

Parallel Algorithms

Dietmar Kühl
Bloomberg LP

WARNING!

this talk is **not** about how to **implement** parallel algorithms but about how to **use** them

Copyright Notice

© 2017 Bloomberg L.P. Permission is granted to copy, distribute, and display this material, and to make derivative works and commercial use of it. The information in this material is provided "AS IS", without warranty of any kind. Neither Bloomberg nor any employee guarantees the correctness or completeness of such information. Bloomberg, its employees, and its affiliated entities and persons shall not be liable, directly or indirectly, in any way, for any inaccuracies, errors or omissions in such information. Nothing herein should be interpreted as stating the opinions, policies, recommendations, or positions of Bloomberg.

Lots of Concurrency

- number of cores keeps growing
- different concurrency approaches are available
 - GPU concurrency for parallelism
 - FPGAs for specialised operations

C++ is Sequential

- statements are executed in sequence
- even when operations are independent:
 - hard for compilers to detect non-trivial cases
 - order may be required accidentally
- => need to express asynchronicity potential

Example

```
for (auto it = begin; it != end; ++it, ++to) {  
    *to = fun(*it);  
}
```

- can be parallel if `fun()` doesn't have side-effects
 - using `to` and `it` doesn't introduce data races
- `size` is reasonably large or `fun()` takes long

Use OpenMP

```
#pragma omp parallel for  
for (auto it(begin); it < end; ++it, ++to) {  
    *to = fun(*it);  
}
```

- outside the language and doesn't quite fit
- it is unspecified if parallel executions nest
- only works with random access iterators

Use `std::thread`

```
std::vector<std::thread> ts;
for (auto it(begin), e(begin); it != end; it = e) {
    e += std::min(std::distance(it, end), batchsize);
    ts.emplace_back([=]() {
        for (; it != e; ++it, ++to) {
            *to = fun(*it);
        }
    });
}
for (auto& t: ts) { t.join(); }
```

- not easy to use and not necessarily efficient

Use `std::async`

```
std::vector<std::future<void>> fs;
for (auto it(begin), e(begin); it != end; it = e) {
    e += std::min(std::distance(it, end), batchsize);
    fs.emplace_back(std::async([=]() {
        for (; it != e; ++it, ++to) {
            *to = fun(*it);
        }
    }));
}
for (auto& f: fs) { f.get(); }
```

- not easy to use and not necessarily efficient

Use TBB

```
using range = tbb::blocked_range<int>;  
tbb::parallel_for(range(0, end - begin),  
    [=](range const& r){  
        for (auto i(r.begin()); i != r.end(); ++i) {  
            to[i] = fun(begin[i]);  
        });  
});
```

- some algorithms are easier to use
- a reasonable direction

Parallel Algorithm

```
std::transform(std::execution::par, b, e, to, fun);
```

- can use different policies (seq, par, ...)
- assumptions about the parameters are made:
 - parameter calls don't introduce data races
 - parameters can be copied (not just moved)

Status Quo

- algorithms execute sequentially

```
std::for_each(begin, end, fun);
```

```
std::transform(begin, end, to, fun);
```

```
std::reduce(begin, end, fun);
```

```
std::inclusive_scan(begin, end, to, op);
```

Objective

- enable easy parallel execution

```
std::for_each(policy, begin, end, fun);
```

```
std::transform(policy, begin, end, to, fun);
```

```
std::reduce(policy, begin, end, fun);
```

```
std::inclusive_scan(policy, begin, end, to, op);
```

Possibly Not That Easy

1. nobody uses algorithms
2. potential of improvements depends on use
 - no point parallelising fast executing small loops
3. parallel execution may introduce data races
 - through iterators or function objects

Concurrency Model

- pass *execution policy* to **allow** concurrency
 - type indicates **permitted** approaches
- *element access functions* **obey** policy-specific **constraints**
- implementation **may** take advantage of these
 - ... but is **not** required to do so

Element Access Functions

functions used on parameters:

- any iterator operation according to its **category**
- operations **specified** to be used on elements
- **specified** uses of function objects
- **required** operations on function objects

Execution Policy

- `std::is_execution_policy<T>::value` for detection
- `std::execution::sequenced_policy`
`std::execution::seq`
- `std::execution::parallel_policy`
`std::execution::par`
- `std::parallel_unsequenced_policy`
`std::execution::par_unseq`

std::execution::seq

- sequential execution
- for debugging, choosing a policy in generic code
- same common constraints and interface changes
 - exceptions result in std::terminate()
 - no [required] support for input iterators
 - changed return types for some algorithms

std::execution::par

- allow parallel [threaded] execution
- element access functions shall not introduce data races
 - they *can* use locks (when really necessary)
- no interleaved execution

std::execution::par_unseq

- allow parallel, interleaved execution
 - for example using multiple threads on a GPU
- element access functions shall not introduce data races and have no order dependency
 - they *cannot* use locks

Supported Algorithms

roughly: all algorithms for which concurrent execution may be a benefit are supported

- no support for sub-linear algorithms
- some algorithms use different names
- some algorithms are rarely used and complicated to parallelise
- some oddballs are not supported

Algorithms

accumulate
adjacent_difference
adjacent_find
all_of
any_of
binary_search
clamp
copy
copy_backward
copy_if
copy_n
count
count_if
destroy
destroy_at
destroy_n
equal
equal_range
exclusive_scan
fill
fill_n
find
find_end
find_first_of
find_if
find_if_not
for_each
for_each_n
gcd
generate

generate_n
includes
inclusive_scan
inner_product
inplace_merge
iota
is_heap
is_heap_until
is_partitioned
is_permutation
is_sorted
is_sorted_until
iter_swap
lcm
lexicographical_compare
lower_bound
make_heap
max
max_element
merge
min
min_element
minmax
minmax_element
mismatch
move
move_backward
next_permutation
none_of
nth_element

partial_sort
partial_sort_copy
partial_sum
partition
partition_copy
partition_point
pop_heap
prev_permutation
push_heap
reduce
remove
remove_copy
remove_copy_if
remove_if
replace
replace_copy
replace_copy_if
replace_if
reverse
reverse_copy
rotate
rotate_copy
sample
search
search_n
set_difference
set_intersection
set_symmetric_difference
set_union
shuffle

sort
sort_heap
stable_partition
stable_sort
swap_ranges
transform
transform_exclusive_scan
transform_inclusive_scan
transform_reduce
uninitialized_copy
uninitialized_copy_n
uninitialized_default_construct
uninitialized_default_construct_n
uninitialized_fill
uninitialized_fill_n
uninitialized_move
uninitialized_move_n
uninitialized_value_construct
uninitialized_value_construct_n
unique
unique_copy
upper_bound

Algorithms: $O(1)$

accumulate
adjacent_difference
adjacent_find
all_of
any_of
binary_search

clamp

copy
copy_backward

copy_if

copy_n

count

count_if

destroy

destroy_at

destroy_n

equal

equal_range

exclusive_scan

fill

fill_n

find

find_end

find_first_of

find_if

find_if_not

for_each

for_each_n

gcd

generate

generate_n

includes

inclusive_scan

inner_product

inplace_merge

iota

is_heap

is_heap_until

is_partitioned

is_permutation

is_sorted

is_sorted_until

iter_swap

lcm

lexicographical_compare

lower_bound

make_heap

max

max_element

merge

min

min_element

minmax

minmax_element

mismatch

move

move_backward

next_permutation

none_of

nth_element

partial_sort

partial_sort_copy

partial_sum

partition

partition_copy

partition_point

pop_heap

prev_permutation

push_heap

reduce

remove

remove_copy

remove_copy_if

remove_if

replace

replace_copy

replace_copy_if

replace_if

reverse

reverse_copy

rotate

rotate_copy

sample

search

search_n

set_difference

set_intersection

set_symmetric_difference

set_union

shuffle

sort

sort_heap

stable_partition

stable_sort

swap_ranges

transform

transform_exclusive_scan

transform_inclusive_scan

transform_reduce

uninitialized_copy

uninitialized_copy_n

uninitialized_default_construct

uninitialized_default_construct_n

uninitialized_fill

uninitialized_fill_n

uninitialized_move

uninitialized_move_n

uninitialized_value_construct

uninitialized_value_construct_n

unique

unique_copy

upper_bound

Algorithms: $O(\ln n)$

accumulate
adjacent_difference
adjacent_find
all_of
any_of

binary_search

copy
copy_backward
copy_if
copy_n
count
count_if
destroy

destroy_n
equal
equal_range

exclusive_scan
fill
fill_n
find
find_end
find_first_of
find_if
find_if_not
for_each
for_each_n

generate

generate_n
includes
inclusive_scan
inner_product
inplace_merge
iota
is_heap
is_heap_until
is_partitioned
is_permutation
is_sorted
is_sorted_until

lexicographical_compare
lower_bound

make_heap

max_element
merge

min_element

minmax_element
mismatch
move
move_backward
next_permutation
none_of
nth_element

partial_sort
partial_sort_copy
partial_sum
partition
partition_copy

partition_point
pop_heap
prev_permutation
push_heap

reduce
remove
remove_copy
remove_copy_if
remove_if
replace
replace_copy
replace_copy_if
replace_if
reverse
reverse_copy
rotate
rotate_copy
sample
search
search_n
set_difference
set_intersection
set_symmetric_difference
set_union
shuffle

sort
sort_heap
stable_partition
stable_sort
swap_ranges
transform
transform_exclusive_scan
transform_inclusive_scan
transform_reduce
uninitialized_copy
uninitialized_copy_n
uninitialized_default_construct
uninitialized_default_construct_n
uninitialized_fill
uninitialized_fill_n
uninitialized_move
uninitialized_move_n
uninitialized_value_construct
uninitialized_value_construct_n
unique
unique_copy
upper_bound

Algorithms: heap

accumulate
adjacent_difference
adjacent_find
all_of
any_of

copy
copy_backward
copy_if
copy_n
count
count_if
destroy

destroy_n
equal

exclusive_scan
fill
fill_n
find
find_end
find_first_of
find_if
find_if_not
for_each
for_each_n

generate

generate_n
includes
inclusive_scan
inner_product
inplace_merge
iota
is_heap
is_heap_until
is_partitioned
is_permutation
is_sorted
is_sorted_until

lexicographical_compare
make_heap

max_element
merge

min_element

minmax_element
mismatch
move
move_backward
next_permutation
none_of
nth_element

partial_sort
partial_sort_copy
partial_sum
partition
partition_copy

prev_permutation

reduce
remove
remove_copy
remove_copy_if
remove_if
replace
replace_copy
replace_copy_if
replace_if
reverse
reverse_copy
rotate
rotate_copy
sample
search
search_n
set_difference
set_intersection
set_symmetric_difference
set_union
shuffle

sort
sort_heap

stable_partition
stable_sort
swap_ranges
transform
transform_exclusive_scan
transform_inclusive_scan
transform_reduce
uninitialized_copy
uninitialized_copy_n
uninitialized_default_construct
uninitialized_default_construct_n
uninitialized_fill
uninitialized_fill_n
uninitialized_move
uninitialized_move_n
uninitialized_value_construct
uninitialized_value_construct_n
unique
unique_copy

Algorithms: permutation

accumulate
adjacent_difference
adjacent_find
all_of
any_of

copy
copy_backward
copy_if
copy_n
count
count_if
destroy

destroy_n
equal

exclusive_scan
fill
fill_n
find
find_end
find_first_of
find_if
find_if_not
for_each
for_each_n

generate

generate_n
includes
inclusive_scan
inner_product
inplace_merge

iota
is_heap
is_heap_until
is_partitioned
is_permutation
is_sorted
is_sorted_until

lexicographical_compare

max_element
merge

min_element

minmax_element
mismatch
move

move_backward

none_of
nth_element

partial_sort
partial_sort_copy
partial_sum
partition
partition_copy

prev_permutation

reduce
remove
remove_copy
remove_copy_if
remove_if
replace
replace_copy
replace_copy_if
replace_if

reverse
reverse_copy
rotate
rotate_copy

sample

search
search_n
set_difference
set_intersection
set_symmetric_difference
set_union
shuffle

sort

stable_partition
stable_sort
swap_ranges
transform
transform_exclusive_scan
transform_inclusive_scan
transform_reduce
uninitialized_copy
uninitialized_copy_n
uninitialized_default_construct
uninitialized_default_construct_n
uninitialized_fill
uninitialized_fill_n
uninitialized_move
uninitialized_move_n
uninitialized_value_construct
uninitialized_value_construct_n
unique
unique_copy

Algorithms: overlapping

accumulate
adjacent_difference
adjacent_find
all_of
any_of

copy
copy_backward

copy_if
copy_n
count
count_if
destroy

destroy_n
equal

exclusive_scan
fill
fill_n
find
find_end
find_first_of
find_if
find_if_not
for_each
for_each_n

generate

generate_n
includes
inclusive_scan
inner_product
inplace_merge
iota
is_heap
is_heap_until
is_partitioned

is_sorted
is_sorted_until

lexicographical_compare

max_element
merge

min_element

minmax_element
mismatch
move

move_backward

none_of
nth_element

partial_sort
partial_sort_copy
partial_sum
partition
partition_copy

reduce
remove
remove_copy
remove_copy_if
remove_if
replace
replace_copy
replace_copy_if
replace_if

reverse
reverse_copy
rotate
rotate_copy

sample

search
search_n
set_difference
set_intersection
set_symmetric_difference
set_union
shuffle

sort

stable_partition
stable_sort
swap_ranges
transform
transform_exclusive_scan
transform_inclusive_scan
transform_reduce
uninitialized_copy
uninitialized_copy_n
uninitialized_default_construct
uninitialized_default_construct_n
uninitialized_fill
uninitialized_fill_n
uninitialized_move
uninitialized_move_n
uninitialized_value_construct
uninitialized_value_construct_n
unique
unique_copy

Algorithms: renamed

accumulate

adjacent_difference
adjacent_find
all_of
any_of

copy

copy_if
copy_n
count
count_if
destroy

destroy_n
equal

exclusive_scan
fill
fill_n
find
find_end
find_first_of
find_if
find_if_not
for_each
for_each_n

generate

generate_n
includes
inclusive_scan
inner_product
inplace_merge
iota
is_heap
is_heap_until
is_partitioned

is_sorted
is_sorted_until

lexicographical_compare

max_element
merge

min_element

minmax_element
mismatch
move

none_of
nth_element

partial_sort
partial_sort_copy
partition
partition_copy

partial_sum

reduce
remove
remove_copy
remove_copy_if
remove_if
replace
replace_copy
replace_copy_if
replace_if
reverse
reverse_copy
rotate
rotate_copy
sample
search
search_n
set_difference
set_intersection
set_symmetric_difference
set_union
shuffle

sort

stable_partition
stable_sort
swap_ranges
transform
transform_exclusive_scan
transform_inclusive_scan
transform_reduce
uninitialized_copy
uninitialized_copy_n
uninitialized_default_construct
uninitialized_default_construct_n
uninitialized_fill
uninitialized_fill_n
uninitialized_move
uninitialized_move_n
uninitialized_value_construct
uninitialized_value_construct_n
unique
unique_copy

Algorithms: oddballs

adjacent_difference
adjacent_find
all_of
any_of

copy

copy_if
copy_n
count
count_if
destroy

destroy_n
equal

exclusive_scan
fill
fill_n
find
find_end
find_first_of
find_if
find_if_not
for_each
for_each_n

generate

generate_n
includes
inclusive_scan
inner_product
inplace_merge
iota
is_heap
is_heap_until
is_partitioned

is_sorted
is_sorted_until

lexicographical_compare

max_element
merge

min_element

minmax_element
mismatch
move

none_of
nth_element

partial_sort
partial_sort_copy

partition
partition_copy

reduce
remove
remove_copy
remove_copy_if
remove_if
replace
replace_copy
replace_copy_if
replace_if

reverse
reverse_copy
rotate
rotate_copy

sample

search
search_n
set_difference
set_intersection
set_symmetric_difference
set_union

shuffle

sort

stable_partition
stable_sort
swap_ranges
transform
transform_exclusive_scan
transform_inclusive_scan
transform_reduce
uninitialized_copy
uninitialized_copy_n
uninitialized_default_construct
uninitialized_default_construct_n
uninitialized_fill
uninitialized_fill_n
uninitialized_move
uninitialized_move_n
uninitialized_value_construct
uninitialized_value_construct_n
unique
unique_copy

Supported Algorithms

adjacent_difference	generate_n	partial_sort	
adjacent_find	includes	partial_sort_copy	
all_of	inclusive_scan		
any_of	inner_product	partition	sort
	inplace_merge	partition_copy	
copy	is_heap		stable_partition
	is_heap_until		stable_sort
	is_partitioned		swap_ranges
copy_if			transform
copy_n	is_sorted	reduce	transform_exclusive_scan
count	is_sorted_until	remove	transform_inclusive_scan
count_if		remove_copy	transform_reduce
destroy		remove_copy_if	uninitialized_copy
	lexicographical_compare	remove_if	uninitialized_copy_n
destroy_n		replace	uninitialized_default_construct
equal		replace_copy	uninitialized_default_construct_n
		replace_copy_if	uninitialized_fill
exclusive_scan	max_element	replace_if	uninitialized_fill_n
fill	merge	reverse	uninitialized_move
fill_n		reverse_copy	uninitialized_move_n
find	min_element	rotate	uninitialized_value_construct
find_end		rotate_copy	uninitialized_value_construct_n
find_first_of	minmax_element		unique
find_if	mismatch	search	unique_copy
find_if_not	move	search_n	
for_each		set_difference	
for_each_n		set_intersection	
		set_symmetric_difference	
generate	none_of	set_union	
	nth_element		

Algorithms: map

adjacent_difference
adjacent_find
all_of
any_of

copy

copy_if
copy_n

count

count_if

destroy

destroy_n

equal

exclusive_scan

fill

fill_n

find

find_end

find_first_of

find_if

find_if_not

for_each

for_each_n

generate

generate_n

includes

inclusive_scan

inner_product

inplace_merge

is_heap

is_heap_until

is_partitioned

is_sorted

is_sorted_until

lexicographical_compare

max_element

merge

min_element

minmax_element

mismatch

move

none_of

nth_element

partial_sort

partial_sort_copy

partition

partition_copy

reduce

remove

remove_copy

remove_copy_if

remove_if

replace

replace_copy

replace_copy_if

replace_if

reverse

reverse_copy

rotate

rotate_copy

search

search_n

set_difference

set_intersection

set_symmetric_difference

set_union

sort

stable_partition

stable_sort

swap_ranges

transform

transform_exclusive_scan

transform_inclusive_scan

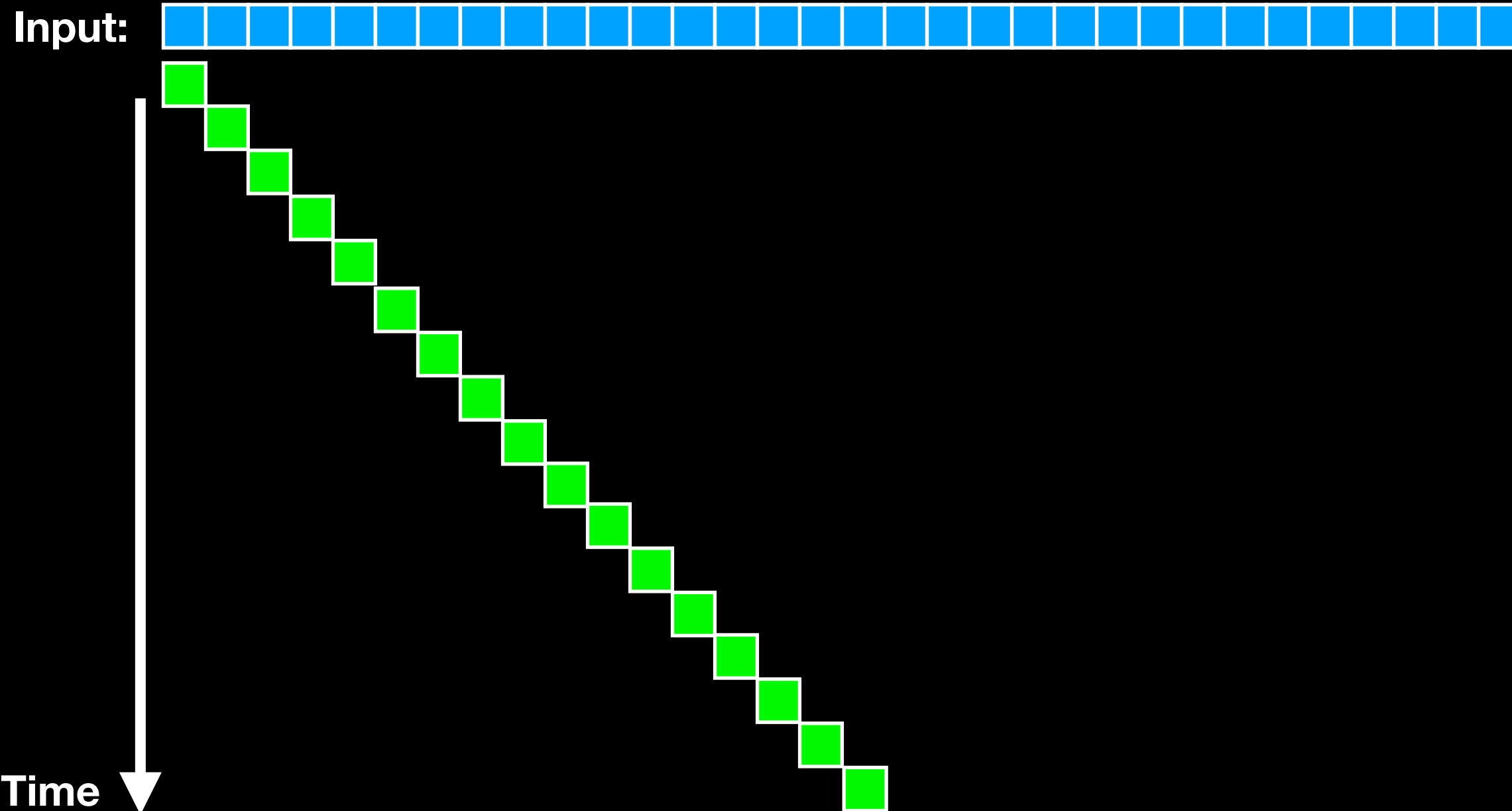
transform_reduce

uninitialized_*

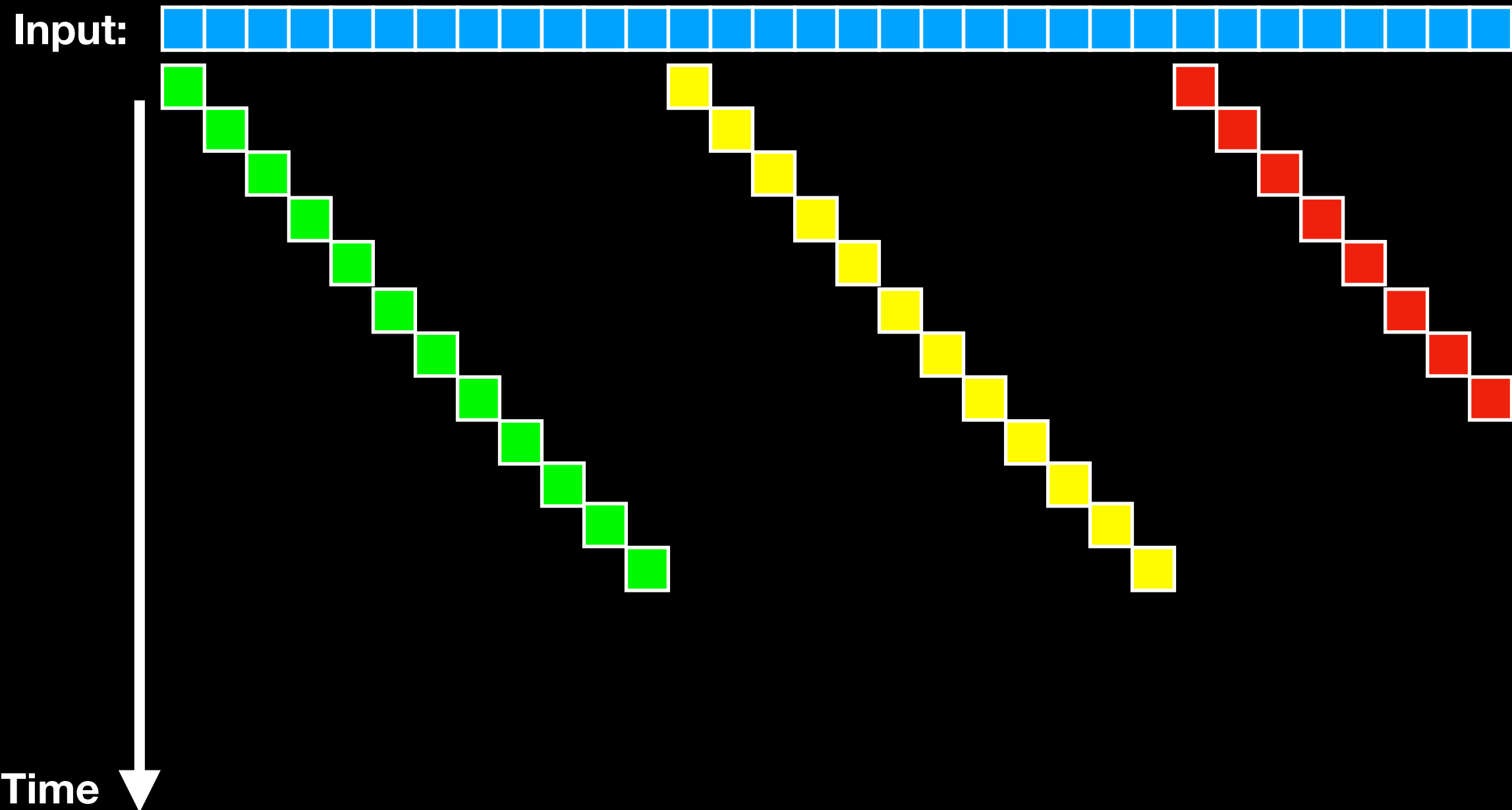
unique

unique_copy

Map: seq

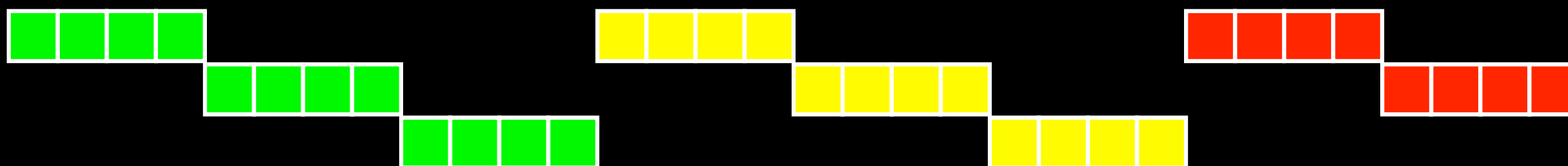


Map: par



Map: par_unseq

Input:



Time



Additional Constraints

- `for_each()`, `for_each_n()` don't return the function
- `copy()`, `move()`: source and range can't overlap
- `copy_n()` can overlap: probably a defect
- for non-random access: may require reduce

Algorithms: reduce

adjacent_difference
all_of
any_of

copy_if
count
count_if

equal

exclusive_scan
find
find_end
find_first_of
find_if
find_if_not

includes
inclusive_scan
inner_product
inplace_merge
is_heap
is_heap_until
is_partitioned
is_sorted
is_sorted_until

lexicographical_compare

max_element

merge

min_element

minmax_element

mismatch

none_of

nth_element

partial_sort
partial_sort_copy

partition
partition_copy

reduce

remove
remove_copy
remove_copy_if
remove_if

rotate
rotate_copy

search

search_n

set_difference
set_intersection
set_symmetric_difference
set_union

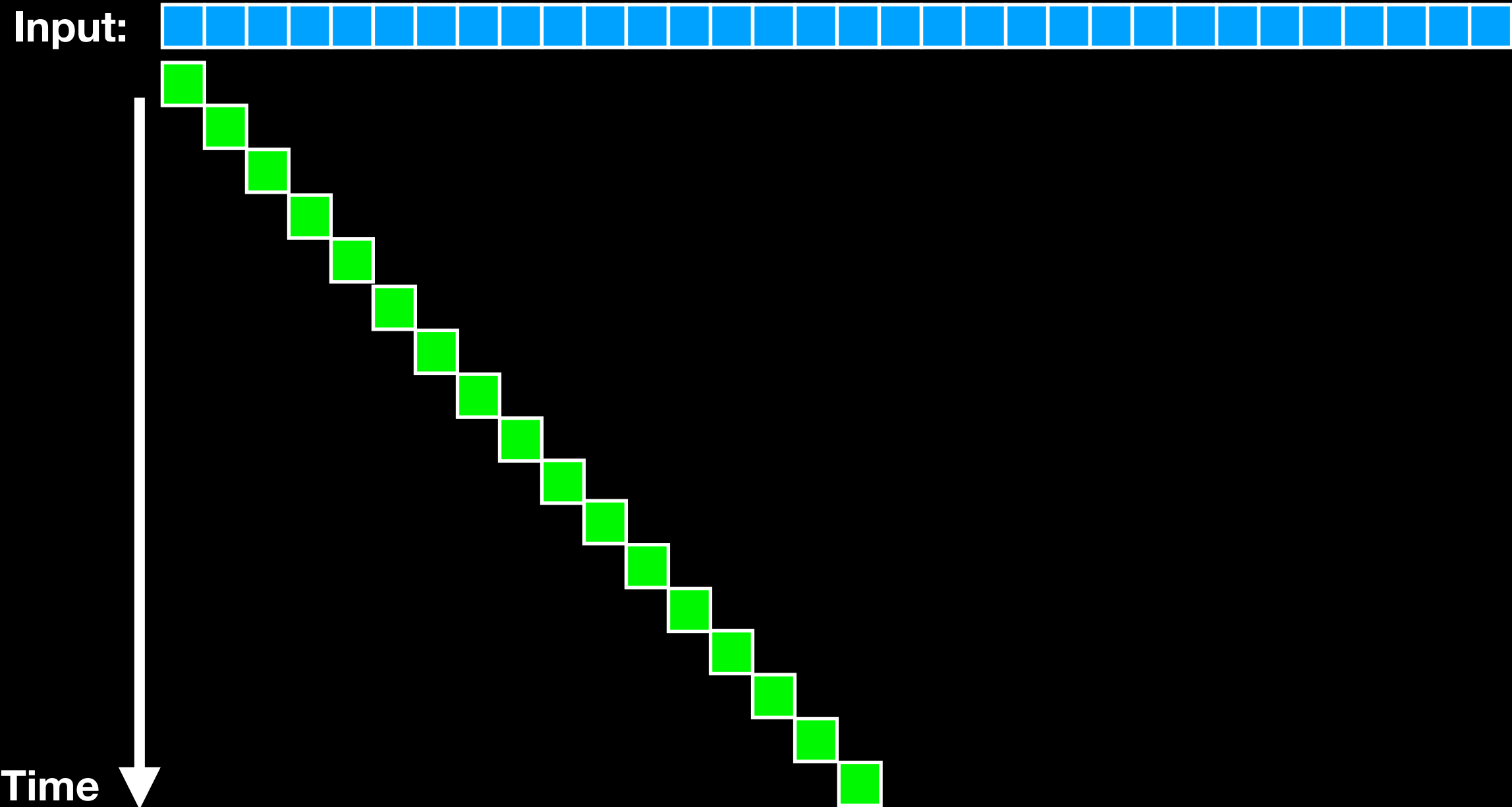
sort

stable_partition
stable_sort

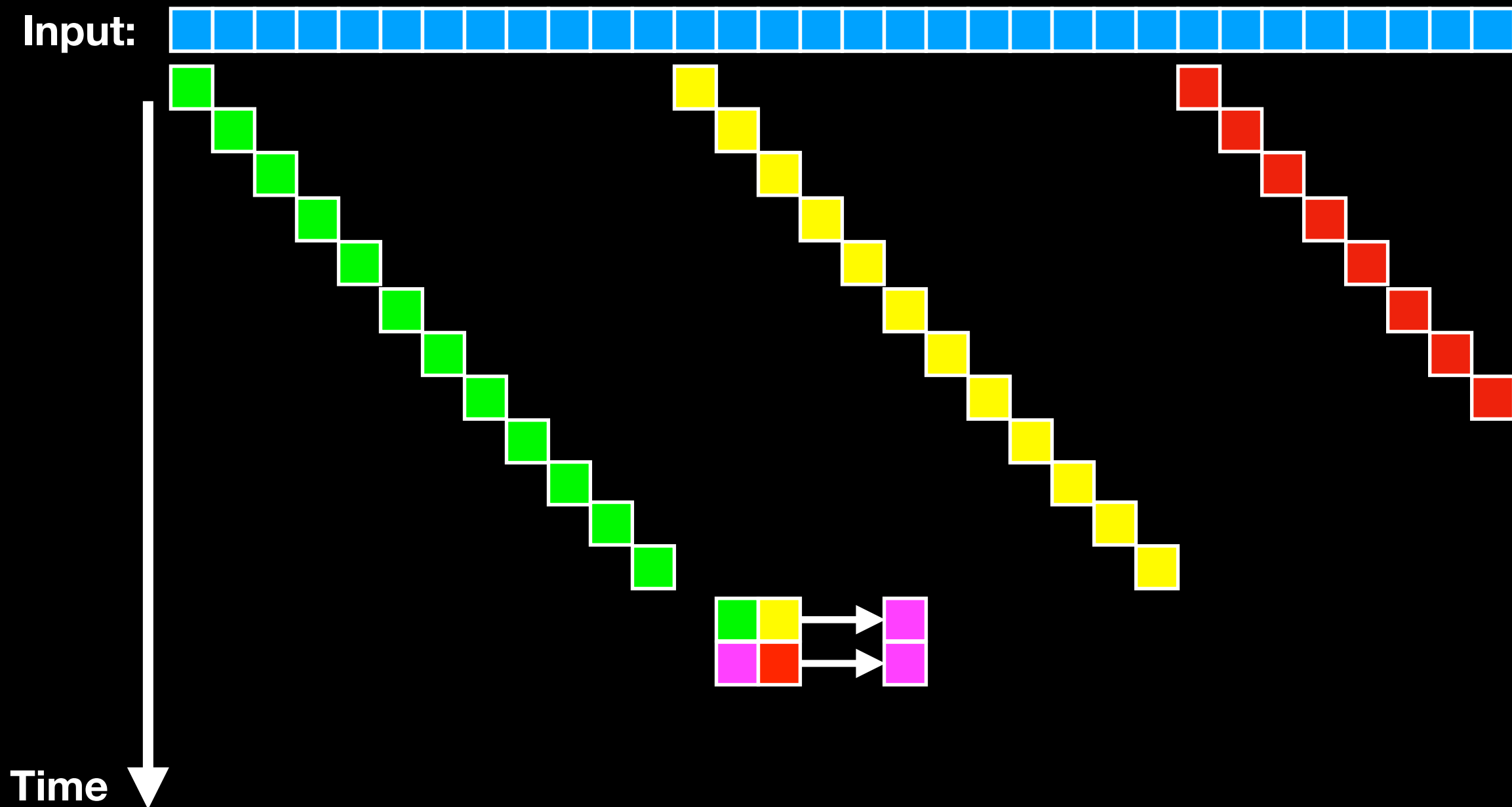
transform_exclusive_scan
transform_inclusive_scan
transform_reduce

unique
unique_copy

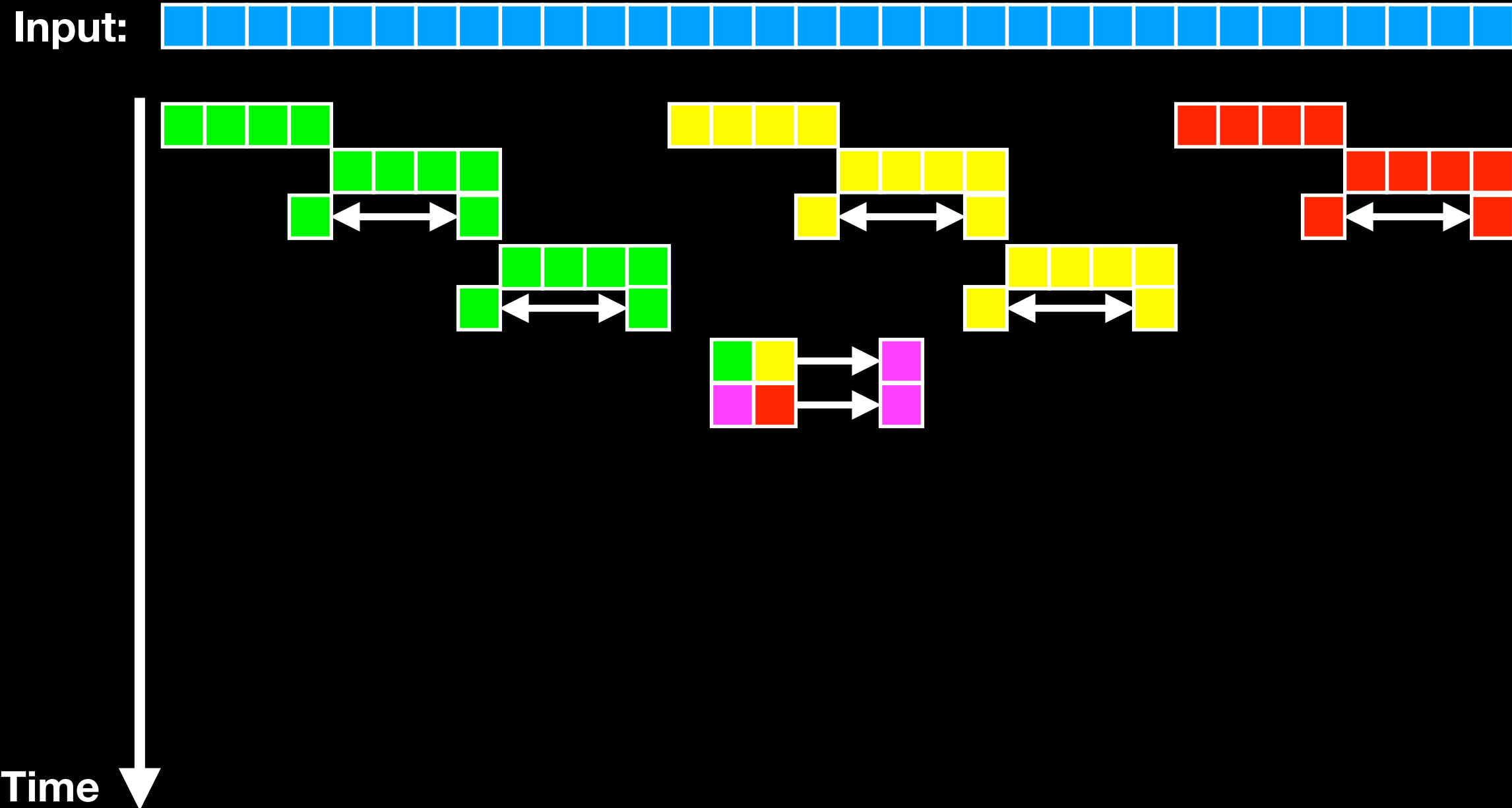
Reduce: seq



Reduce: par



Reduce: par_unseq



Algorithms: reduce

- `accumulate()` becomes `reduce()`
- operations need to be **associative**
 - otherwise wrong results are produced
- find algorithms may omit using the entire range

Algorithms: scan

adjacent_difference

inclusive_scan

inplace_merge

partial_sort
partial_sort_copy

partition
partition_copy

sort

stable_partition
stable_sort

copy_if

remove
remove_copy
remove_copy_if
remove_if

transform_exclusive_scan
transform_inclusive_scan
transform_reduce

exclusive_scan

merge

rotate
rotate_copy

unique
unique_copy

set_difference
set_intersection
set_symmetric_difference
set_union

nth_element

Algorithms: scan

- `partial_sum()` becomes `inclusive_scan()`
- may produce different results when operation isn't associative
- `inclusive_scan()`: $r[i]$ uses $s[0], \dots, s[i]$
- `exclusive_scan()`: $r[i]$ uses $s[0], \dots, s[i - 1]$
- note: order of initial value and operation differ between `inclusive_scan()` and `exclusive_scan()`!

Algorithms: fused

adjacent_difference

copy_if

inplace_merge

merge

nth_element

partial_sort
partial_sort_copy

partition
partition_copy

remove
remove_copy
remove_copy_if
remove_if

rotate
rotate_copy

set_difference
set_intersection
set_symmetric_difference
set_union

sort

stable_partition
stable_sort

transform_exclusive_scan
transform_inclusive_scan
transform_reduce

unique
unique_copy

Algorithms: gather

adjacent_difference

inplace_merge

copy_if

partial_sort
partial_sort_copy

partition
partition_copy

sort
stable_partition
stable_sort

remove
remove_copy
remove_copy_if
remove_if

merge

rotate
rotate_copy

unique
unique_copy

set_difference
set_intersection
set_symmetric_difference
set_union

nth_element

Algorithms: special

adjacent_difference

partial_sort
partial_sort_copy

inplace_merge

partition

sort

stable_partition
stable_sort

merge

rotate

set_difference
set_intersection
set_symmetric_difference
set_union

nth_element

Availability

- part of C++17 standard library
- according to P0024R1 multiple implementations
 - of the parallel algorithms proposal N3554
 - all implementations seem to be partial
- **not**, yet, shipping with compilers

Current Implementations

- often only a subset of algorithms is implemented
- typically no support for non-random access
- no support for `std::execution::par_unseq`
- implementations don't implement a fallback
- HPX seems complete for x86 Linux

HPX Usage

- in namespace `hpx::parallel` and needs setup:

```
int hpx_main(int ac, char* av[]) {  
    ...  
    return hpx::finalize();  
}  
int main(int ac, char* av[]) {  
    std::vector<std::string> cfg{"hpx.os_threads=all"};  
    hpx::init(ac, av, cfg);  
}
```


Results: Machines

- Intel Xeon Phi: 64 cores, 4 hyper threaded, 96GB
- Intel I7: 4 cores, 2x hyper threaded, 32GB
- ARM: 4 cores, not hyper threaded, 1GB

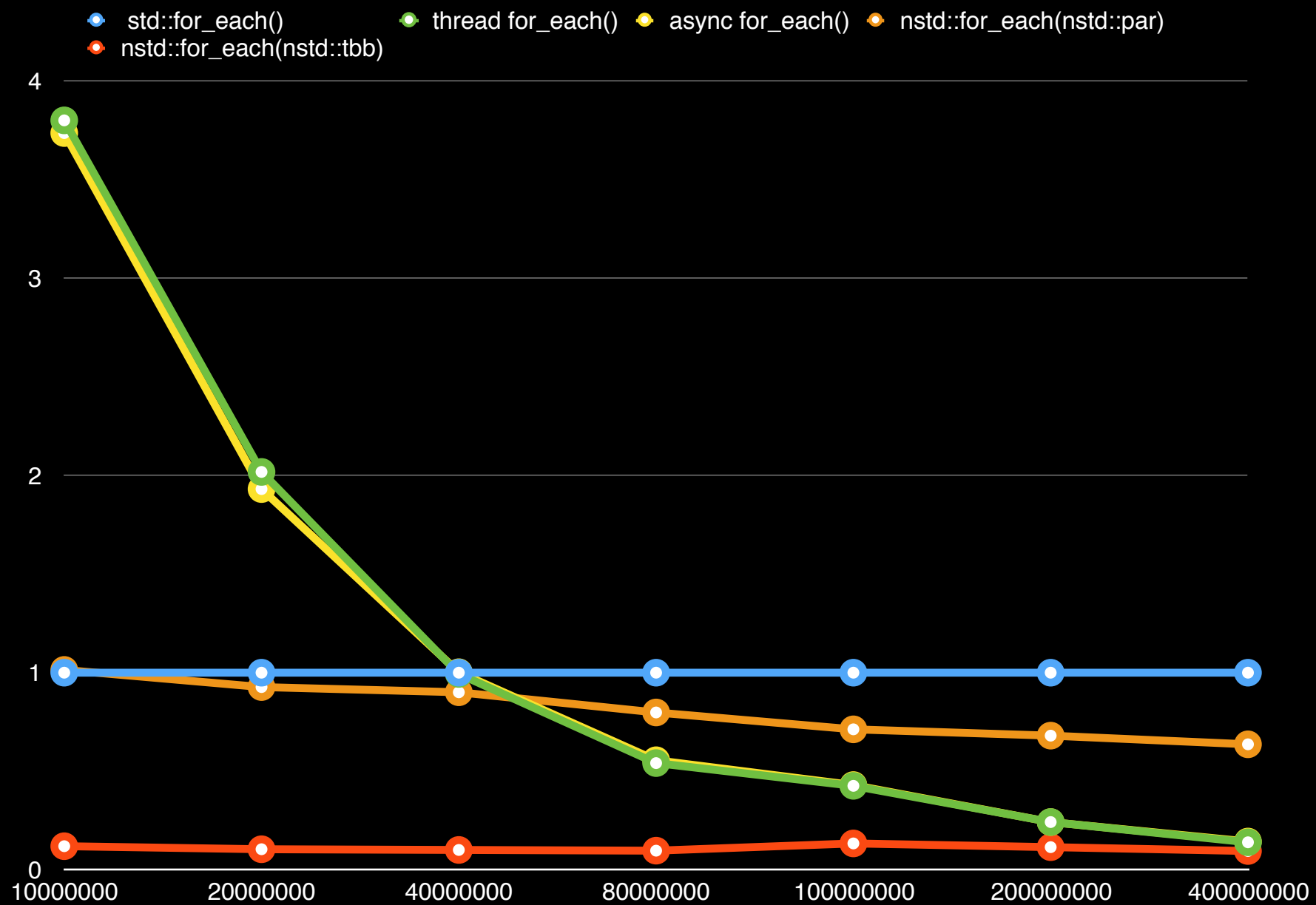
Results: map

```
for (; it != end; ++it) {  
    *it *= 17;  
}
```

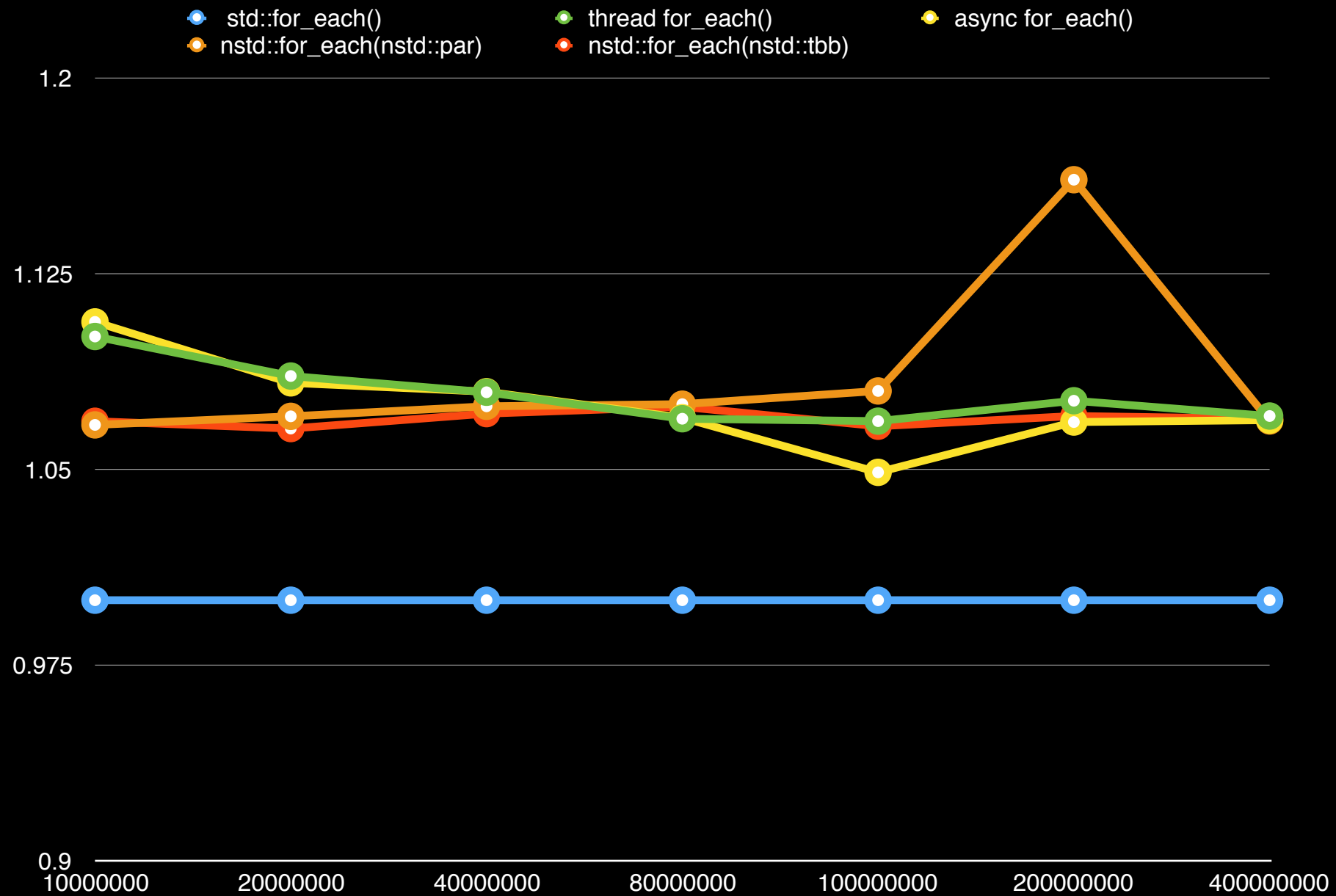
Implementations

- `std::for_each()`: sequential base line
- `thread for_each()`: use threads (see earlier slide)
- `async for_each()`: use `async` (see earlier slide)
- `....::par`: home grown using a thread pool
- `....::tbb`: home grown wrapper of `tbb`

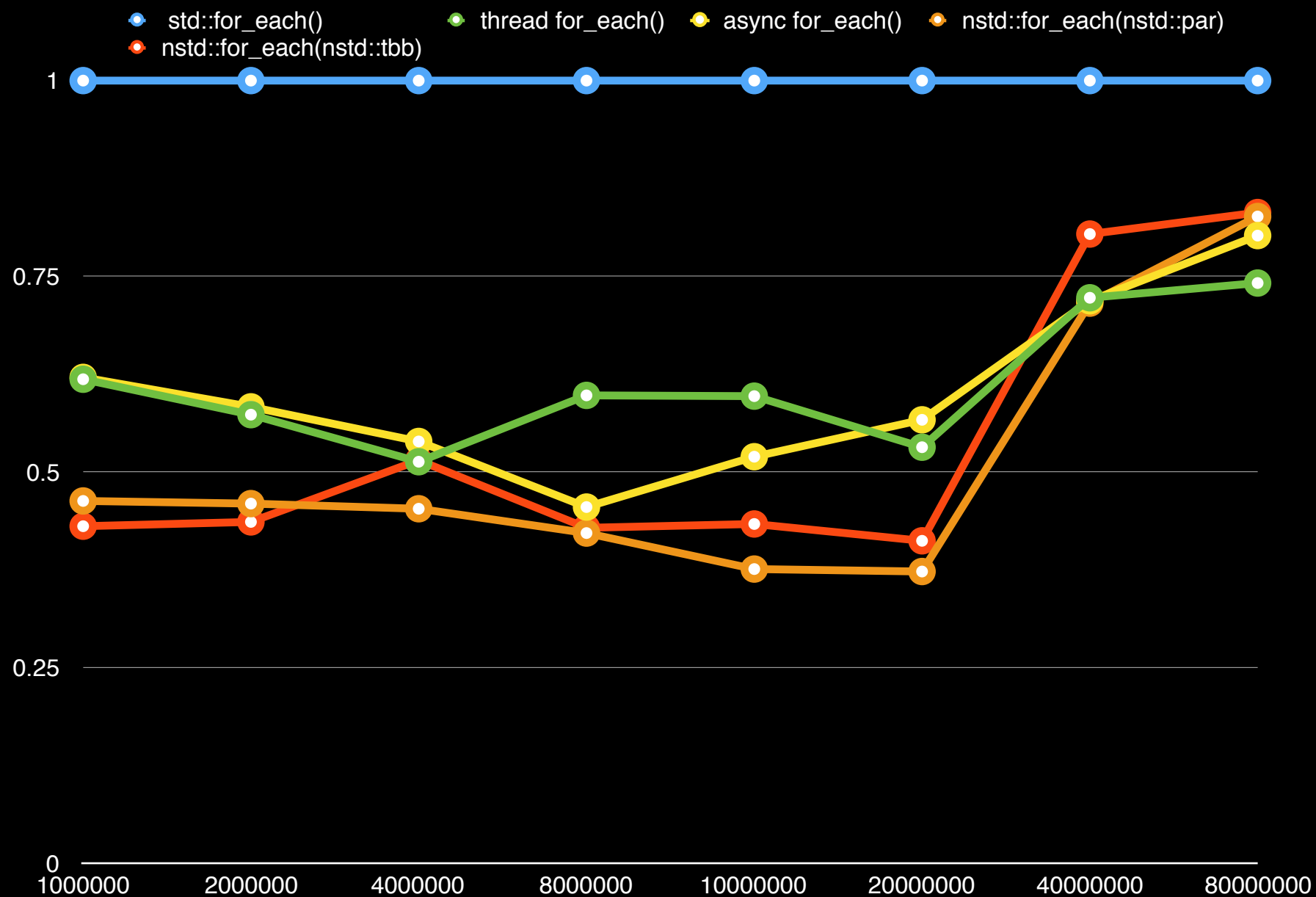
Results: map gcc phi



Results: map gcc 17



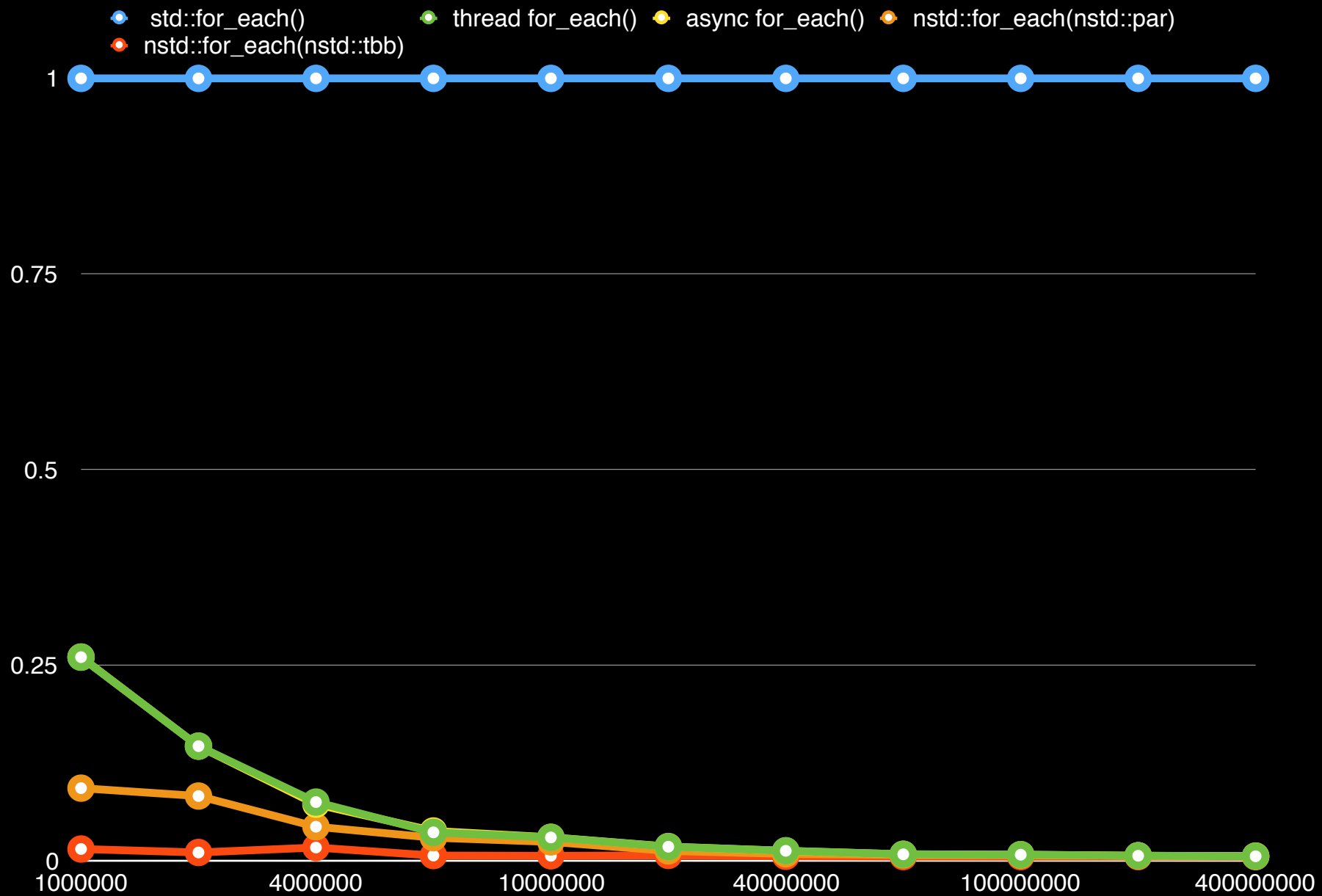
Results: map gcc ARM



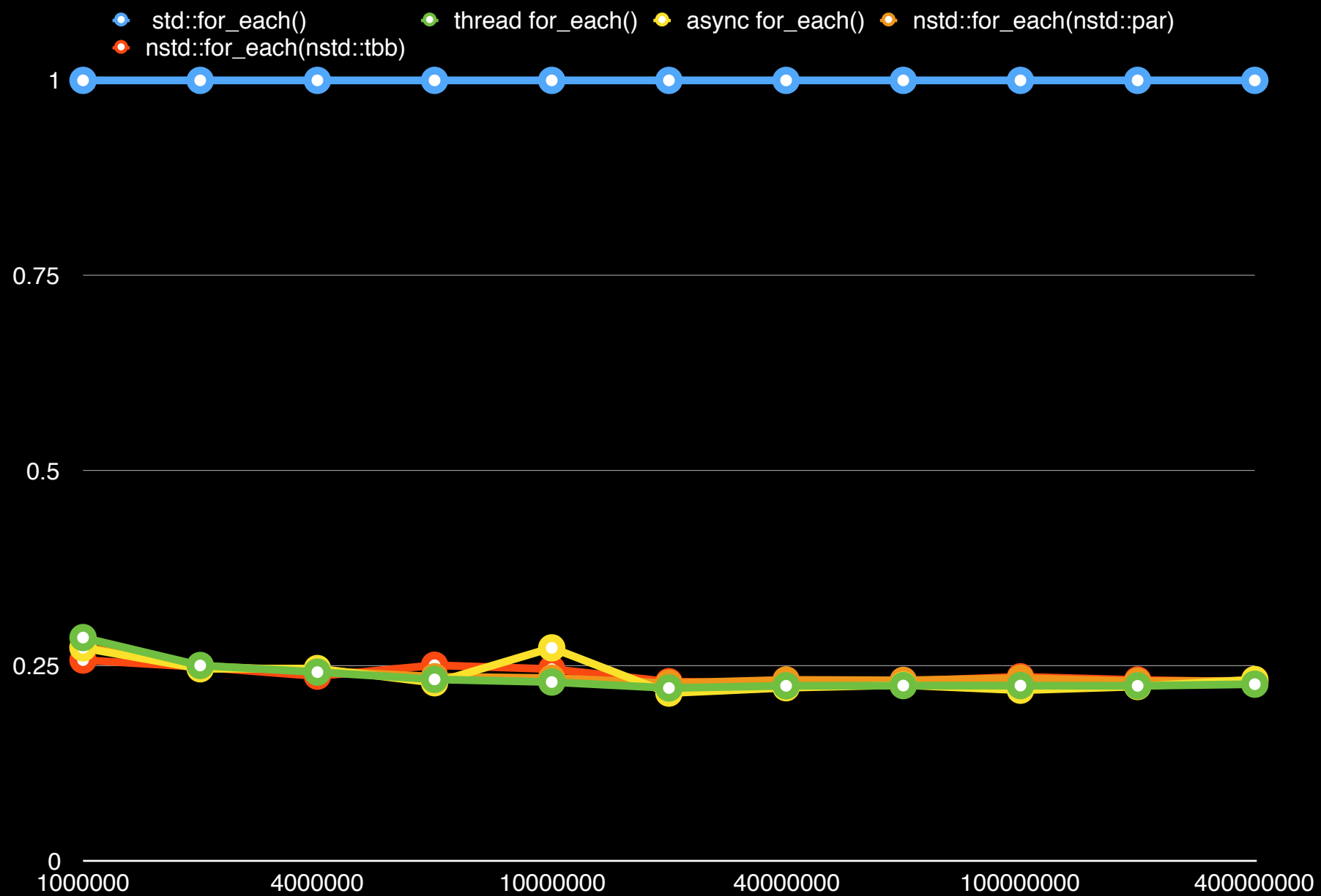
Results: map

```
for (; it != end; ++it) {  
    constexpr int max(2000);  
    std::complex<double> p(2.5 * *it / s - 0.5, 0.001);  
    int count(0);  
  
    for (std::complex<double> v(p);  
         norm(v) < 4.0 && count != max; ++count) {  
        v = v * v - p;  
    }  
    *it = count;  
}
```

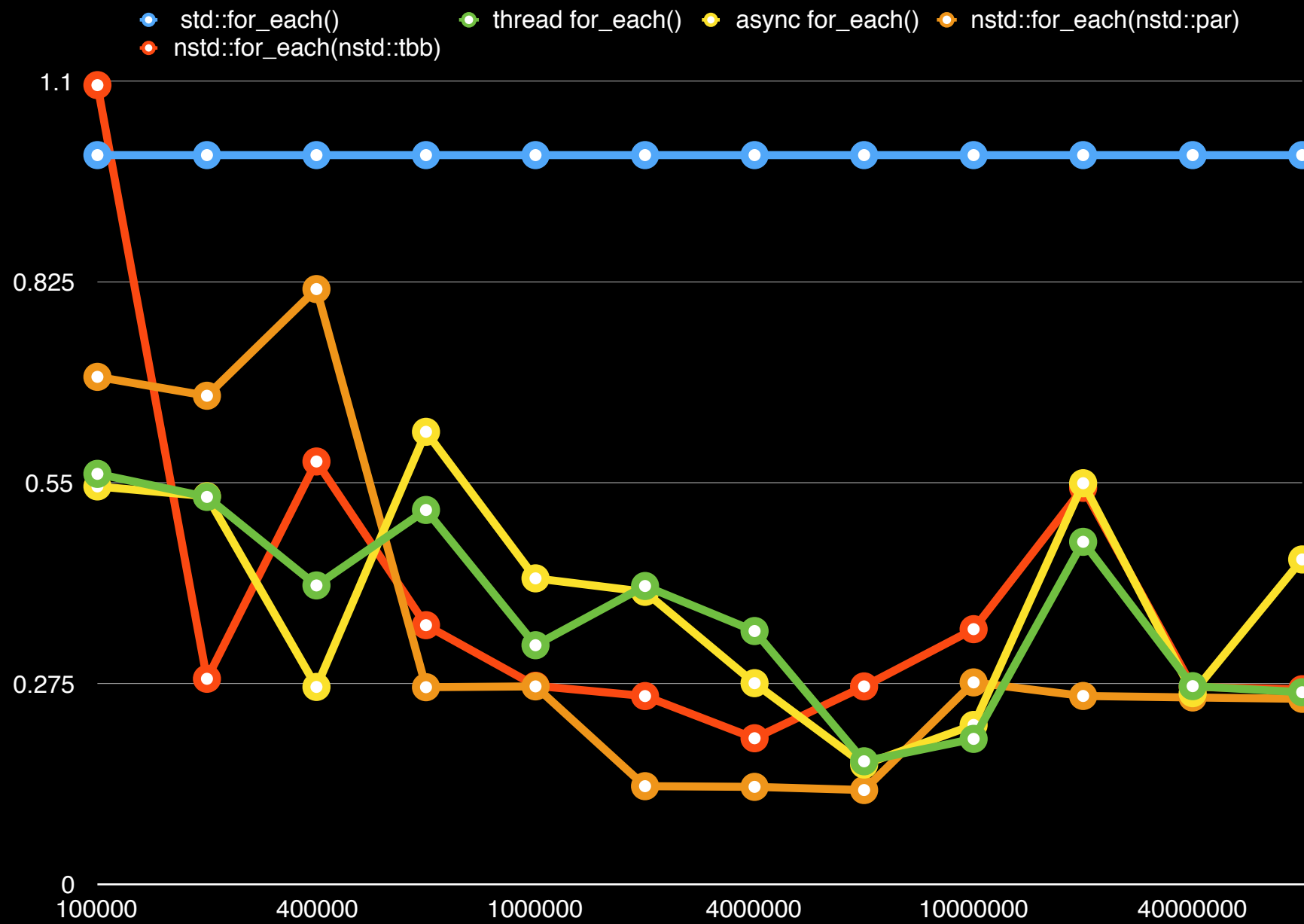
Results: work gcc phi



Results: work gcc 17



Results: work gcc ARM



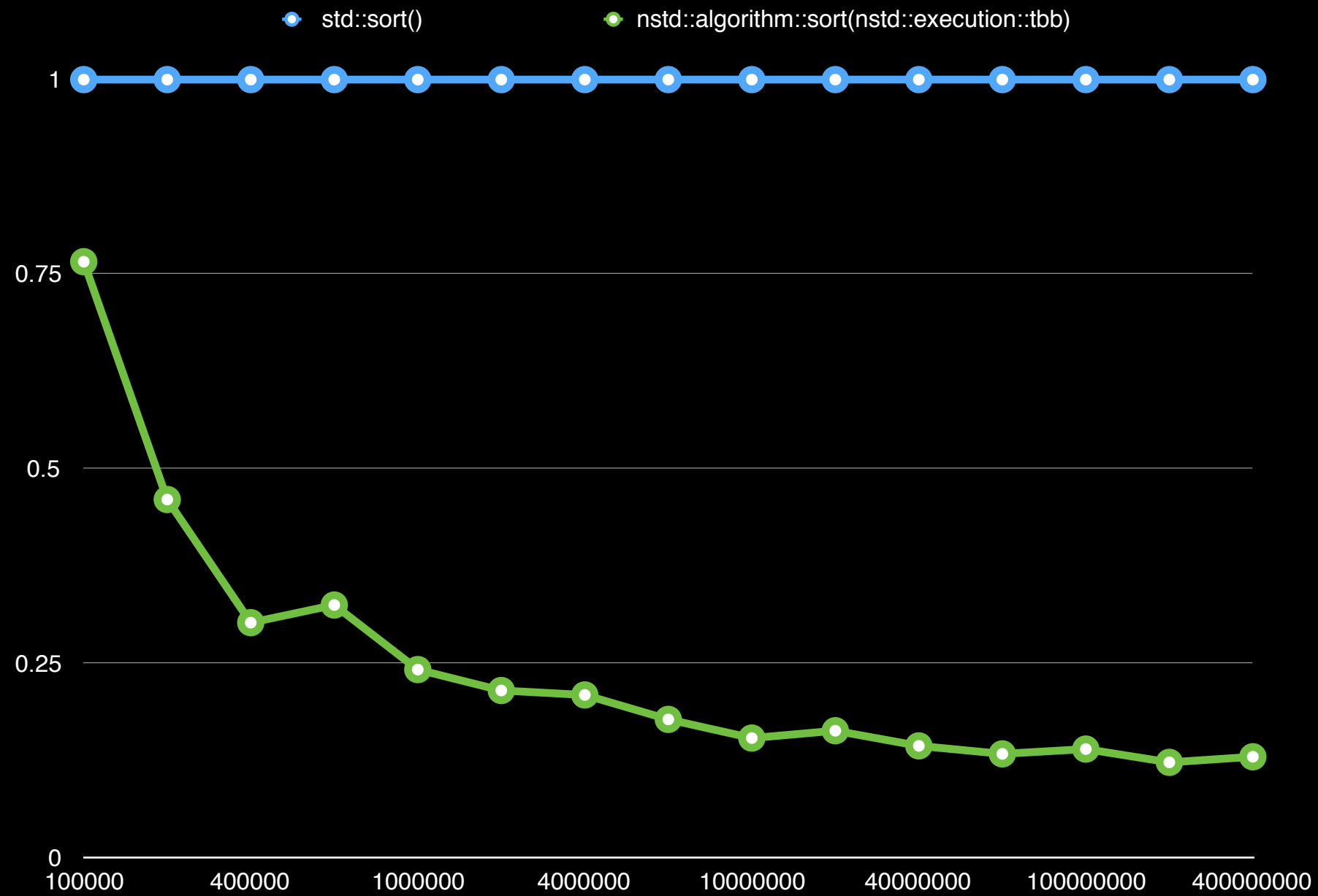
Results: sort

```
sort(begin, end);
```

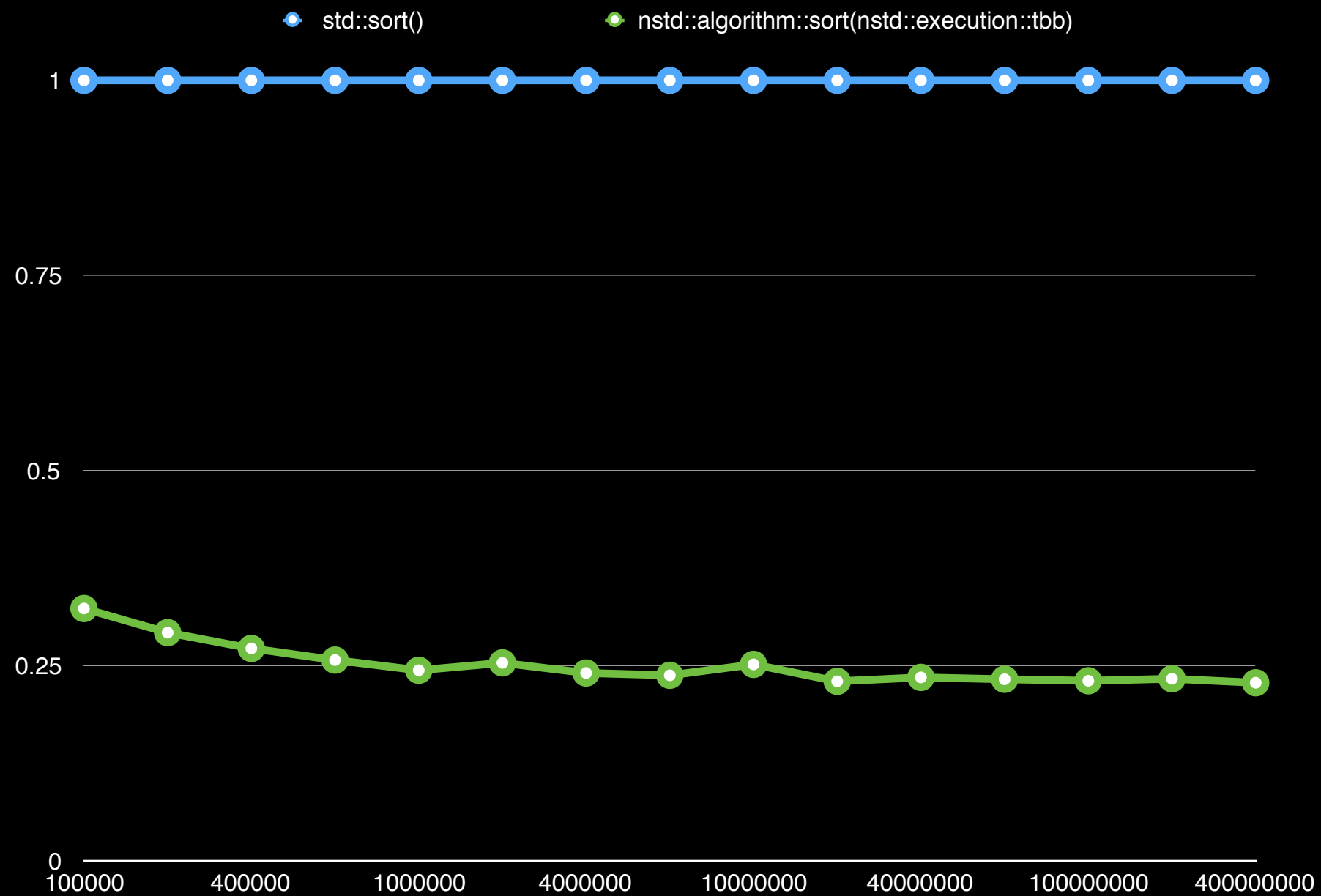
Implementations

- `std::sort()`: sequential base line
- `....::tbb`: home grown wrapper of tbb

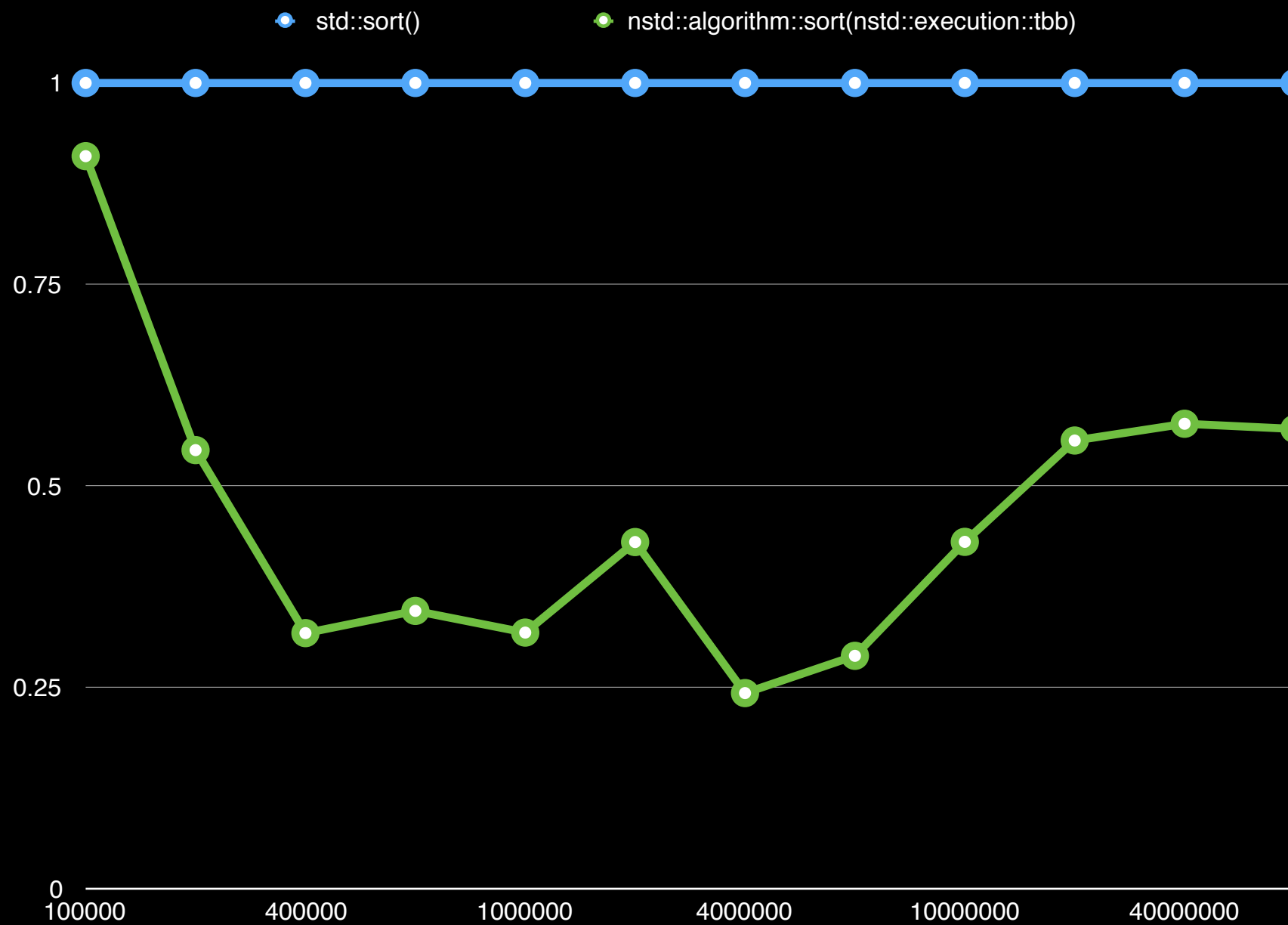
Results: sort gcc phi



Results: sort gcc 17



Results: sort gcc ARM



Results: reduce

```
accumulate(begin, end);
```

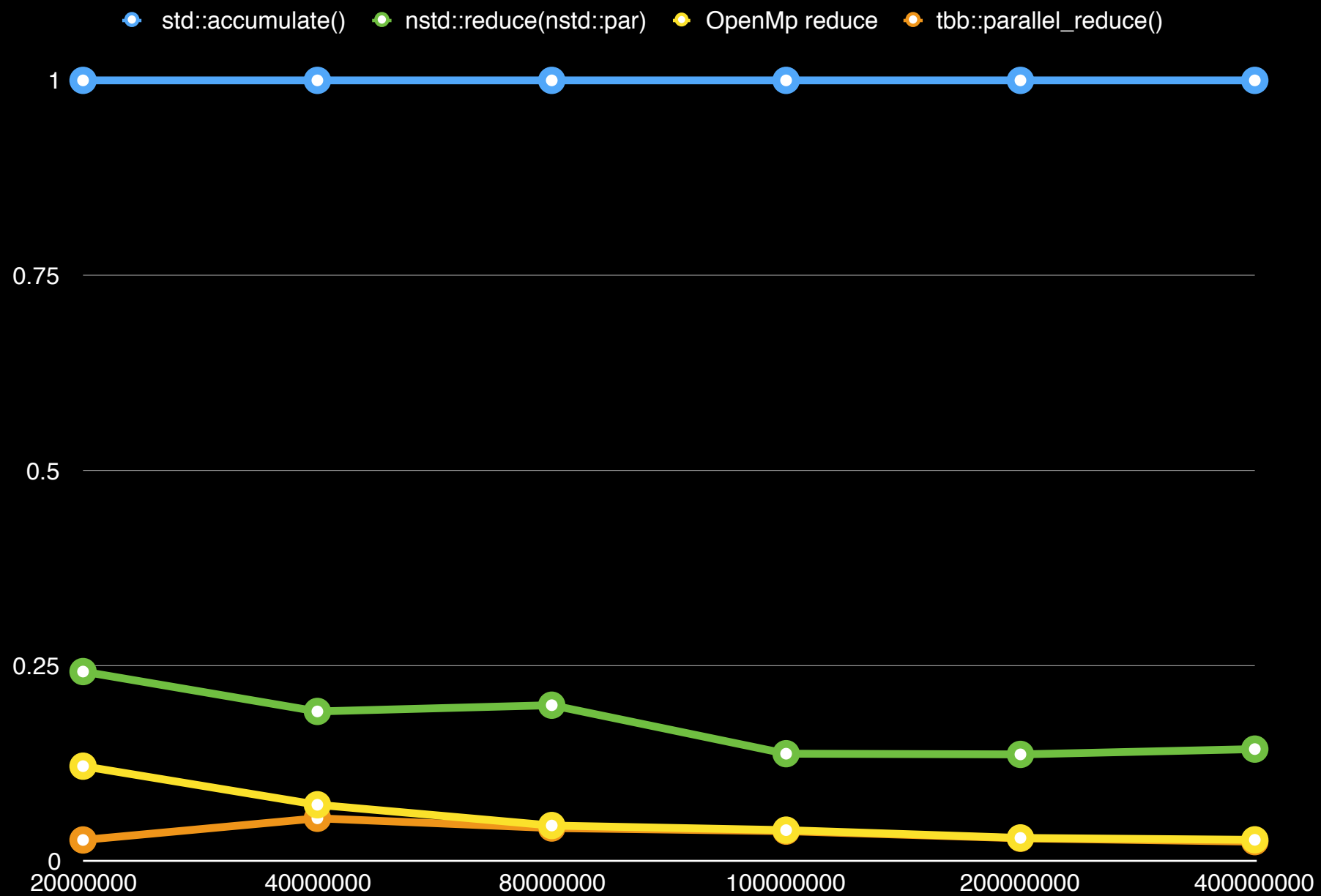

Results: reduce

```
reduce(begin, end);
```

Implementations

- `std::accumulate()`: sequential base line
- `....::par`: home grown using a thread pool
- `....::omp`: home grown wrapper using openmp
- `....::tbb`: home grown wrapper of tbb

Results: redu gcc phi



Results: redu gcc 17



Results: redu gcc ARM



Usage Guidance

- use random access iterator if at all possible
 - for the time being the only option anyway
- it isn't worth parallelising small operations
 - sequence needs to be large
 - operations need to be expensive

Future Directions

- more execution policies
- integration with executors
- continuation/future support
- some control over chunking

Conclusions

- using STL algorithms is good
- parallel algorithms work best
 - on random access sequences
 - with large ranges
 - expensive operations

Questions

