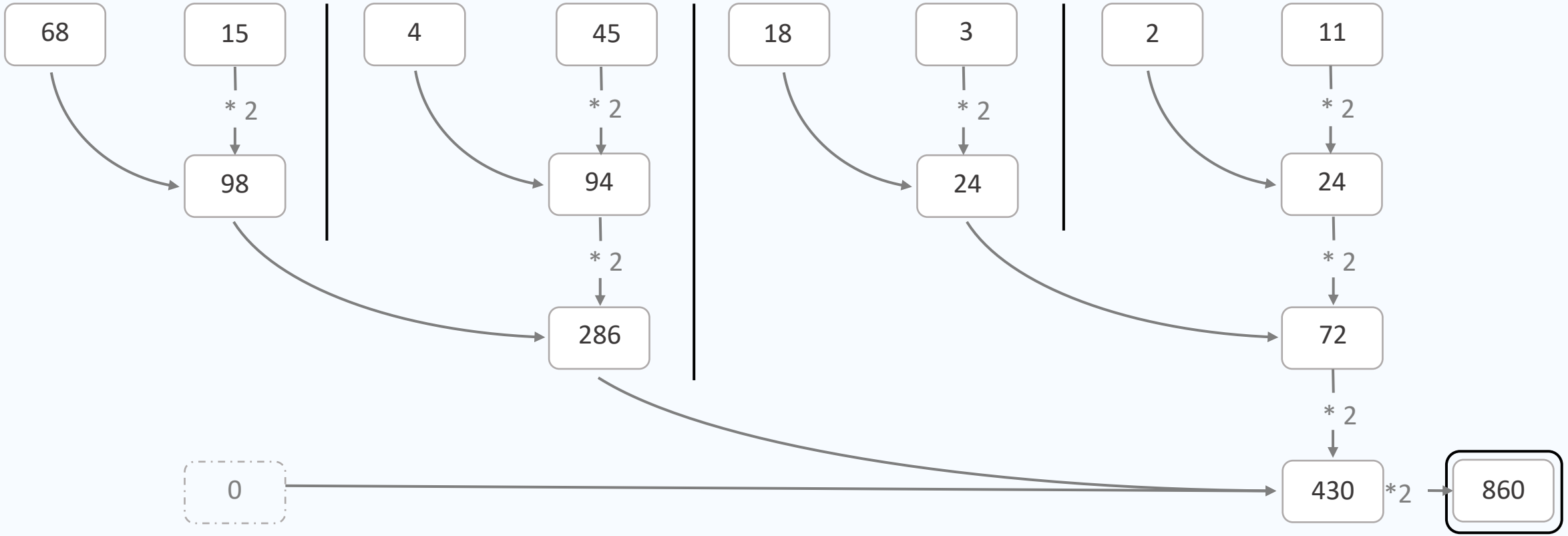
```
vector<int> counts {68, 15, 4, 45, 18, 3, 2, 11};
int         growth_factor = 2;

accumulate(counts.begin(), counts.end(), 0,
    [growth_factor](const auto first, const auto second){
        return first + second * growth_factor;
});
```

output: **332**

```
transform_reduce(execution::par_unseq, counts.begin(), counts.end(), 0, std::plus{},
    [growth_factor](const auto item){
        return item * growth_factor;
});
```

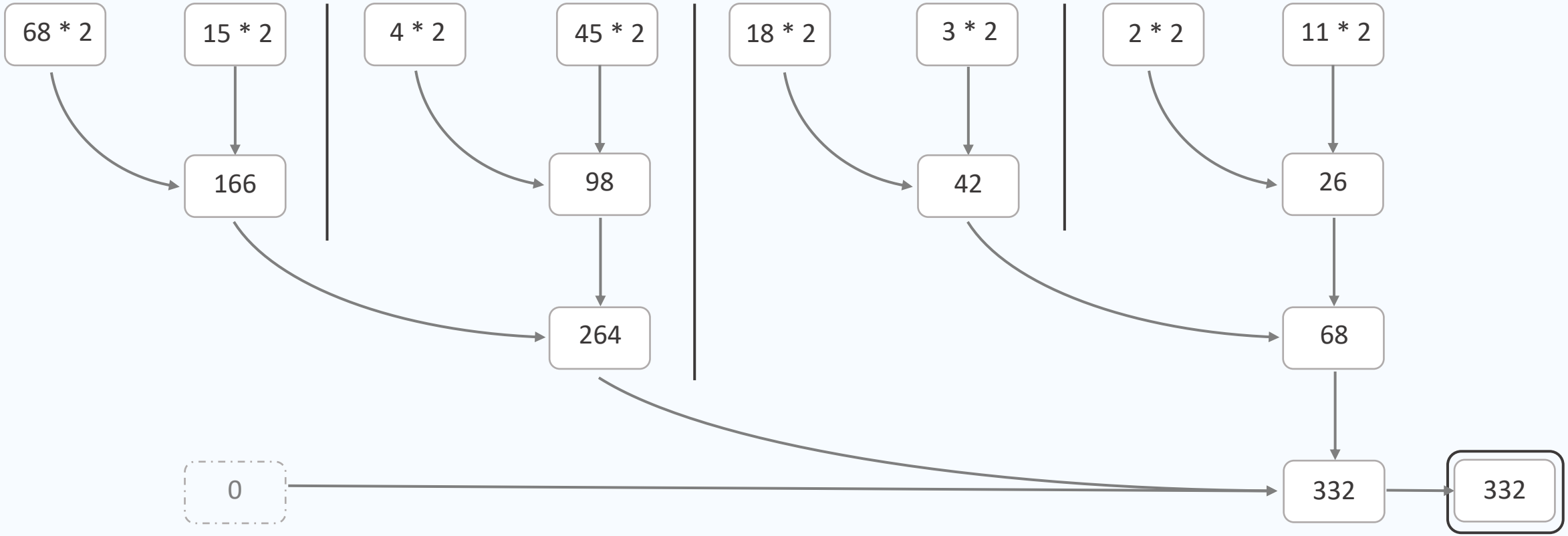
output: **332**

reduce walktrough



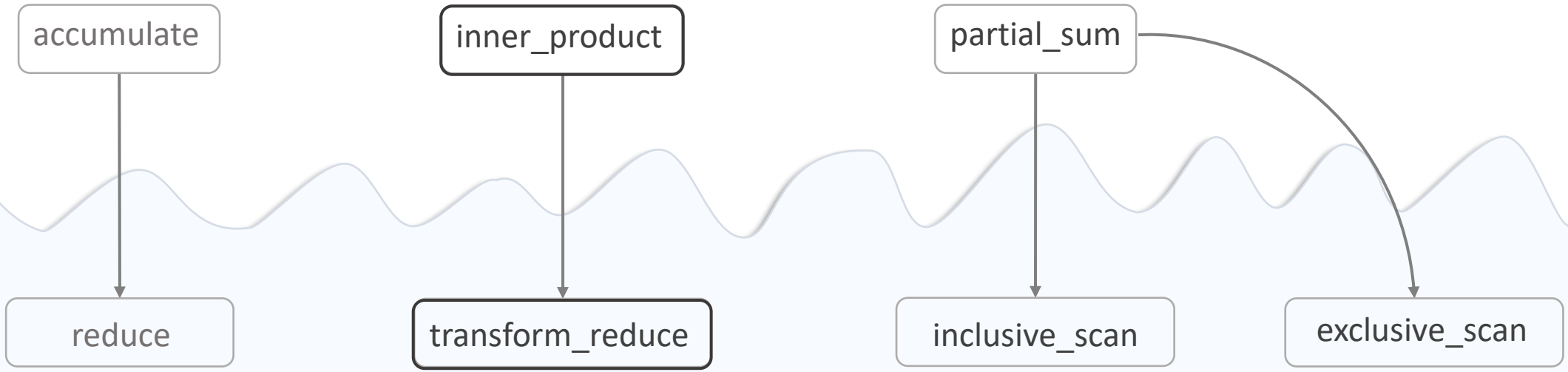
0 + (68 + 2*15 + 2*(4 + 2*45) + 2*(18 + 2*3 + 2*(2 + 2* 11)))*2 != 0 + 2*68 + 2*15 + 2*4 + 2*45 + 2*18 + 2*3 + 2*2 + 2* 11

transform_reduce walktrough



0 + (136 + 30 + (8 + 90) + (36 + 6 + (4 + 22))) == 0 + 2*68 + 2*15 + 2*4 + 2*45 + 2*18 + 2*3 + 2*2 + 2* 11

New algorithms introduced



inner_product vs transform_reduce

```
vector<int> weights {1, 2, 3, 4, 5, 6, 7, 8};
vector<int> counts {68, 15, 4, 45, 18, 3, 2, 11};

inner_product(counts.begin(), counts.end(), weights.begin(), 0);
```

output: **500**

```
transform_reduce(execution::par_unseq, counts.begin(), counts.end(), weights.begin(), 0);
```

output: **500**

inner_product vs transform_reduce

```
vector<int> weights {1, 2, 3, 4, 5, 6, 7, 8};
vector<int> counts {68, 15, 4, 45, 18, 3, 2, 11};

inner_product(counts.begin(), counts.end(), weights.begin(), 0,
              std::plus{}, std::multiply{});
```

output: **500**

```
transform_reduce(execution::par_unseq, counts.begin(), counts.end(), weights.begin(), 0,
                 std::plus{}, std::multiply{});
```

output: **500**

inner_product vs transform_reduce

```
vector<float> weights {1e-6f, 2.1e-5f, 3.f, 43.3f, 5.1e3f, 6.5e4f, 7.7e6f, 1.5e8f};
vector<int>   counts {68, 15, 4, 45, 18, 3, 2, 11};

inner_product(counts.begin(), counts.end(), weights.begin(), 0);
```

output: **1 665 688 704 g**

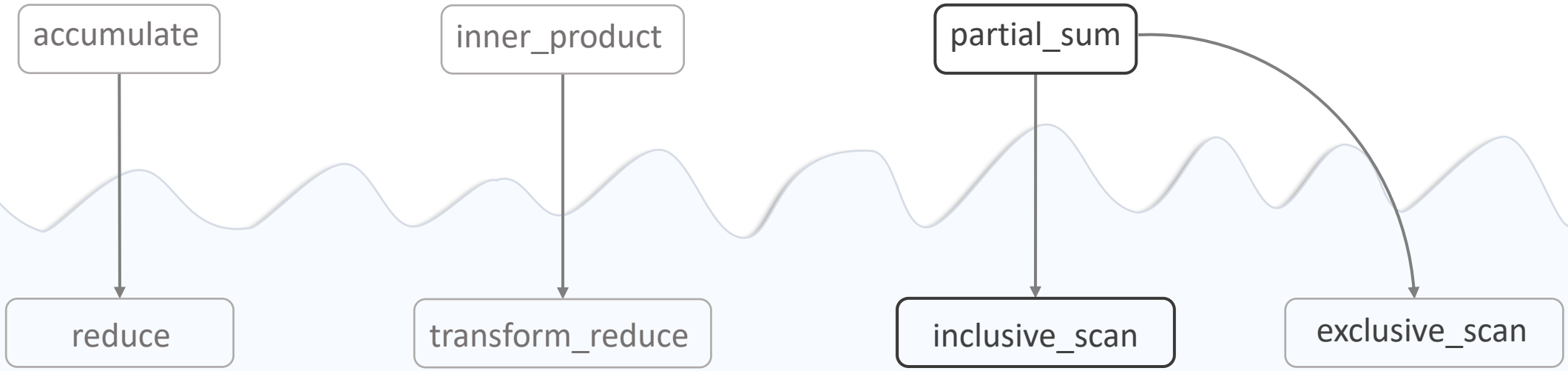
output with double: **1 665 688 761 g**

```
transform_reduce(execution::par_unseq, counts.begin(), counts.end(), weights.begin(), 0);
```

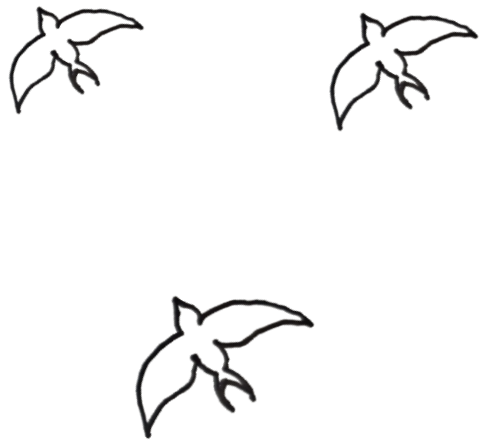
output: **1 665 688 832 g**

output with double: **1 665 688 761 g**

New algorithms introduced



partial_sum vs inclusive_scan



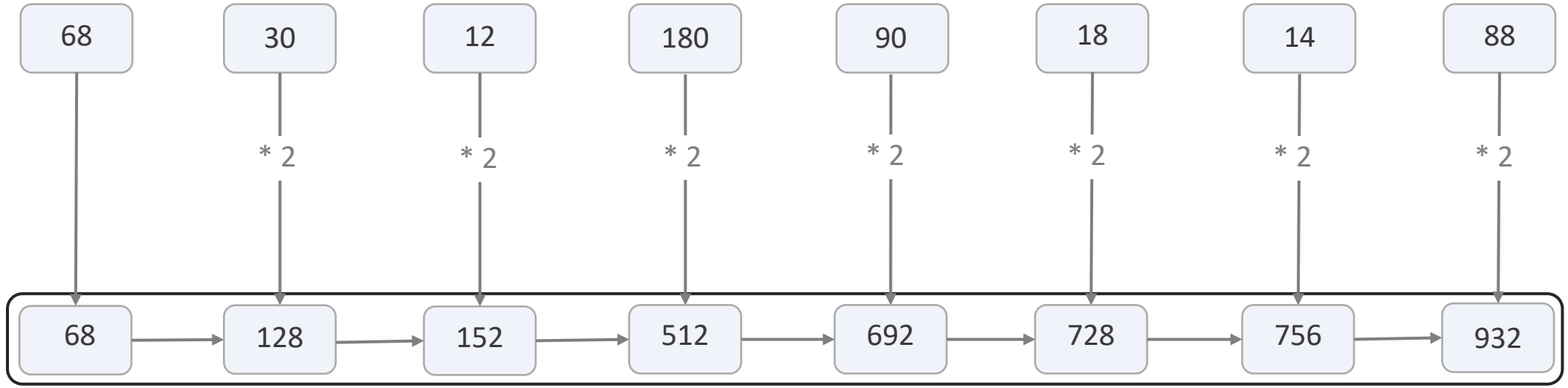
```
array<int, 8> mass {68, 30, 12, 180, 90, 18, 14, 88};
array<int, 8> food_mass;
int           growth_factor = 2;

partial_sum(mass.begin(), mass.end(), food_mass.begin(),
    [growth_factor](auto& smaller_food, auto& equal_food){
        return equal_food * growth_factor + smaller_food;
});
```

output: **68, 128, 152, 512, 692, 728, 756, 932**

```
inclusive_scan(execution::par_unseq, mass.begin(), mass.end(), food_mass.begin(),
    [growth_factor](auto& smaller_food, auto& equal_food){
        return equal_food * growth_factor + smaller_food;
});
```

output: **68, 128, 176, 872, 1616, 1376, 1080, 1884**

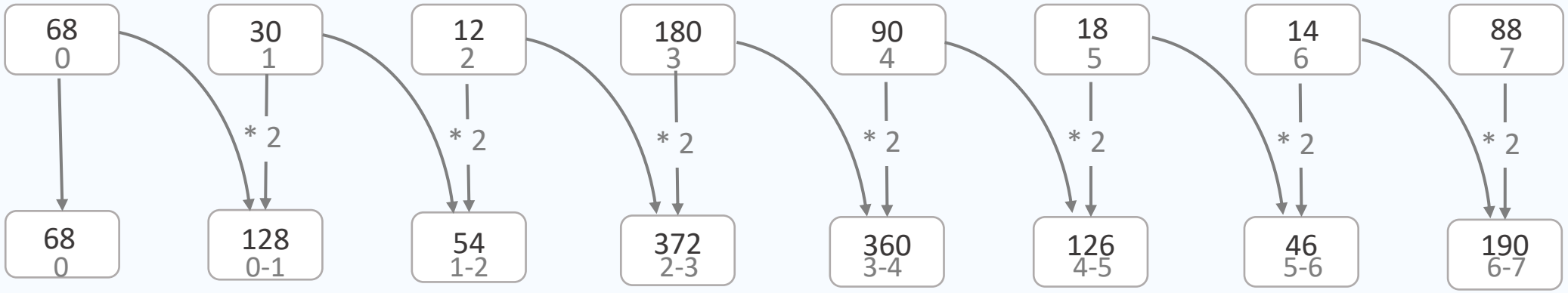
partial_sum walktrough



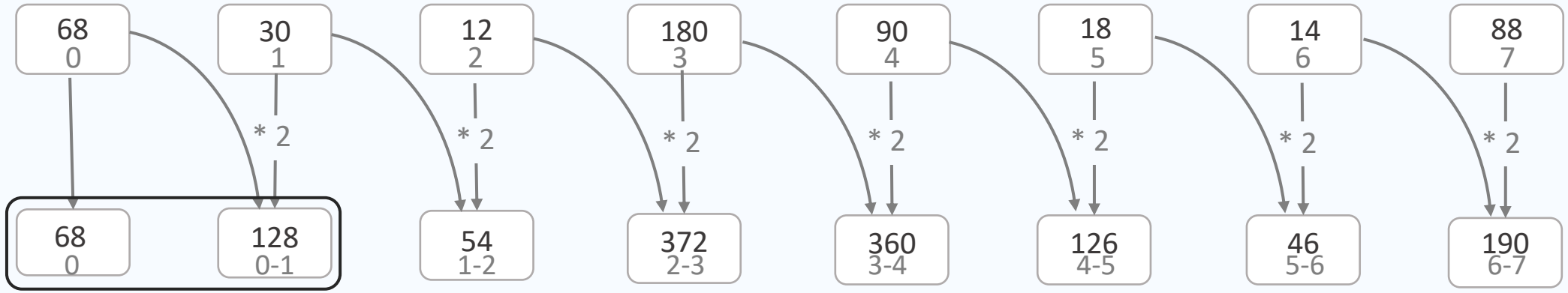
naïve inclusive_scan walktrough

68	30	12	180	90	18	14	88
0	1	2	3	4	5	6	7

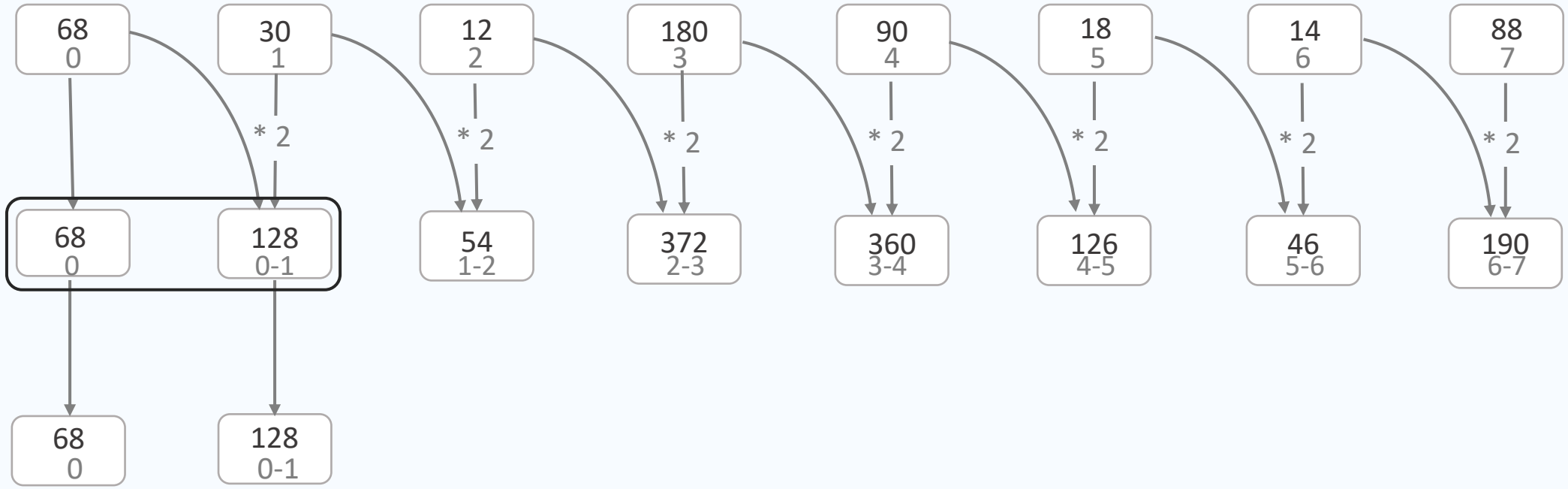
naïve inclusive_scan walktrough



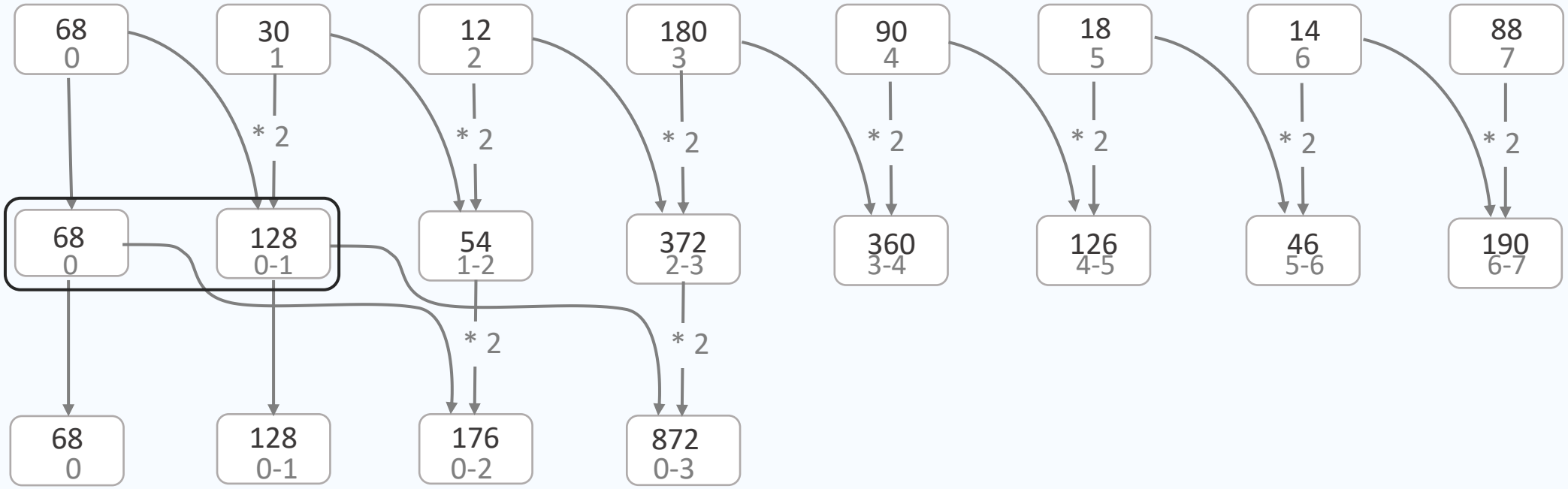
naïve inclusive_scan walktrough



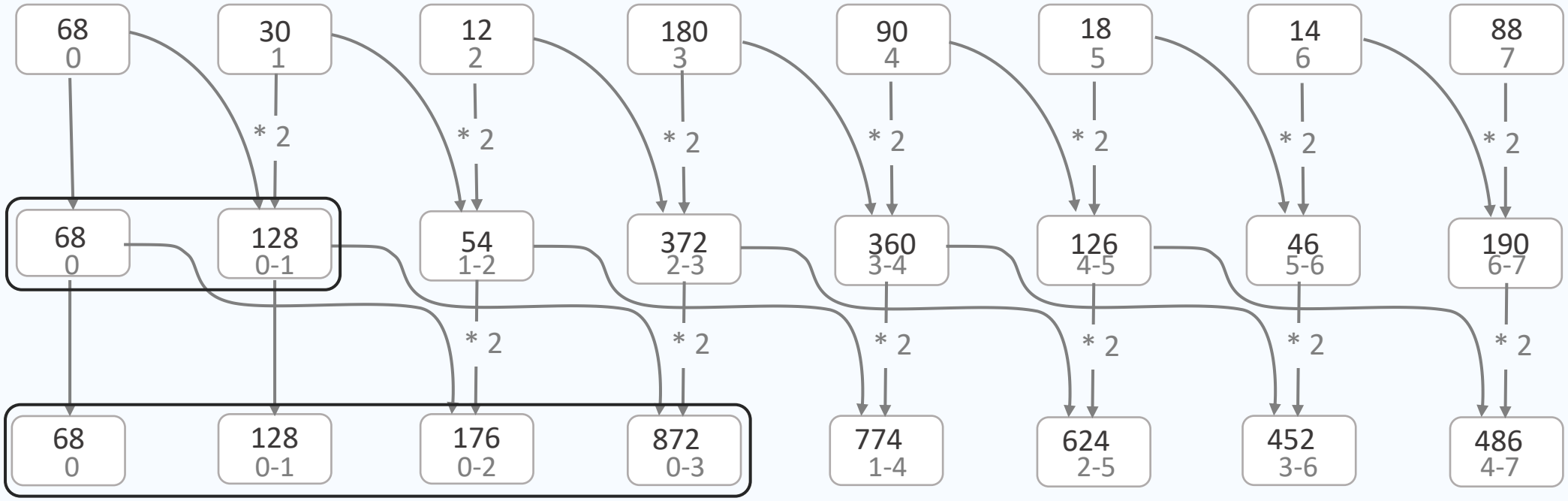
naïve inclusive_scan walktrough



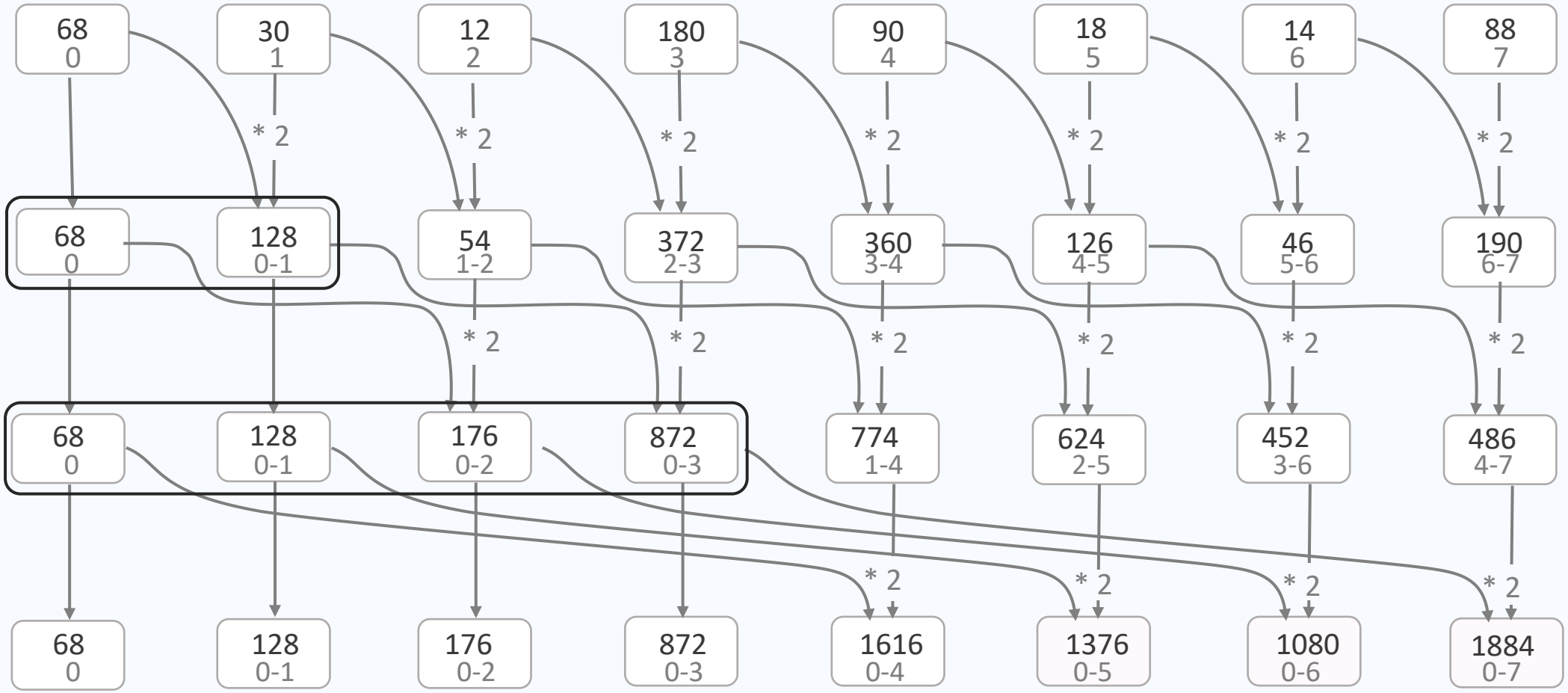
naïve inclusive_scan walktrough



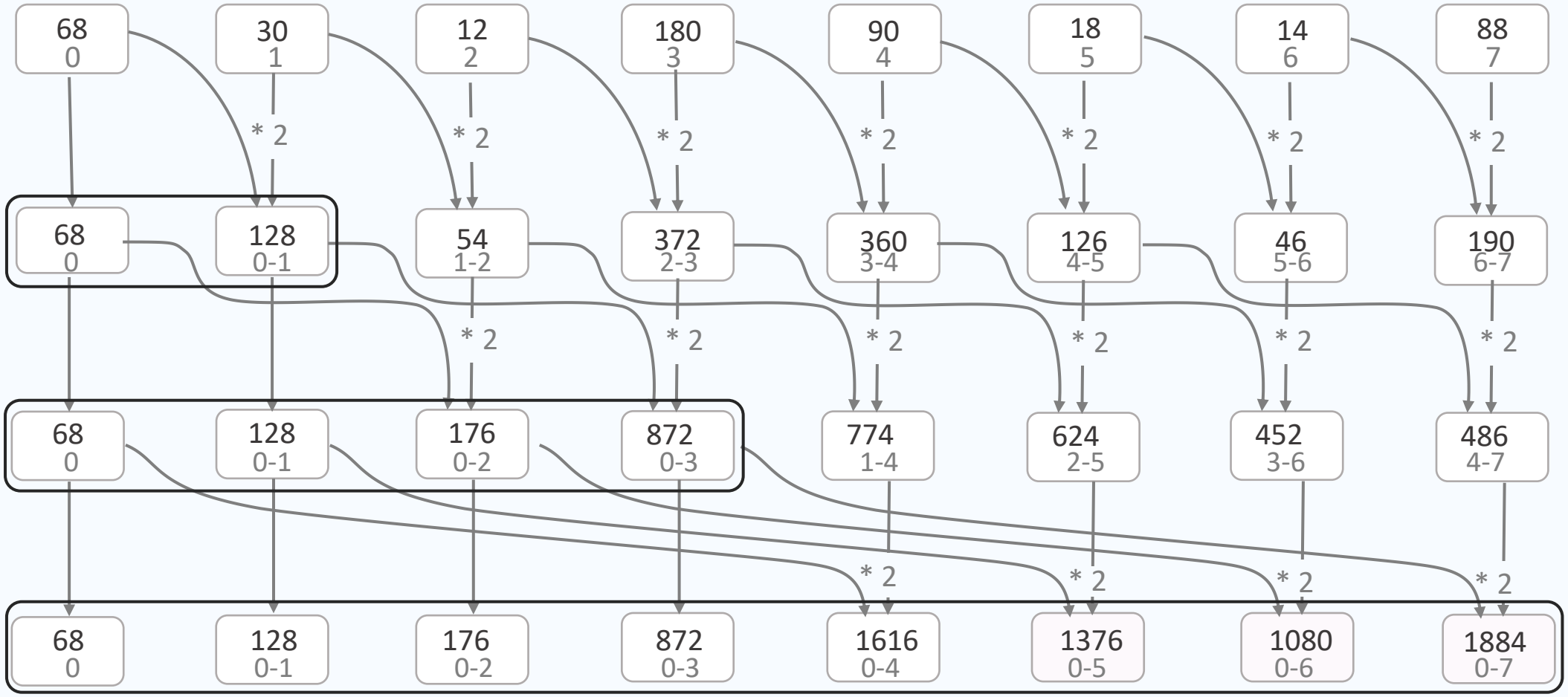
naïve inclusive_scan walktrough



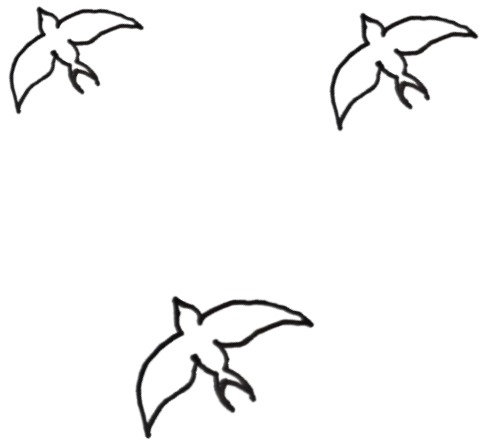
naïve inclusive_scan walktrough



naïve inclusive_scan walktrough



partial_sum vs inclusive_scan



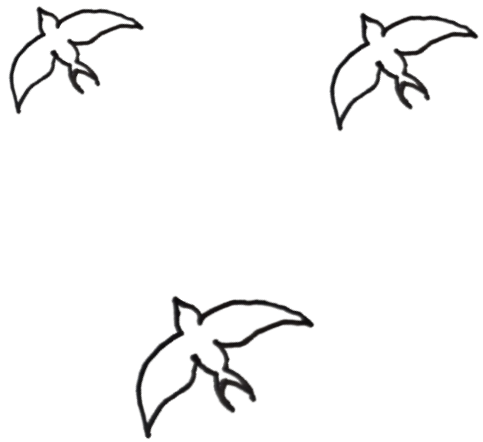
```
array<int, 8> mass {68, 30, 12, 180, 90, 18, 14, 88};
array<int, 8> food_mass;
int           growth_factor = 2;

partial_sum(mass.begin(), mass.end(), food_mass.begin(),
    [growth_factor](auto& smaller_food, auto& equal_food){
        return equal_food * growth_factor + smaller_food;
});
```

output: **68, 128, 152, 512, 692, 728, 756, 932**

```
inclusive_scan(execution::par_unseq, mass.begin(), mass.end(), food_mass.begin(),
    [growth_factor](auto& smaller_food, auto& equal_food){
        return equal_food * growth_factor + smaller_food;
});
```

output: **68, 128, 176, 872, 1616, 1376, 1080, 1884**

partial_sum vs inclusive_scan



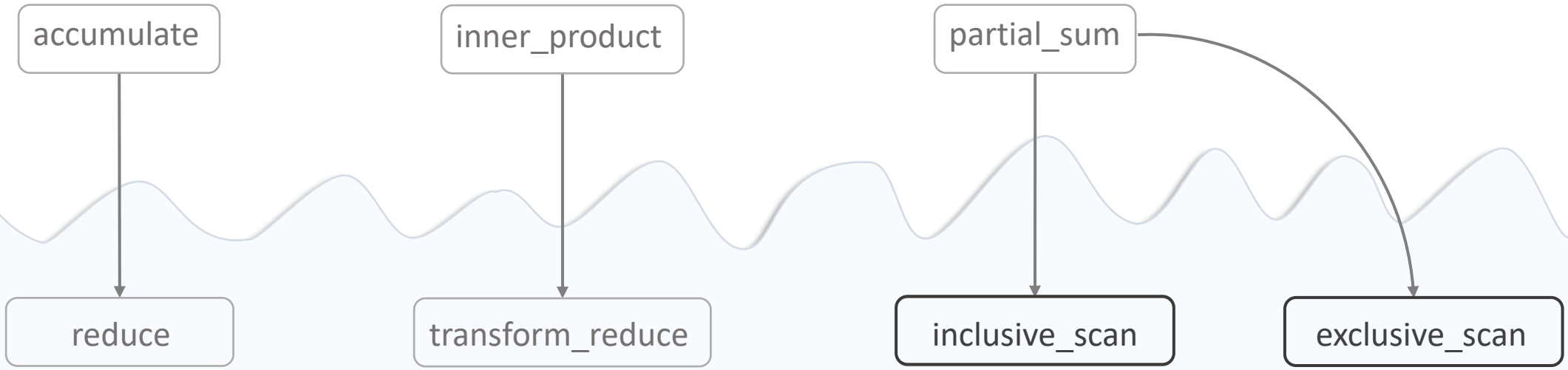
```
array<int, 8> mass {68, 30, 12, 180, 90, 18, 14, 88};
array<int, 8> food_mass;
int           growth_factor = 2;

partial_sum(mass.begin(), mass.end(), food_mass.begin(),
    [growth_factor](auto& smaller_food, auto& equal_food){
        return equal_food * growth_factor + smaller_food;
});
```

output: **68, 128, 152, 512, 692, 728, 756, 932**

```
transform_inclusive_scan(execution::par_unseq, mass.begin(), mass.end(), food_mass.begin(),
    std::plus{},
    [growth_factor](auto& cnt){
        return cnt * growth_factor;
});
```

output: **136, 196, 220, 580, 760, 796, 824, 1000**

New algorithms introduced



inclusive_scan vs exclusive_scan

```
array<int, 8> mass {136, 60, 24, 360, 180, 36, 28, 176};
array<int, 8> food_mass;

inclusive_scan(mass.begin(), mass.end(), food_mass.begin());
```

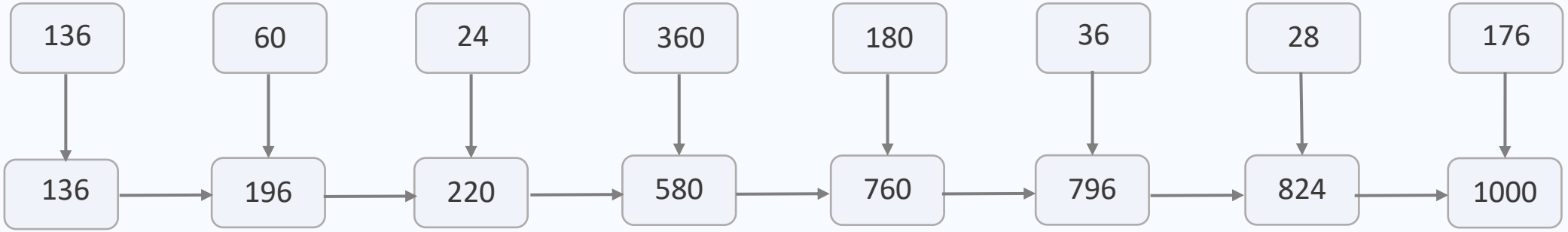
output: **136, 196, 220, 580, 760, 796, 824, 1000**

```
exclusive_scan(mass.begin(), mass.end(), food_mass.begin(), 0);
```

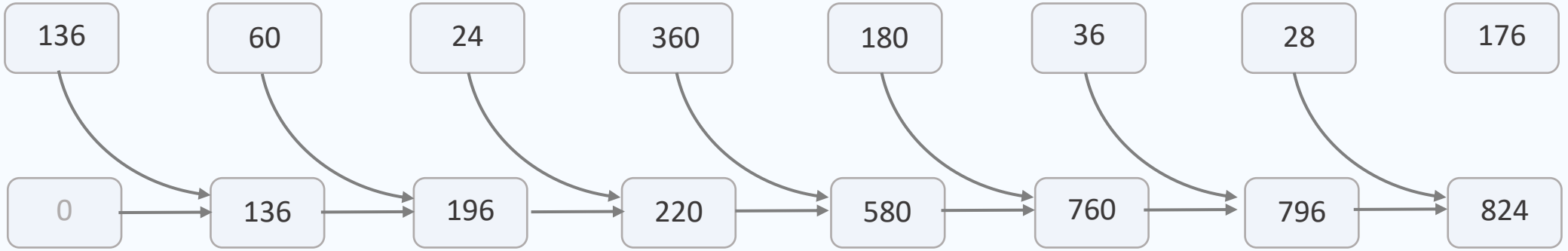
output: **0, 136, 196, 220, 580, 760, 796, 824**

inclusive_scan vs exclusive_scan

inclusive_scan:



exclusive_scan:



partial_sum vs inclusive_scan

```
array<int, 8> mass {68, 30, 12, 180, 90, 18, 14, 88};
array<int, 8> food_mass;
int           growth_factor = 2;

partial_sum(mass.begin(), mass.end(), food_mass.begin(),
    [growth_factor](auto& smaller_food, auto& equal_food){
        return equal_food * growth_factor + smaller_food;
});
```

output: **68, 128, 152, 512, 692, 728, 756, 932**

```
transform_inclusive_scan(execution::par_unseq, mass.begin(), mass.end(), food_mass.begin(),
    std::plus{},
    [growth_factor](auto& cnt){
        return cnt * growth_factor;
});
```

output: **136, 196, 220, 580, 760, 796, 824, 1000**

partial_sum vs exclusive_scan

```
array<int, 8> mass {68, 30, 12, 180, 90, 18, 14, 88};
array<int, 8> food_mass;
int           growth_factor = 2;

partial_sum(mass.begin(), mass.end(), food_mass.begin(),
    [growth_factor](auto& smaller_food, auto& equal_food){
        return equal_food * growth_factor + smaller_food;
});
```

output: **68, 128, 152, 512, 692, 728, 756, 932**

```
transform_exclusive_scan(execution::par, mass.begin() + 1, mass.end(), food_mass.begin(),
    mass[0], std::plus{},
    [growth_factor](auto& cnt){
        return cnt * growth_factor;
});
```

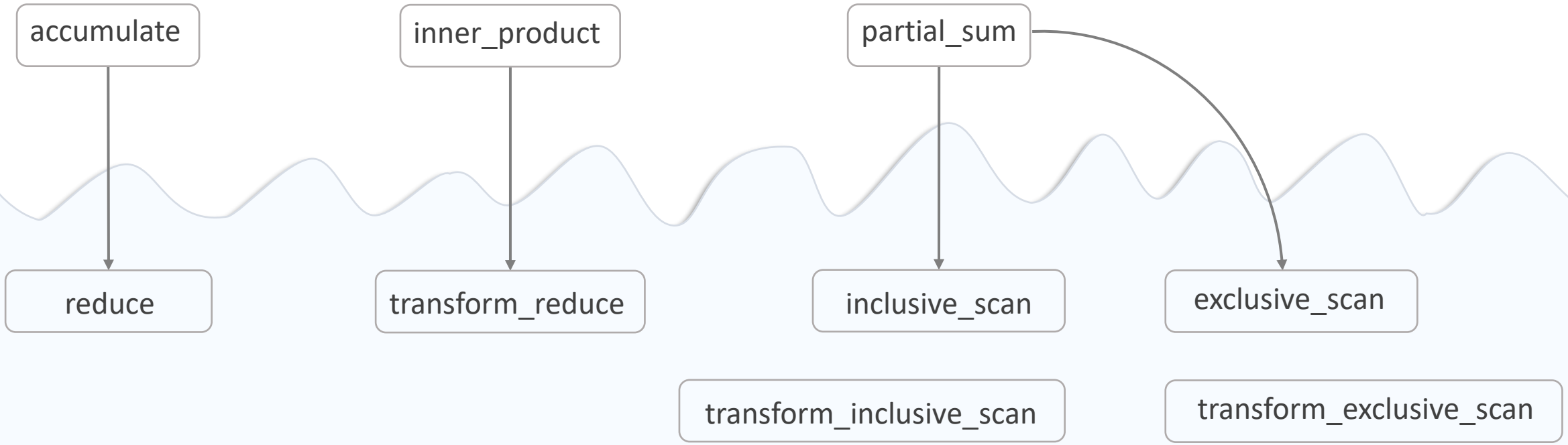
output: **68, 128, 152, 512, 692, 728, 756, ?**

New algorithms introduced

Always prefer **reduce**, **inclusive_scan**, **exclusive scan** and their transformed version if possible

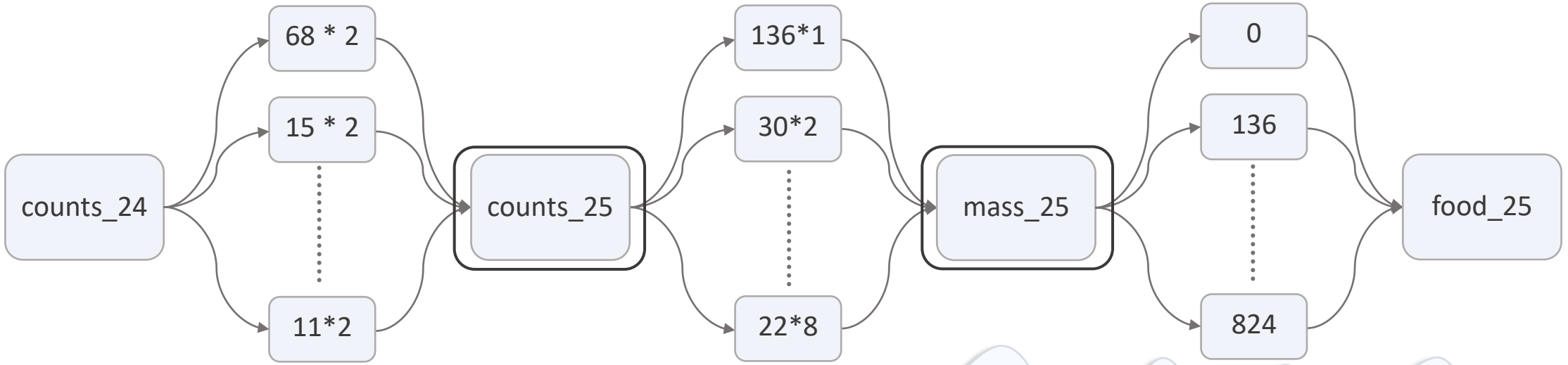
Use **accumulate**, **inner_product** and **partial_sum** only for non-associative operations

Be aware of floating point non-associativity

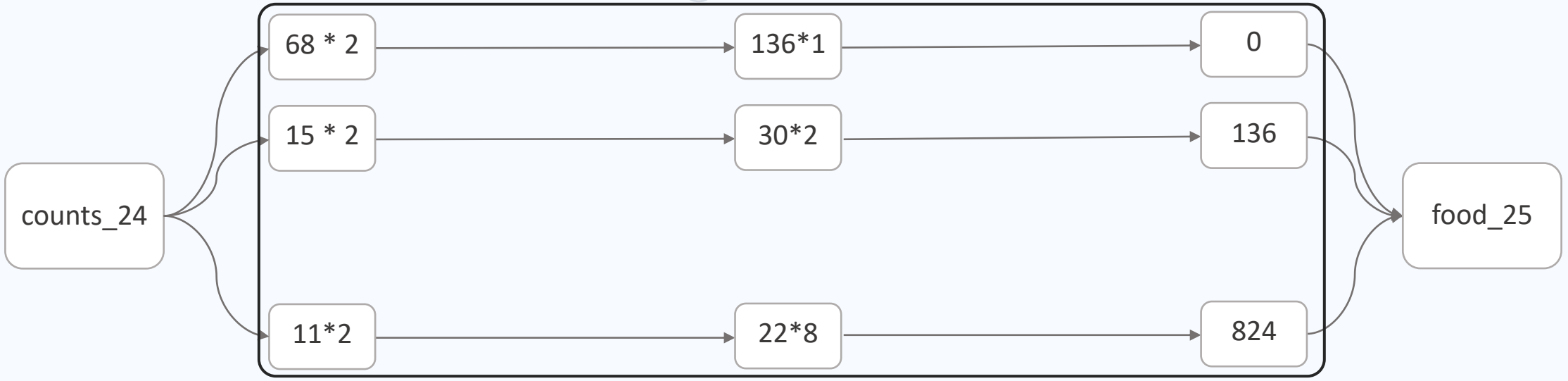


Unwanted synchronization points

```
array<int, 8> count_24   {68, 15, 4, 45, 18, 3, 2, 11};
array<int, 8> weights    {1, 2, 3, 4, 5, 6, 7, 8};

transform(par_unseq, counts_24.begin(), counts_24.end(), counts_25.begin(), [](auto cnt){return cnt * 2;});
transform(par_unseq, counts_25.begin(), counts_25.end(), weights.begin(), mass_25.begin(), multiplies{});
exclusive_scan(par_unseq, mass_25.begin(), mass_25.end(), food_25.begin(), 0);
```

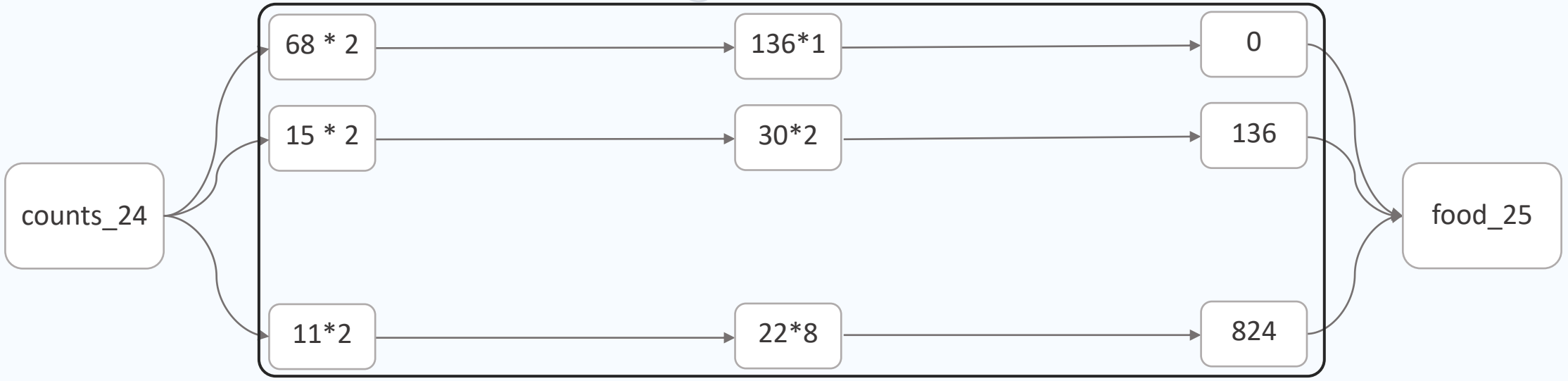


Unwanted synchronization points




```
auto mass_25 = views::transform([&](auto item){return item * 2;}) |
               views::zip_transform(multiplies{}, counts_25, weights);


exclusive_scan(par_unseq, mass_25.begin(), mass_25.end(), food_25.begin(), 0);
```

Unwanted synchronization points




```
auto mass_24 = views::zip_transform(multiplies{}, counts_24, weights);


transform_exclusive_scan(par_unseq, mass_24.begin(), mass_24.end(), food_25.begin(), 0,
                         plus{}, ([&](auto item){return item * 2;});
```

Parallel algorithms

All the rest of the standard library algorithms could be used in parallel version

Some of them may require additional allocation

And some may not be implemented so far

But don’t worry; the worst you can get is sequential performance you already had before

All you need to do is to add an execution policy

Execution policy

The first param of each parallel algorithm

Defines how much is it possible to parallelize code inside algorithm body

For the best results, go for the strongest policy you can

If you are unable to use strongest one, maybe you won’t gain anything from parallelization

Always measure performance before and after parallelization

Execution policy

std::execution::seq for sequenced policy

std::execution::par for parallel policy

std::execution::unseq for unsequenced policy

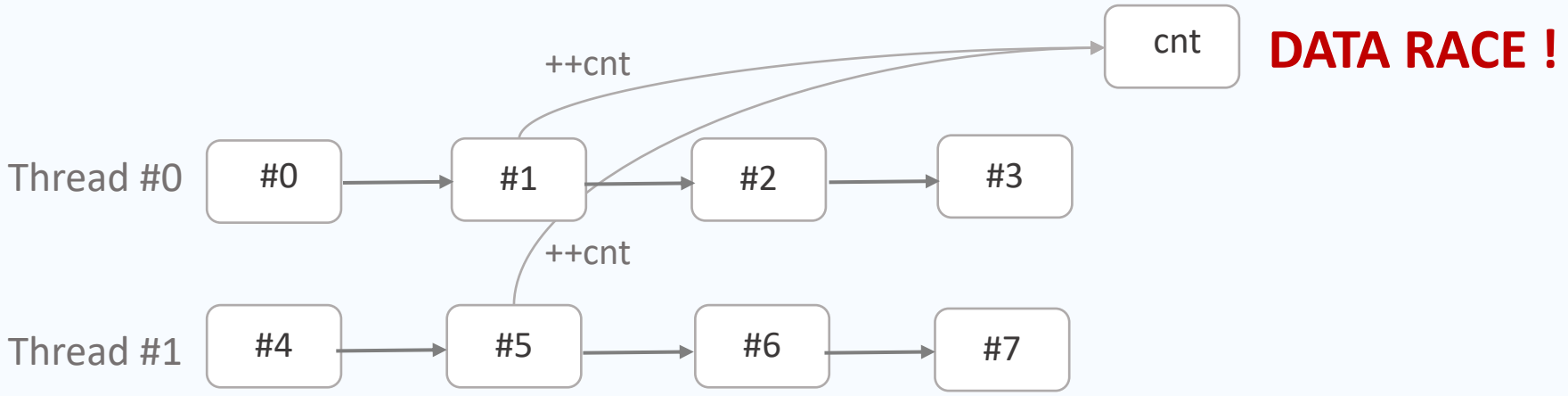
std::execution::par_unseq for parallel unsequenced policy

execution::seq

- Code is executed sequentially
- All computations are executed on single thread one by one
- No parallelization at all
- If there is no policy defined, it runs in execution::seq

execution::par

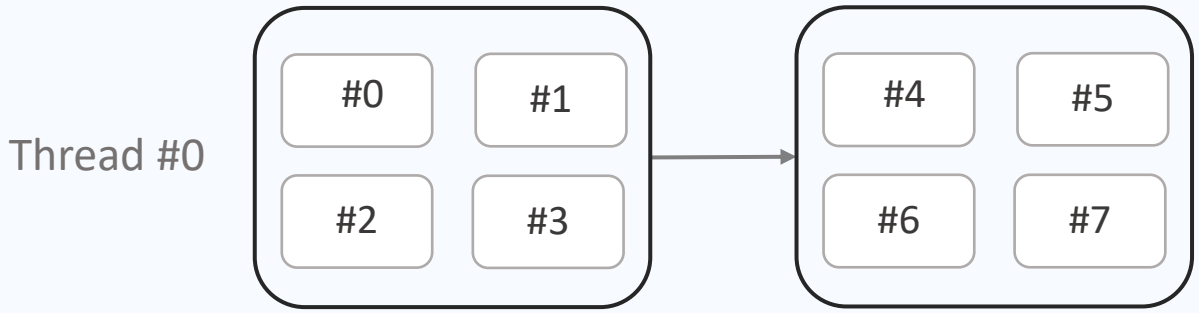
- Code is executed on multiple threads
- Order in which are elements processed is not predictable
- Shared data needs to be protected as it may be accessed by different threads concurrently
- If blocking protection is needed it may not perform better than the sequential approach – measure it



execution::unseq

- Code is executed in single thread
- The whole vector of data is executed together – their instruction are in an unpredictable order
- No synchronization is allowed
- No, not even atomics

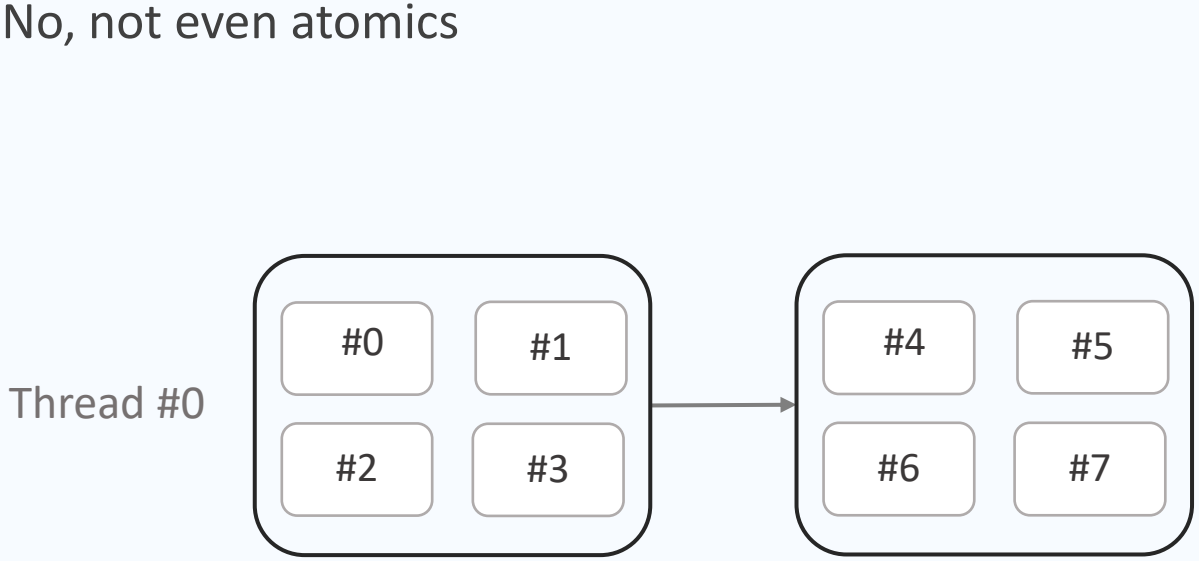
cnt



lock(cnt); ++cnt; unlock(cnt);

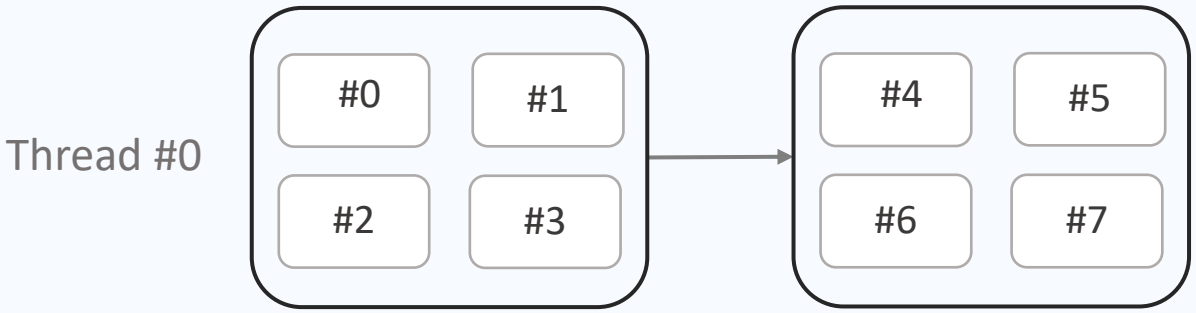
execution::unseq

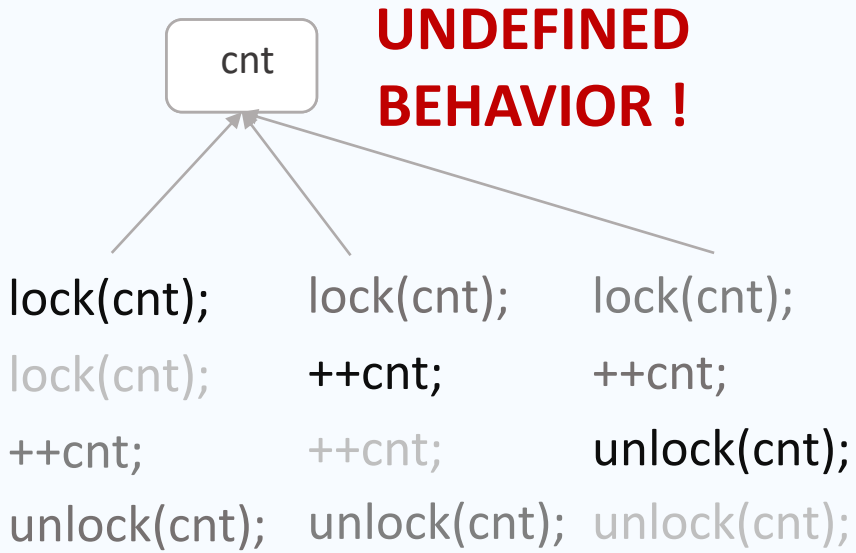
- Code is executed in single thread
- The whole vector of data is executed together – their instruction are in an unpredictable order
- No synchronization is allowed
- No, not even atomics



execution::unseq

- Code is executed in single thread
- The whole vector of data is executed together – their instruction are in an unpredictable order
- No synchronization is allowed
- No, not even atomics





execution::par_unseq

- Data is vectorized and executed on multiple threads
- The strongest possible version of parallelization
- Keeps the same limitation as unseq – no synchronization is allowed
- You should try to target this policy

cnt

DON'T EVEN THINK ABOUT IT

Plenty of fish in the sea – choose correct policy

```
struct Fish  { int pos; int velocity; };
vector<Fish> fishes{…};
auto         time_interval = 1;


for_each(execution::par_unseq, fishes.begin(), fishes.end(), [&](auto& fish)
{
    fish.pos += time_interval * fish.velocity;
});
```

Beware of shark – choose correct policy

```
struct Fish  { int pos; int velocity;};
vector<Fish> fishes{…};
auto         time_interval = 1;
atomic<bool> shark = false;  //shark is set somewhere else - this will signal his presence

for_each(execution::par, fishes.begin(), fishes.end(), [&](auto& fish)
{
    if(shark)
       fish.run_away();
    else
       fish.pos += time_interval * fish.velocity;
});
```

Look for a hideout - choose correct policy

```
struct Hideout {int pos; int slots; bool take_free_slot(); };
vector<Hideout> hideouts{…};

for_each(execution::par_unseq, fishes.begin(), fishes.end(), [&](auto& fish)
{
    auto& selected = closest_free_hideout();

    swim_for(std::abs(selected->pos - fish.pos) * fish.velocity);

    if(selected->take_free_slot())
       stay_hidden();
    else
       try_mimicry();
});
```

Look for a hideout - choose correct policy

```
struct Hideout {int atomic<int> slots; bool take_free_slot(); };
vector<Hideout> hideouts{…};

for_each(execution::par, fishes.begin(), fishes.end(), [&](auto& fish)
{
    auto& selected = closest_free_hideout();

    swim_for(std::abs(selected->pos - fish.pos) * fish.velocity);

    if(selected->take_free_slot())
       stay_hidden();
    else
       try_mimicry();
});
```

Become a leader – unseq trap

```
bool has_leader = false;
mutex leaders_mutex;
vector<Fish> fishes;

for_each(execution::par_unseq, fishes.begin(), fishes.end(), [&](auto& fish)
{
    unique_lock lock(leaders_mutex);
    while(has_leader){
        lock.unlock(); swim_around(); lock.lock();
    }
    has_leader = true;
    lock.unlock();

    live_leaders_life();

    lock.lock();
    has_leader = false;
});
```

Become a leader – unseq trap

```
bool has_leader = false;
mutex leaders_mutex;
vector<Fish> fishes;

for_each(execution::par_unseq, fishes.begin(), fishes.end(), [&](auto& fish)
{
    unique_lock lock(leaders_mutex);
    while(has_leader){
        lock.unlock(); swim_around(); lock.lock();
    }
    has_leader = true;
    lock.unlock();

    live_leaders_life();

    lock.lock();
    has_leader = false;
});
```

```
unique_lock lock(leaders_mutex);
unique_lock lock(leaders_mutex);
unique_lock lock(leaders_mutex);
unique_lock lock(leaders_mutex);
```

Become a leader – unseq trap

```
bool has_leader = false;
mutex leaders_mutex;
vector<Fish> fishes;

for_each(execution::par_unseq, fishes.begin(), fishes.end(), [&](auto& fish)
{
    unique_lock lock(leaders_mutex);
    while(has_leader){
        lock.unlock(); swim_around(); lock.lock();
    }
    has_leader = true;
    lock.unlock();

    live_leaders_life();

    lock.lock();
    has_leader = false;
});
```

```
unique_lock lock(leaders_mutex);
unique_lock lock(leaders_mutex);
unique_lock lock(leaders_mutex);
unique_lock lock(leaders_mutex);
```

UNDEFINED BEHAVIOR !

Become a leader – atomic unseq trap

```
atomic<bool> has_leader = false;
vector<Fish> fishes;


for_each(execution::par_unseq, fishes.begin(), fishes.end(), [&](auto& fish)
{
    while(has_leader.exchange(true))
        swim_around();

    live_leaders_life();

    has_leader = false;
});
```

Become a leader – atomic unseq trap

```
atomic<bool> has_leader = false;
vector<Fish> fishes;


for_each(execution::par_unseq, fishes.begin(), fishes.end(), [&](auto& fish)
{
    while(has_leader.exchange(true))
        swim_around();

    live_leaders_life();

    has_leader = false;
});
```

```
while(has_leader.exchange(true))
    swim_around();

live_leaders_life();

has_leader = false;
```

Become a leader – atomic unseq trap

```
atomic<bool> has_leader = false;
vector<Fish> fishes;


for_each(execution::par_unseq, fishes.begin(), fishes.end(), [&](auto& fish)
{
    while(has_leader.exchange(true))
        swim_around();

    live_leaders_life();

    has_leader = false;
});
```

```
while(has_leader.exchange(true))
    swim_around();

live_leaders_life();

while(has_leader.exchange(true))
    swim_around();

has_leader = false;
```

Become a leader – atomic unseq trap

```
atomic<bool> has_leader = false;
vector<Fish> fishes;


for_each(execution::par_unseq, fishes.begin(), fishes.end(), [&](auto& fish)
{
    while(has_leader.exchange(true))
        swim_around();

    live_leaders_life();

    has_leader = false;
});
```

```
while(has_leader.exchange(true))
    swim_around();

live_leaders_life();

while(has_leader.exchange(true))
    swim_around();

has_leader = false;
```

DEADLOCK!

Summary

Use standard library algorithms – your life will be easier

Use the strongest possible policy

Try to avoid data dependencies – synchronization impact is huge

Always measure your performance

Beware of non-associative operations

Deeper insight

Think Parallel (by Bryce Adelstein Lelbach)

The C++ Execution Model (by Bryce Adelstein Lelbach)

Portable floating-point calculations (by Guy Davidson)

Parallel STL https://github.com/llvm/llvm-project/tree/main/pstl

Thrust https://github.com/NVIDIA/cccl/tree/main/thrust

Sources

All examples and naive algorithms as presented

https://github.com/EmptySquareBubble/Stories_from_a_parallel_universe

Thank you!