Stories from a parallel universe

Jana Machutová

Do you feel sudden tension in your brain?

SEMAPHORE

JOIN

JTHREAD

BARRIER

WAIT

DEADLOCKS

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

PARALLEL ALGORITHMS

CRITICAL SECTION

LOCK

MEMORY ORDER MUTEX

NOTIFY ALL

DETACH

RACE CONDITION

FETCH

FENCES

LOCK-FREE

Standard library parallel algorithm

- Provides parallel version of most of standard library algorithms
- Introduced execution policy parameter – all the rest is same as before
- Easy switch between sequential and few types of parallel executions

- Well optimized solution to common problems
- No need to reinvent parallel wheels

- In standard since version 17 (with some changes in 20 and another on their way for future versions)

- Several new algorithms appeared
- Several old don’t have any parallel version

- Not available for ranges so far

No need to take care of any threads at all – it's handled automatically
Underlying thread management saves price of creating/destructing threads, split the load equally and don't create crazy amount of threads

Create new worlds

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

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

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

one honey bee hive contains 60,000 to 80,000 bees

Create new worlds

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

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

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(),
         [](const auto name){
                 swim_deep(name);
});
```

Thread #0 species [0 – 9 999]

Thread #1 species [10 000 – 19 999]

Thread #100 species [90 000 – 99 999]

one honey bee hive contains 60,000 to 80,000 bees

Why do you want to use parallel algorithms

- No need for direct manipulation with threads
- No need to care about threads lifetime
- Clever thread management is already part of the package

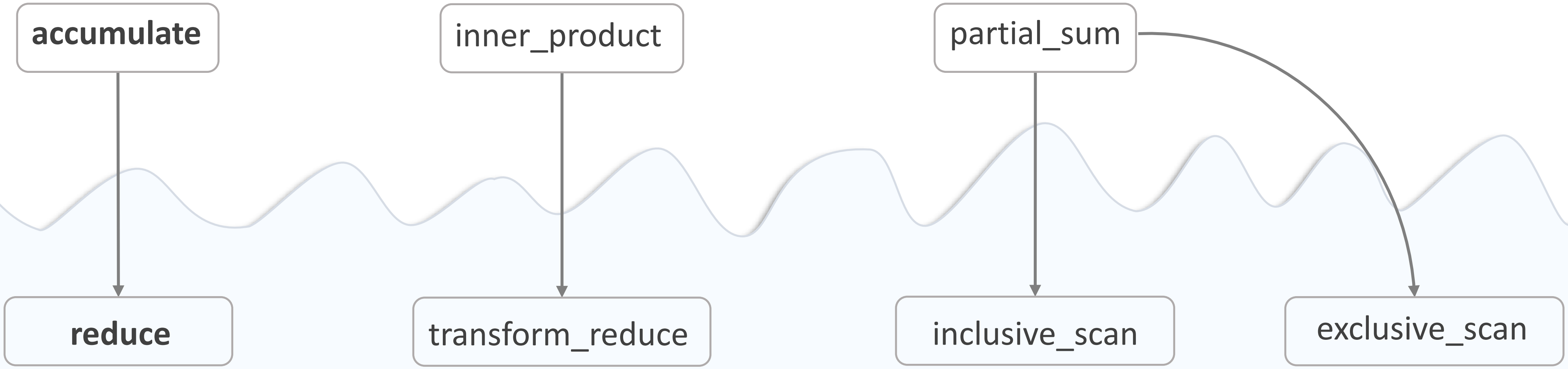
- Widely tested and optimized solution
- Easy to switch between different execution policies

- Improved readability
- Less error prone
- If you can use sequential version, you will be able to use parallel without any additional studies

Do not reinvent a wheel – use standard library as much as you can

New algorithms introduced

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



one honey bee hive contains 60,000 to 80,000 bees

accumulate vs reduce

First sight the only difference is possibility to add policy

Second sight try to modify operation to some custom

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

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

output: **166**

```
reduce(execution::par, species_count.begin(), species_count.end(), 0 , plus{});
```

output: **166**

one honey bee hive contains 60,000 to 80,000 bees

accumulate vs reduce

How many animals will be there in next generation with growth_factor 2

Non-associative reduction function makes mess

Have a look what's going on in both cases on next slides

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

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

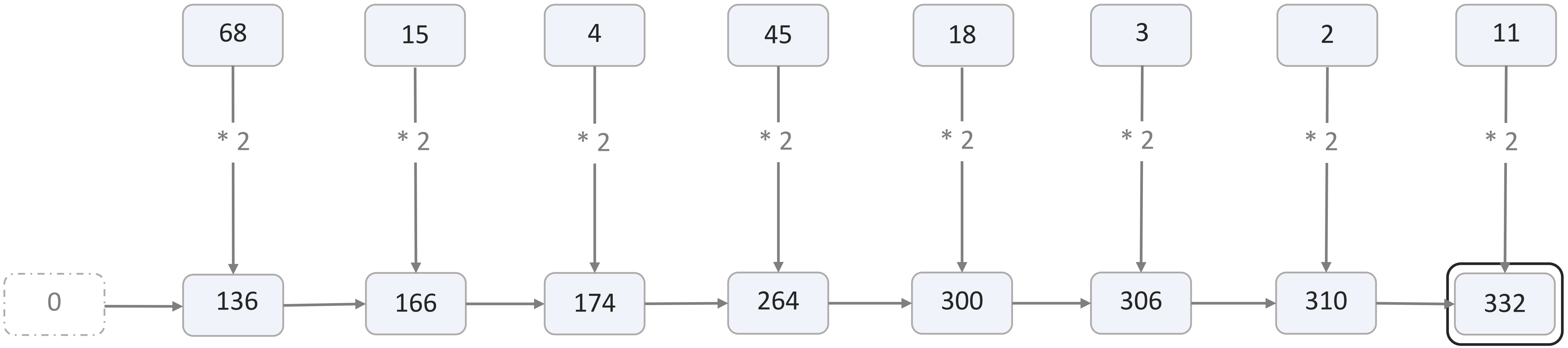
output: **332**

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

output: **716**

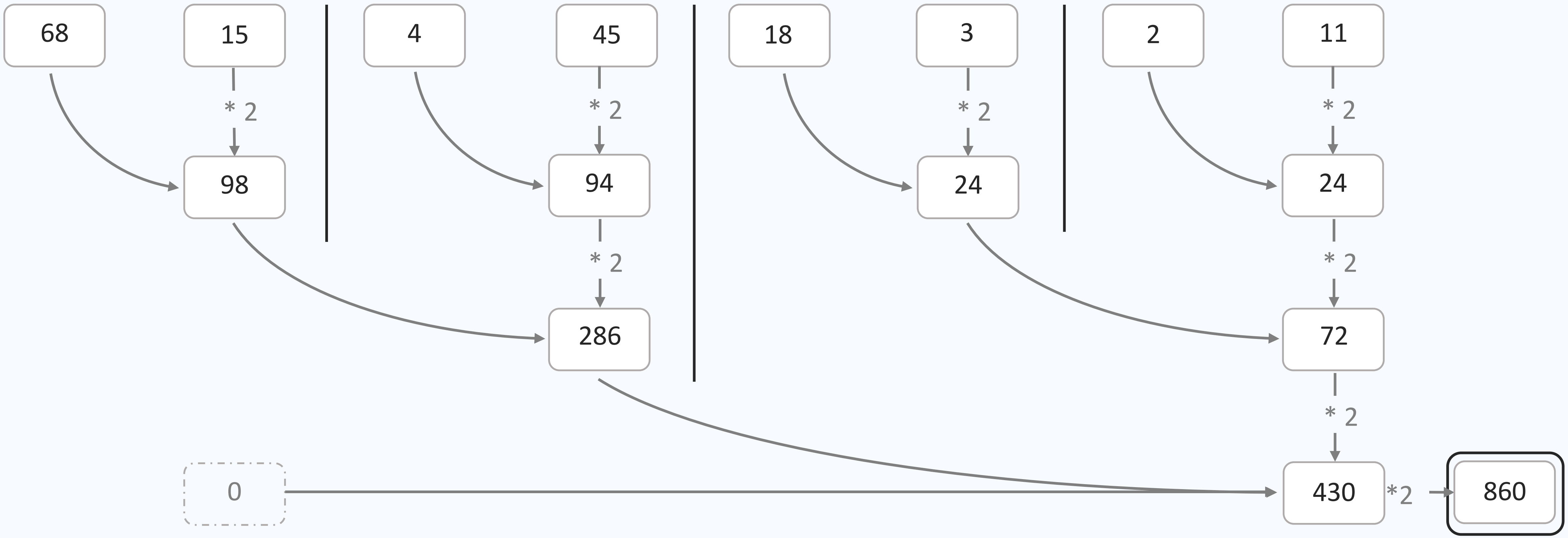
one honey bee hive contains 60,000 to 80,000 bees

accumulate walktrough



one honey bee hive contains 60,000 to 80,000 bees

reduce walktrough



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

one honey bee hive contains 60,000 to 80,000 bees

accumulate vs reduce

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

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

output: **322**

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

output: **716**

one honey bee hive contains 60,000 to 80,000 bees

We can fix it with transform_reduce - now our reduction is associative (trasformation doesn't need to be)

accumulate vs reduce - fixed

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

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

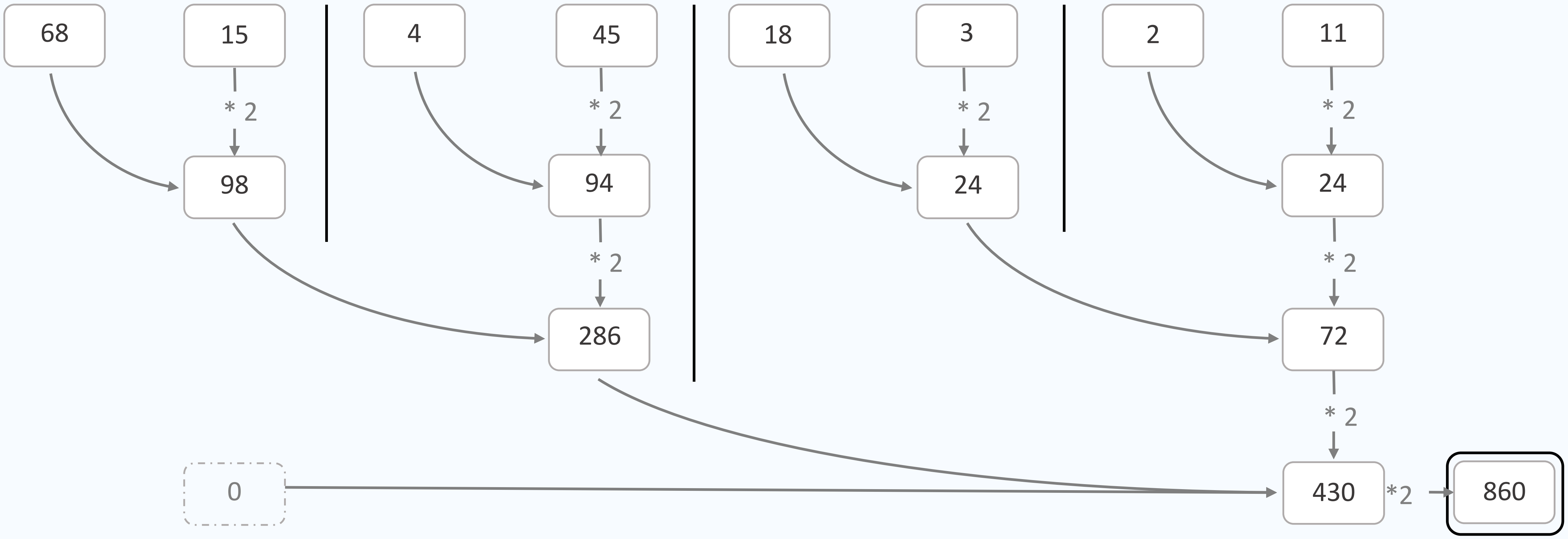
output: **332**

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

output: **332**

one honey bee hive contains 60,000 to 80,000 bees

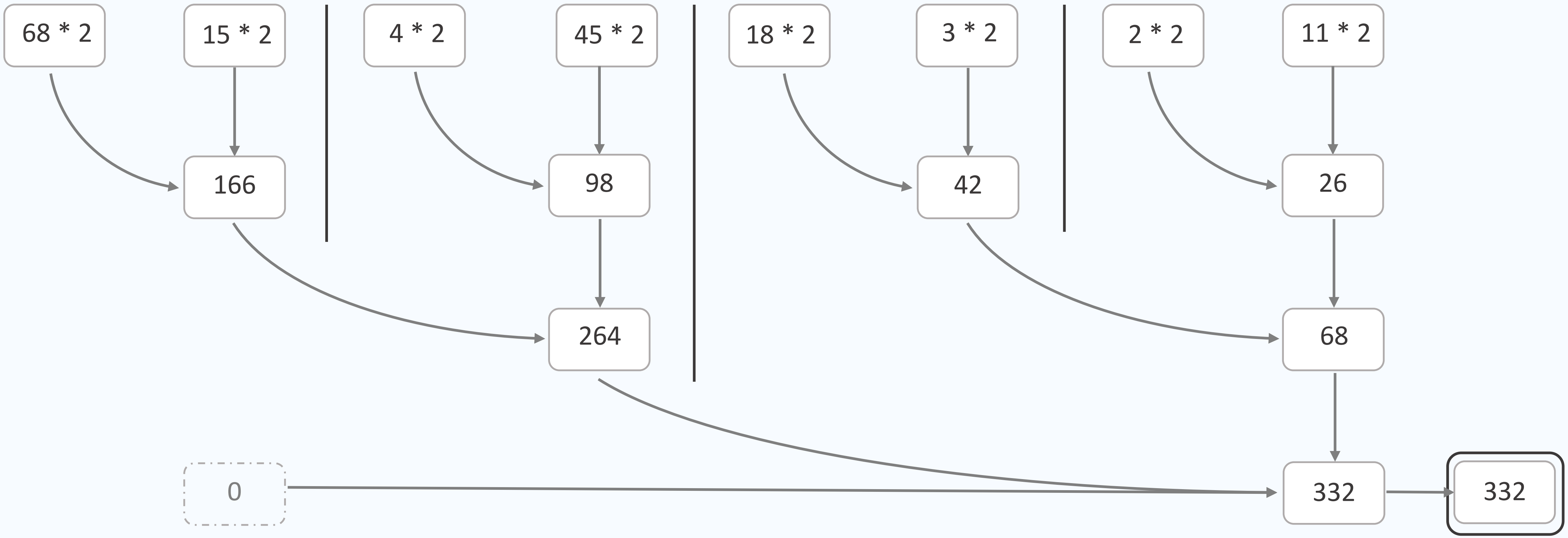
reduce walktrough



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

one honey bee hive contains 60,000 to 80,000 bees

transform_reduce walktrough

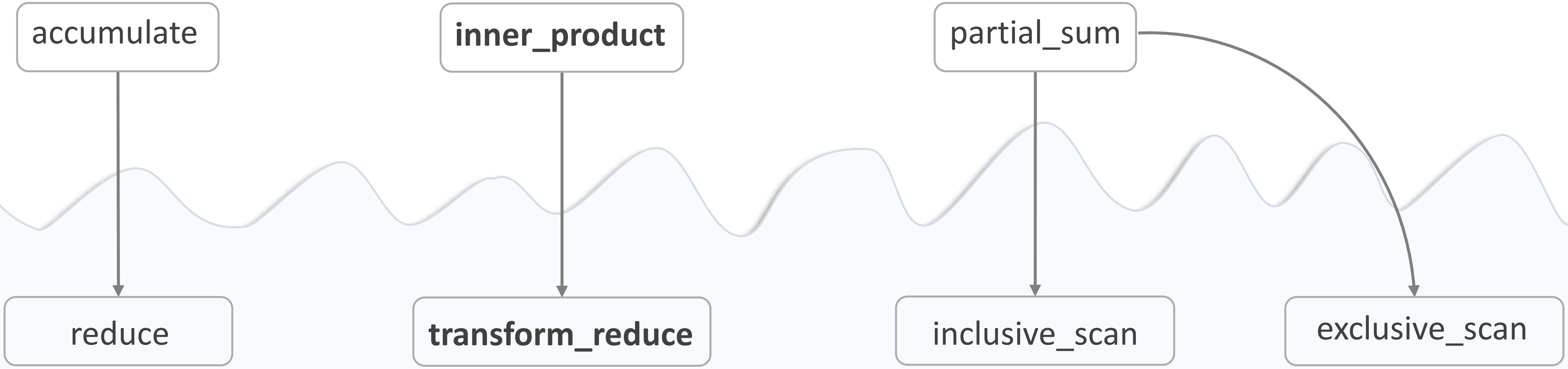


0 + (136 + 30 + (8 + 90) + (36 + 6 + (4 + 22)))   ==   0 + 2*68 + 2*15 + 2*4 + 2*45 + 2*18 + 2*3 + 2*2 + 2* 11

one honey bee hive contains 60,000 to 80,000 bees

New algorithms introduced

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



one honey bee hive contains 60,000 to 80,000 bees

Floating point addition is non-associative (Hi Guy!)

inner_product vs transform_reduce

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

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

0 + 1e-6f*68 + 2.1e-5f*15 + 3.f*4 + 43.3f*45 + 5.1e3f*18 + 6.5e4f*3 + 7.7e6f*2 + 1.5e8f*11

output: **1 665 688 704 g**

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

```
transform_reduce(execution::par, species_count.begin(), species_count.end(), weights.begin(), 0);
```

0 + (1e-6f*68 + 2.1e-5f*15 + (3.f*4 + 43.3f*45) + (5.1e3f*18 + 6.5e4f*3 + (7.7e6f*2 + 1.5e8f*11)))

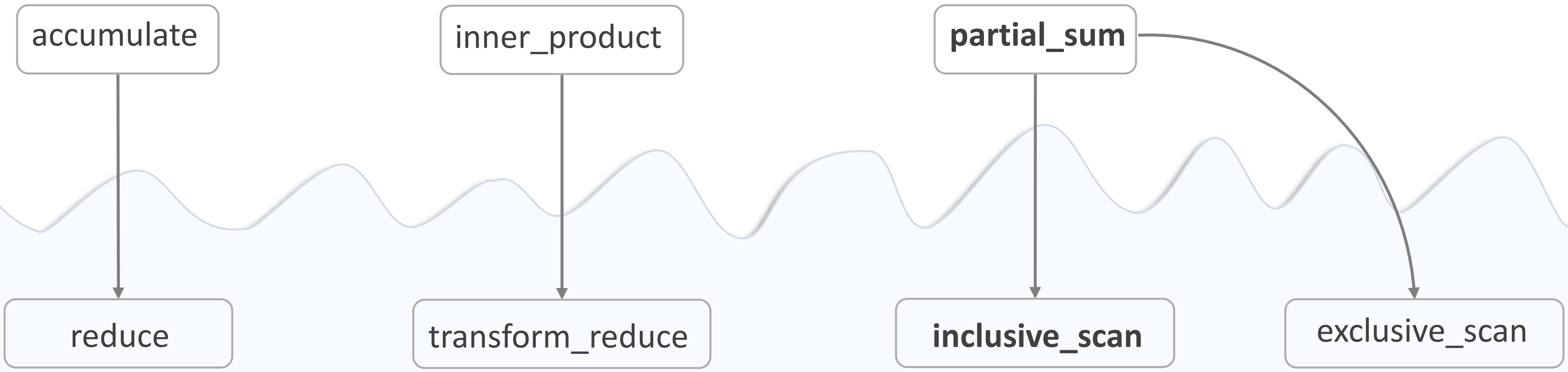
output: **1 665 688 832 g**

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

one honey bee hive contains 60,000 to 80,000 bees

New algorithms introduced

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



one honey bee hive contains 60,000 to 80,000 bees

partial_sum vs inclusive_scan

```
vector<int> species_mass{68, 30, 12, 180, 90, 18, 14, 88};
array<int, 8> edible_mass;
const int growth_factor = 2;

partial_sum(species_mass.begin(), species_mass.end(), edible_mass.begin(),
         [growth_factor](const auto& smaller_pray, const auto& equal_pray){
                  return equal_pray * growth_factor + smaller_pray;
});
```

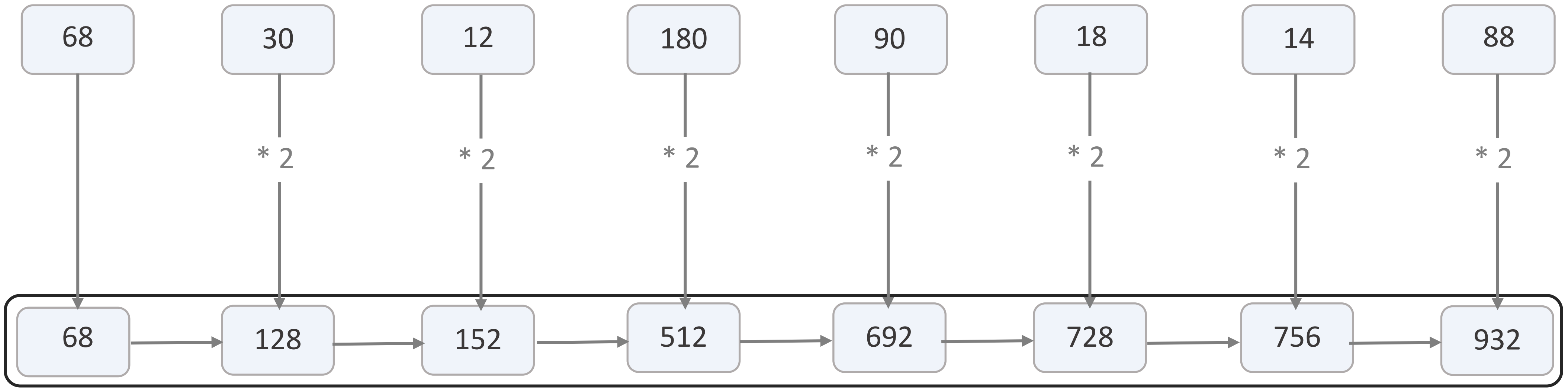
output: **68, 128, 152, 512, 692, 728, 756, 932**

```
inclusive_scan(execution::par, species_mass.begin(), species_mass.end(), edible_mass.begin(),
         [growth_factor](const auto& smaller_pray, const auto& equal_pray){
                  return equal_pray * growth_factor + smaller_pray;
});
```

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

one honey bee hive contains 60,000 to 80,000 bees

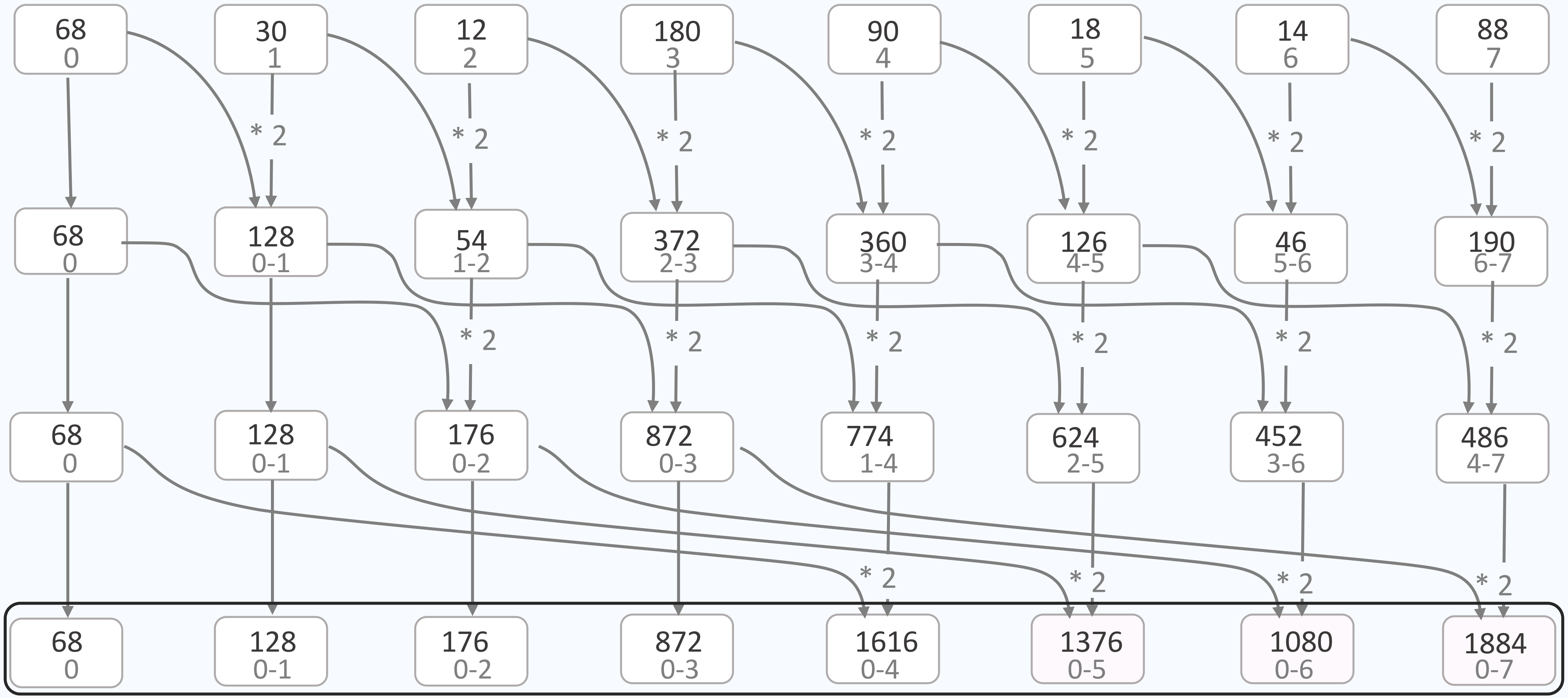
partial_sum walktrough



one honey bee hive contains 60,000 to 80,000 bees

Naïve inclusive_scan - point out synchronization points which are eliminated in real implementation (as bryce describes in Think Parallel)

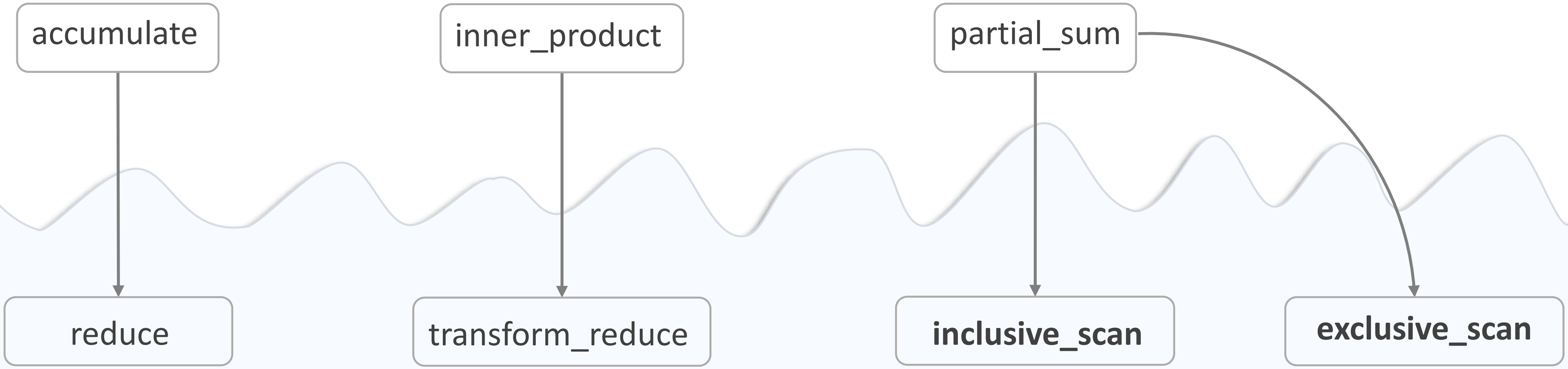
naïve inclusive_scan walktrough



one honey bee hive contains 60,000 to 80,000 bees

New algorithms introduced

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



one honey bee hive contains 60,000 to 80,000 bees

Demonstrate difference between inclusive and exclusive scan

Partial sum is inclusive!

inclusive_scan vs exclusive_scan

```
vector<int> species_mass{136, 60, 24, 360, 180, 36, 28, 176};
array<int, 8> edible_mass;
```

```
inclusive_scan(species_mass.begin(), species_mass.end(), edible_mass.begin());
```

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

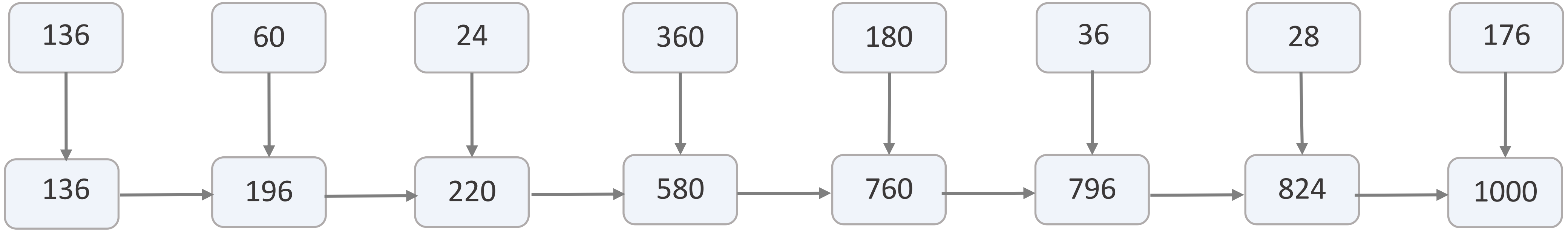
```
exclusive_scan(species_mass.begin(), species_mass.end(), edible_mass.begin(), 0);
```

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

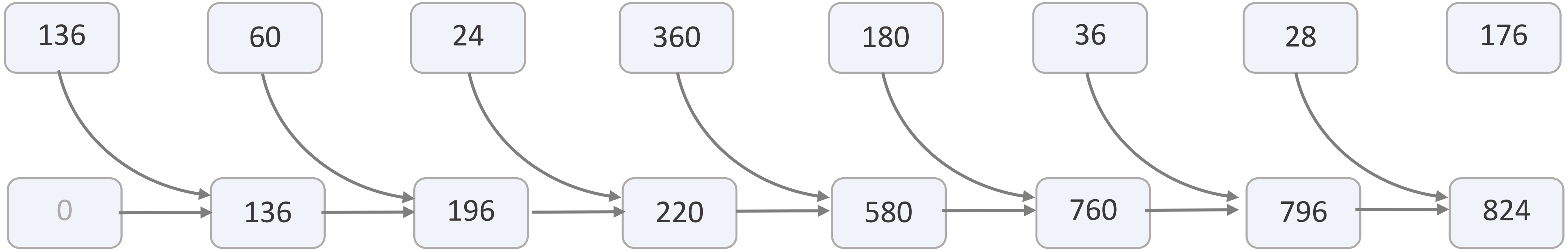
one honey bee hive contains 60,000 to 80,000 bees

inclusive_scan vs exclusive_scan

inclusive_scan:



exclusive_scan:



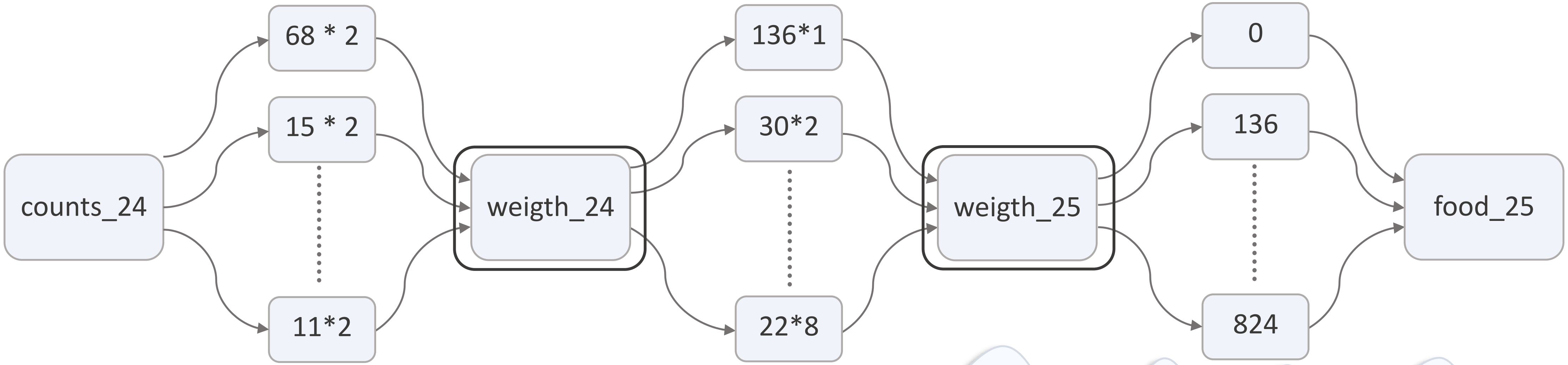
one honey bee hive contains 60,000 to 80,000 bees

Even with 3 parallel algs we end up syncing after each of it - we can pipe it using views so the flow is fluent

Unwanted synchronization points

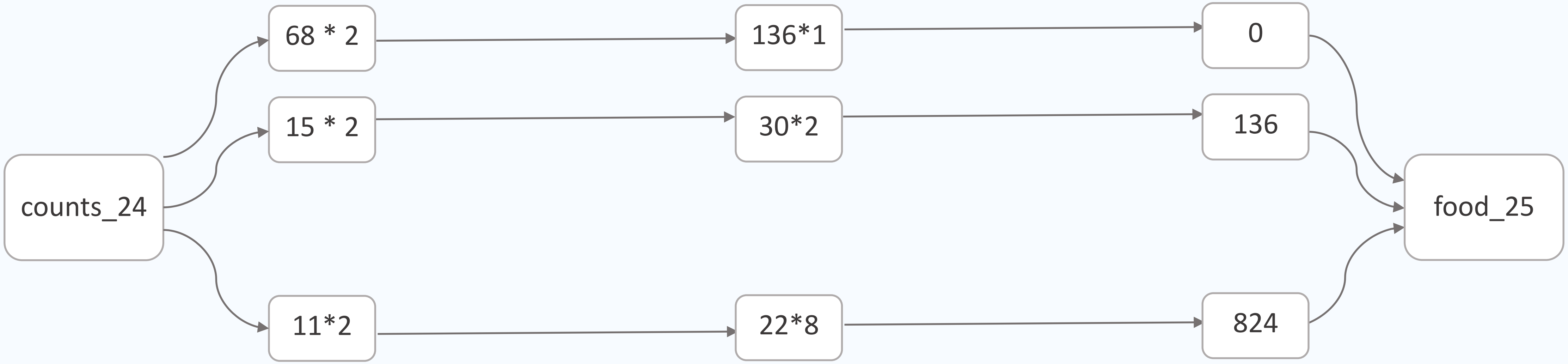
```
array<int, 8> counts_24{68, 15, 4, 45, 18, 3, 2, 11};
array<int, 8> weights{1, 2, 3, 4, 5, 6, 7, 8};

transform(par_unseq, counts_24.begin(), counts_24.end(), counts_25.begin(), [](auto i){return i * 2;});
transform(par_unseq, counts_25.begin(), counts_25.end(), weights.begin(), weight_25.begin(), multiplies{});
exclusive_scan(par_unseq, weight_25.begin(), weight_25.end(), food_25.begin(), 0);
```



one honey bee hive contains 60,000 to 80,000 bees

Unwanted synchronization points



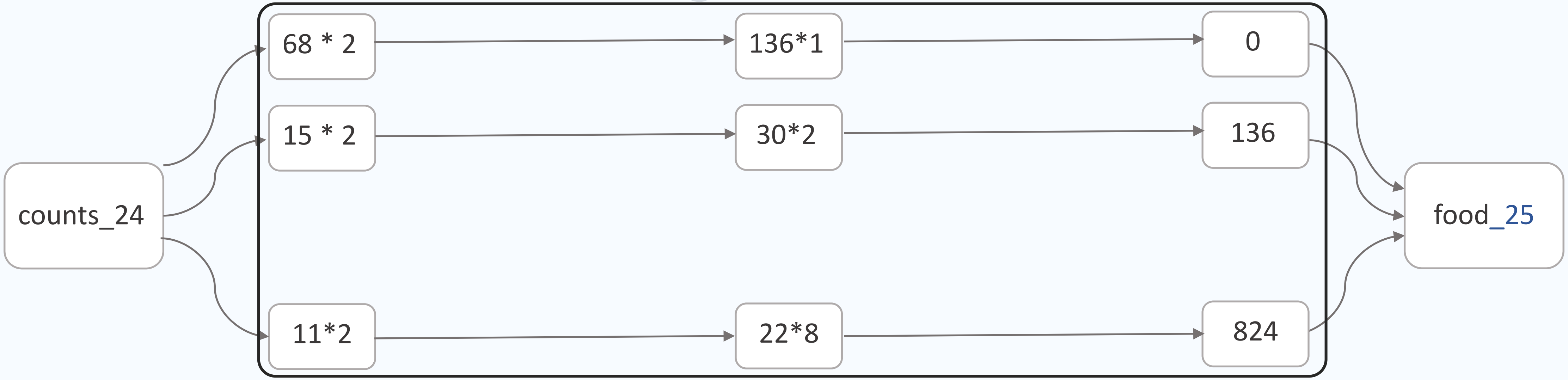

```
auto weighs_25 = views::transform([&](const auto item){return item * 2;}) |
                 views::zip_transform(multiplies{}, counts_24, weights);

exclusive_scan(par_unseq, weighs_25.begin(), weighs_25.end(), food_25.begin(), 0);
```

one honey bee hive contains 60,000 to 80,000 bees

Each of new algs has it transform_alg version which may be usefull and is cheap to implement

Unwanted synchronization points



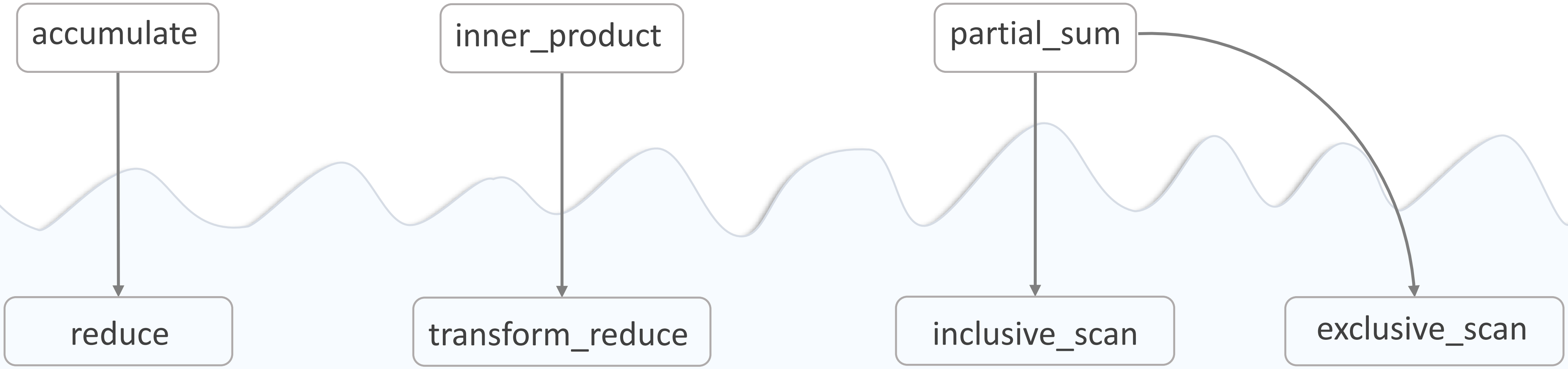

```
auto weighs_25 = views::zip_transform(multiplies{}, counts_24, weights);

transform_exclusive_scan(par_unseq, weighs_25.begin(), weighs_25.end(), food_25.begin(), 0,
                        plus{}, ([&](const auto item){return item * 2;});
```

one honey bee hive contains 60,000 to 80,000 bees

New algorithms introduced

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



one honey bee hive contains 60,000 to 80,000 bees

All new algs (does for_each counts too?)

New algorithms introduced

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

accumulate → reduce

inner_product → transform_reduce

partial_sum → inclusive_scan

partial_sum → exclusive_scan

transform_inclusive_scan

transform_exclusive_scan

one honey bee hive contains 60,000 to 80,000 bees

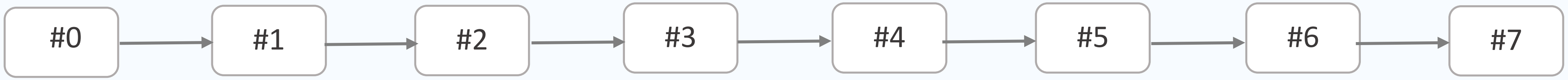
Execution policy

- 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
- The stronger policy you get, the more limited you are inside the algorithm body
- If you are unable to use strongest one, maybe you won’t gain from parallelization
- Always measure performance before and after parallelization

- execution::seq
- execution::par
- execution::unseq
- execution::par_unseq

execution::seq

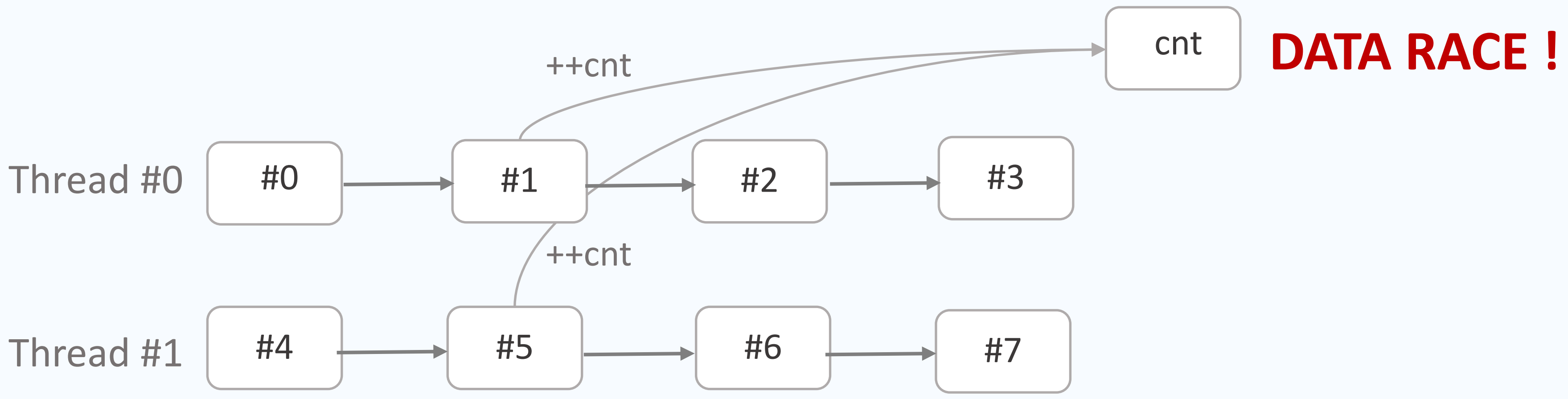
- Code is executed sequentially
- All computations are executed on single thread one by one
- No parallelization at all
- If there is no policy defined it runs in execution::seq



one honey bee hive contains 60,000 to 80,000 bees

execution::par

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

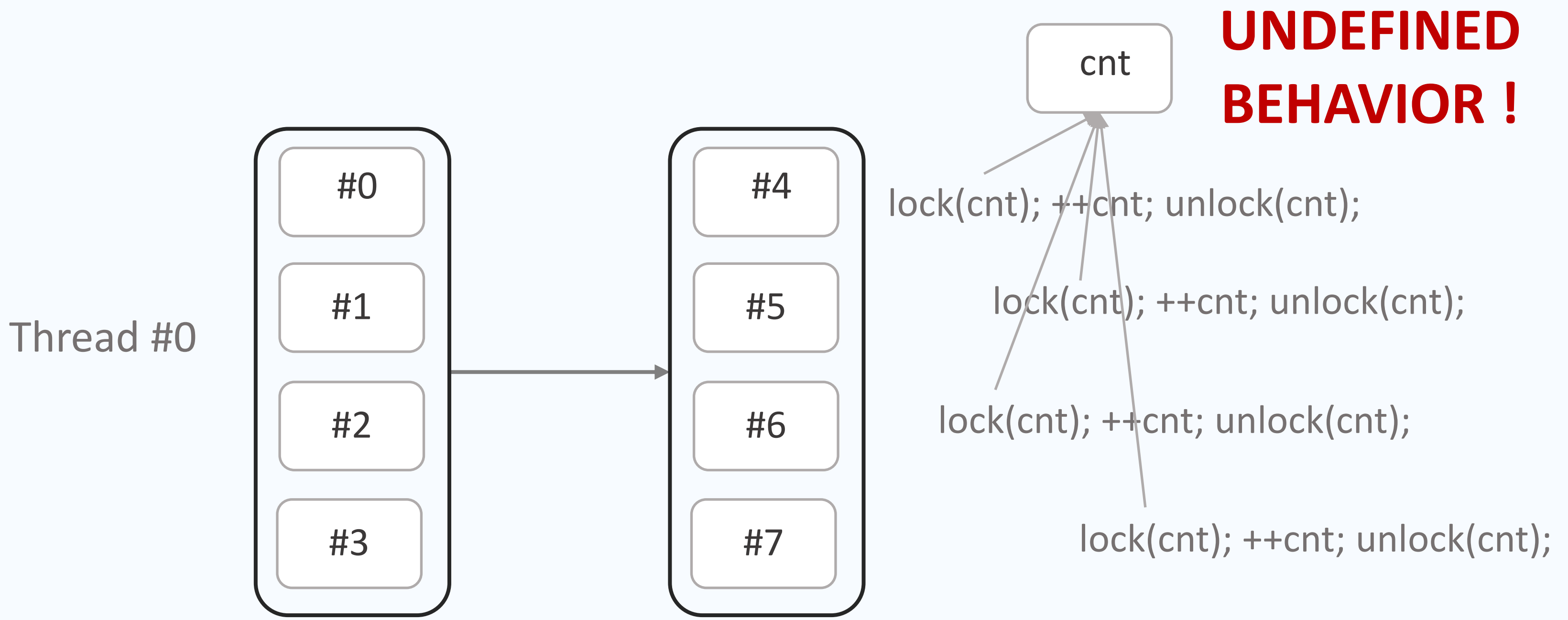


one honey bee hive contains 60,000 to 80,000 bees

Locking locked lock from single thread is UB

execution::unseq

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



one honey bee hive contains 60,000 to 80,000 bees

execution::par_unseq

- Data are vectorized and executed on multiple threads
- Strongest possible version of parallelization
- Keeps the same limitation as unseq – no synchronization is allowed
- If your code may run on unseq it may run on par_unseq there is no difference in rules
- You should try to target this policy

Easiest interactive example

What policy could we use?

Plenty of fish in the sea

```
struct Fish{ int pos; int velocity;};
vector<Fish> fishes{{32, 10}, {54, 15}, {55, 5}, {123, 20}, {124, 18}, {124, 18}, {320, 2}, {323, 5}, {480, 16}};
const auto time_interval = 1;


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

one honey bee hive contains 60,000 to 80,000 bees

Par_unseq with no problem - always go for this one first

Plenty of fish in the sea

```
struct Fish{ int pos; int velocity;};
vector<Fish> fishes{{32, 10}, {54, 15}, {55, 5}, {123, 20}, {124, 18}, {124, 18}, {320, 2}, {323, 5}, {480, 16}};
const auto time_interval = 1;


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

one honey bee hive contains 60,000 to 80,000 bees

What's wrong and how can we fix it to maintain strongest possible policy?

Beware of shark

```
struct Fish{ int pos; int velocity;};
vector<Fish> fishes{{32, 10}, {54, 15}, {55, 5}, {123, 20}, {124, 18}, {124, 18}, {320, 2}, {323, 5}, {480, 16}};
const auto time_interval = 1;
bool shark = false;             //shark is handled somewhere else and this will signal his presence

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

one honey bee hive contains 60,000 to 80,000 bees

Beware of shark

Atomic shark allows us to go par, but it's synchronization so it's not allowed to go par_unseq. We can reason that this very specific scenario would be safe, but it's still not allowed by standard.

But you don't loose much. It will be impossible to vectorize atomic operations anyway, so we would end up with par anyway

```
struct Fish{ int pos; int velocity;};
vector<Fish> fishes{{32, 10}, {54, 15}, {55, 5}, {123, 20}, {124, 18}, {124, 18}, {320, 2}, {323, 5}, {480, 16}};
const auto time_interval = 1;
atomic<bool> shark = false;      //shark is handled somewhere else and 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;
});
```

one honey bee hive contains 60,000 to 80,000 bees

Look for a hideout

Again - what's wrong and how could we fix it

Fish is looking for safety.
Each hideout has just limited capacity.
Check only those with free slots.
Choose the closest on left and right
Go for the closest
Swimming takes some time
Try to get inside and hope nobody was faster

```
struct Hideout{ int pos; int slots; };
vector<Hideout> hideouts{{30, 1}, {54, 1}, {123, 1}, {145, 1}, {345, 1}, {423, 1}, {700, 1}};

for_each(execution::par_unseq, fishes.begin(), fishes.end(), [&](auto& fish)
{
    auto safety = filter_view(hideouts, [](auto& hideout){ return hideout.slots > 0;});
    auto closest = adjacent_find(safety.begin(), safety.end(), [&](auto& first, auto& second) {
         return (first.pos < fish.pos && second.pos >= fish.pos)});

    auto& selected = choose_closer_neighbor(closest);

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

    if(selected_hideout->take_slot())
         stay_hidden();
    else
         try_mimicry();
});
```

Again with the presence of sync it can't be any unseq

We go for par by securing access to slots by mutex

Look for a hideout

```
struct Hideout{ int pos; int slots_cnt(); bool take_slot(); private: int slots; mutex slot_mutex; };
vector<Hideout> hideouts{{30, 1}, {54, 1}, {123, 1}, {145, 1}, {345, 1}, {423, 1}, {700, 1}};

for_each(execution::par, fishes.begin(), fishes.end(), [&](auto& fish)
{
    auto safety = filter_view(hideouts, [](auto& hideout){ return hideout.slots_cnt());
    auto closest = adjacent_find(safety.begin(), safety.end(), [&](auto& first, auto& second) {
         return (first.pos < fish.pos && second.pos >= fish.pos)});

    auto& selected = choose_closer_neighbor(closest);

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

    if(selected_hideout->take_slot())
         stay_hidden();
    else
         try_mimicry();
});
```

one honey bee hive contains 60,000 to 80,000 bees

Find out what would really happen if you try to sync in unseq

Become a leader

```
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;
});
```

one honey bee hive contains 60,000 to 80,000 bees

With locks you may end up by several locking instructions without any unlocking inbetween -> UB

Become a leader

```
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!

one honey bee hive contains 60,000 to 80,000 bees

Try it with atomic - elegant isn't it?

Become a leader

```
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;
});
```

one honey bee hive contains 60,000 to 80,000 bees

Become a leader

Nope

First fish will take a leadership and lives it's priviledge life.
Second fish will attemp to take the leadership before the first loose it.
Pitty they live in the same thread
Second one will wait forever as the loosing of leadership is sequenced after
Deadlock my friends

```
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();
```

DEADLOCK!

one honey bee hive contains 60,000 to 80,000 bees

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

Avoid synchronization points by utilizing views

one honey bee hive contains 60,000 to 80,000 bees

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!

one honey bee hive contains 60,000 to 80,000 bees