Parallel Algorithms

Dietmar Kühl
Bloomberg LP

WARNING!

- this talk is **not** about how to implemented parallel algorithms but about how to use them
- this talk is about C++

Copyright Notice

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
- 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), buf);
    ts.emplace_back([=](){
        for (; it != e; ++it, ++to) {
            *to = fun(*it);
    }});
}
for (auto& t: ts) { t.join(); }
```

- not easy to use

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), buf);
    fs.emplace_back(std::async([=](){
        for (; it != e; ++it, ++to) {
            *to = fun(*it);
    }}));
}
for (auto& f: fs) { f.get(); }
```

- not easy to use

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::par, begin, end, to, fun);

- can use different policies (std::seq, std::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::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::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::sequenced_policy std::seq
- std::parallel_policy std::par
- std::parallel_unsequenced_policy std::par_unseq

std::seq

- sequential execution
- primarily intended for debugging
- same common constraints and interface changes
  - exceptions result in std::terminate()
  - no [required] support for input iterators
  - changed return types

std::par

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

std::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

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
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

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
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: 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
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_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: 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
replace
remove_if
replace_copy
replace_copy_if
replace_if
reverse
rotate
reverse_copy
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

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
adjacent_find
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

Algorithms: reduce

- accumulate() becomes reduce()
- operations need to be associative
- find algorithms may omit using the entire range

Algorithms: scan

adjacent_difference

copy_if

exclusive_scan

inclusive_scan

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: scan

- partial_sum() becomes inclusive_scan()
- may produce different results when operation isn't associative
- inclusive_scan(): r[i] uses s[0],…, s[i]
- exclusive_scan(): r[i] uses s[0], …, 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

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

unique
unique_copy

Algorithms: special

adjacent_difference

inplace_merge

merge

nth_element

partial_sort
partial_sort_copy

partition

rotate

set_difference
set_intersection
set_symmetric_difference
set_union

sort

stable_partition
stable_sort

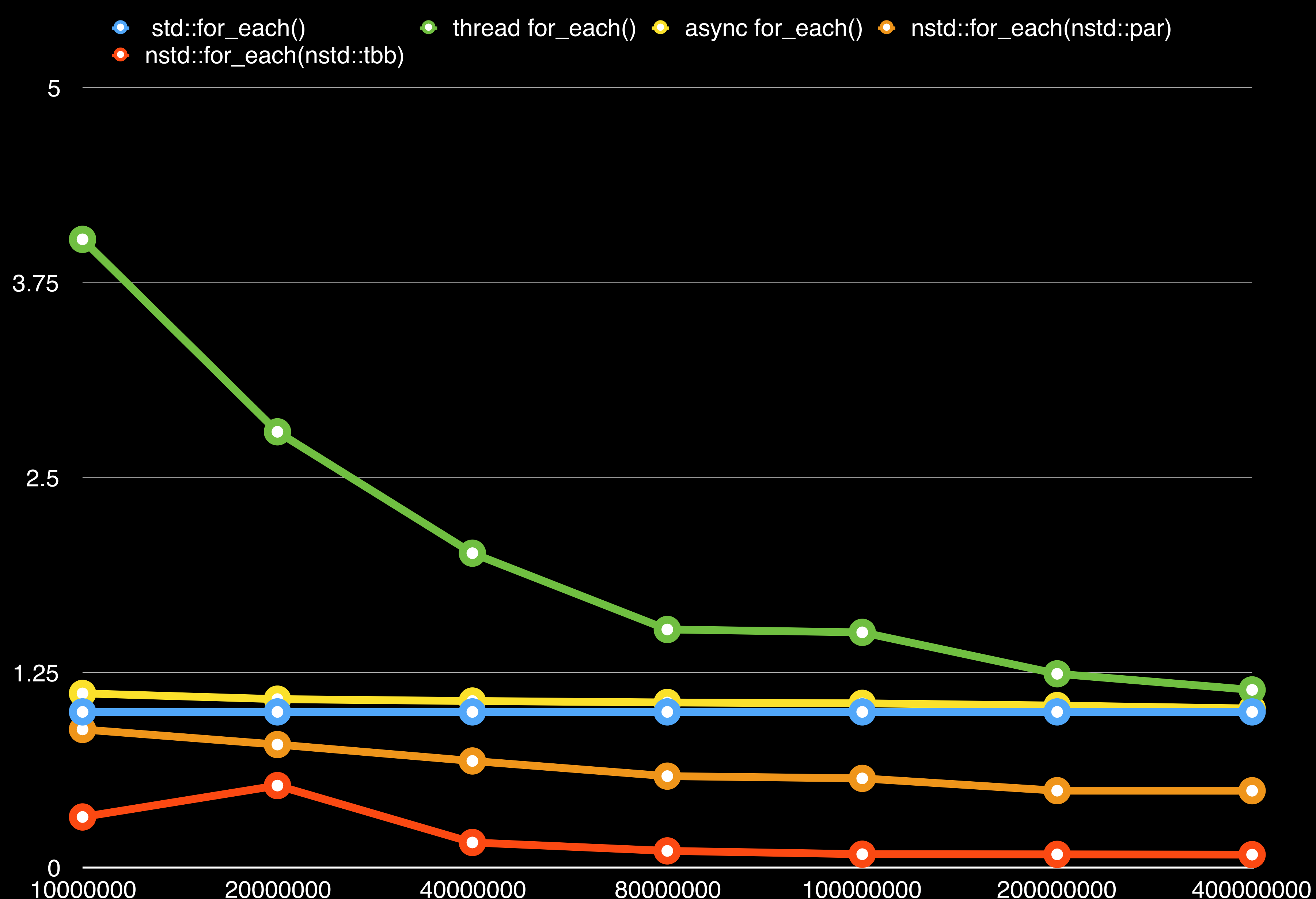
Results: Machines

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

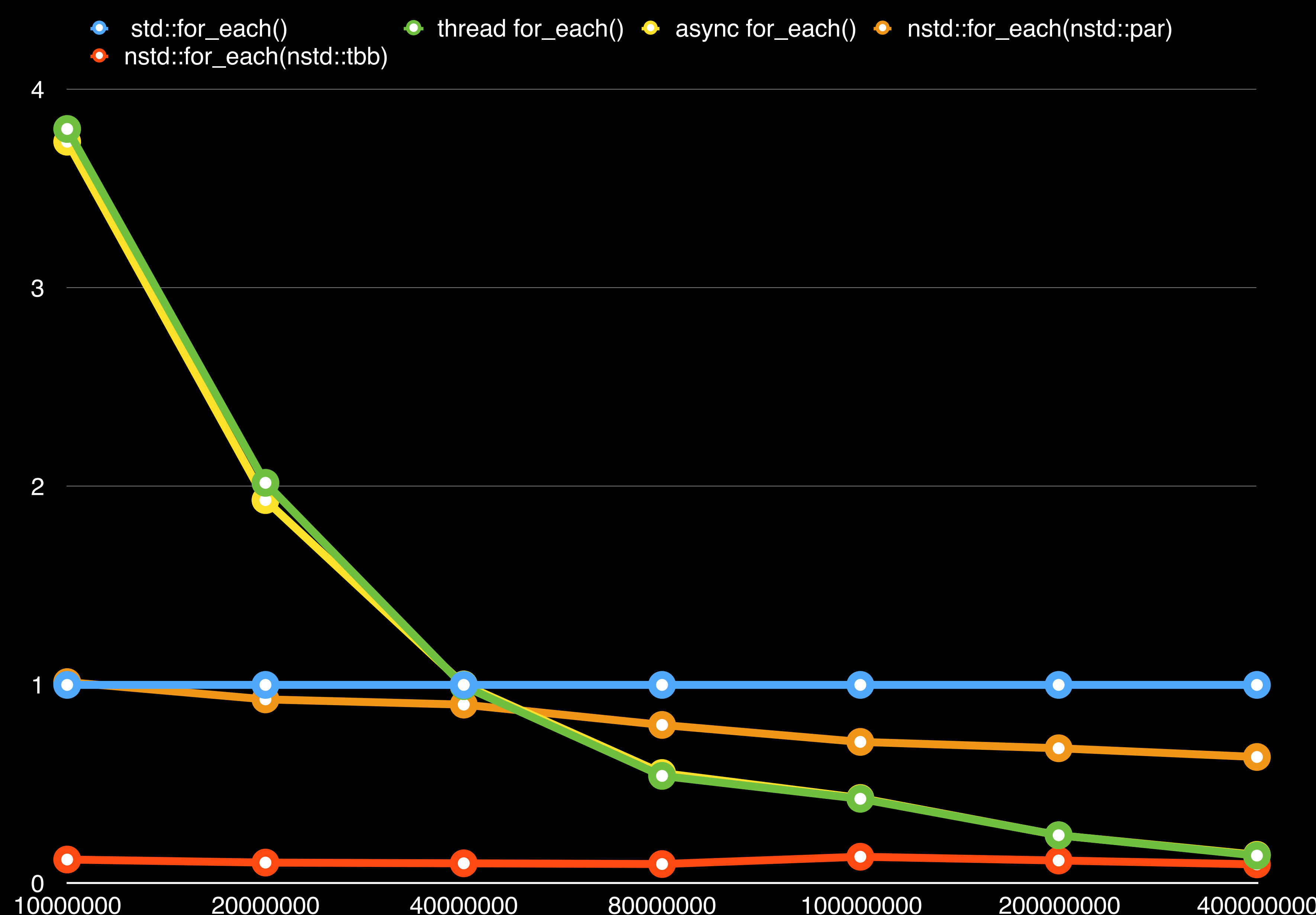
Results: map

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

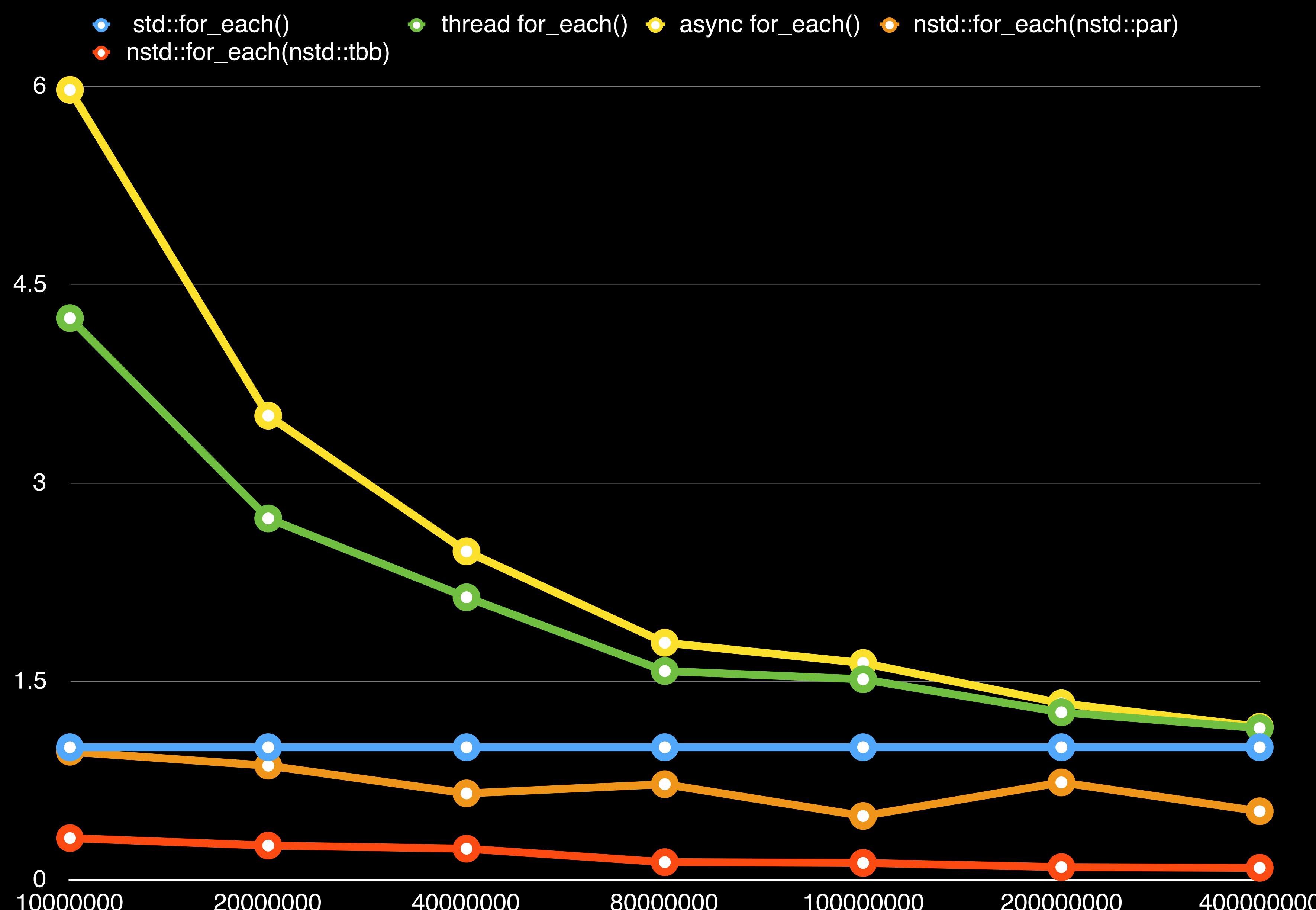
Results: map Intel phi



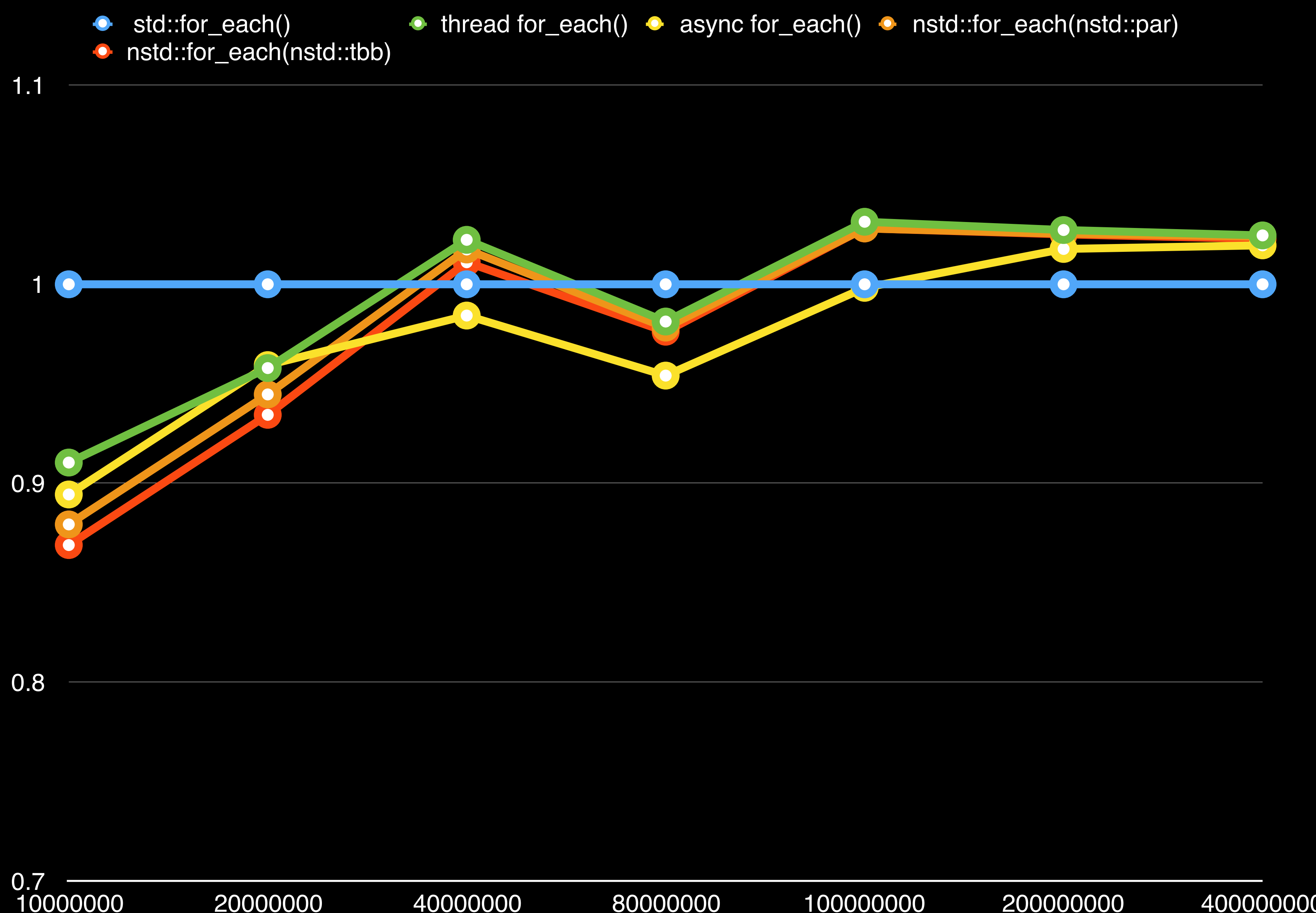
Results: map gcc phi



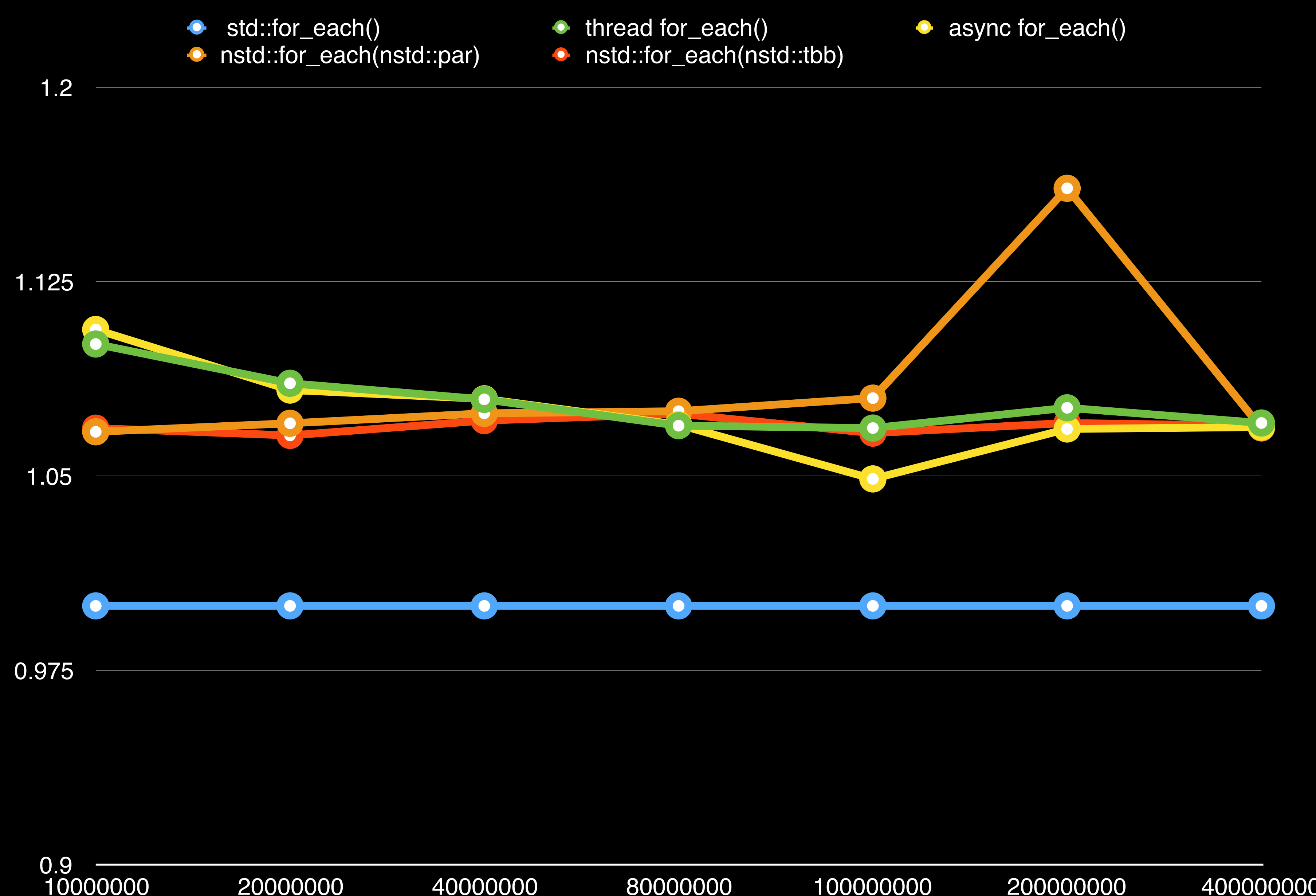
Results: map clang phi



Results: map Intel I7

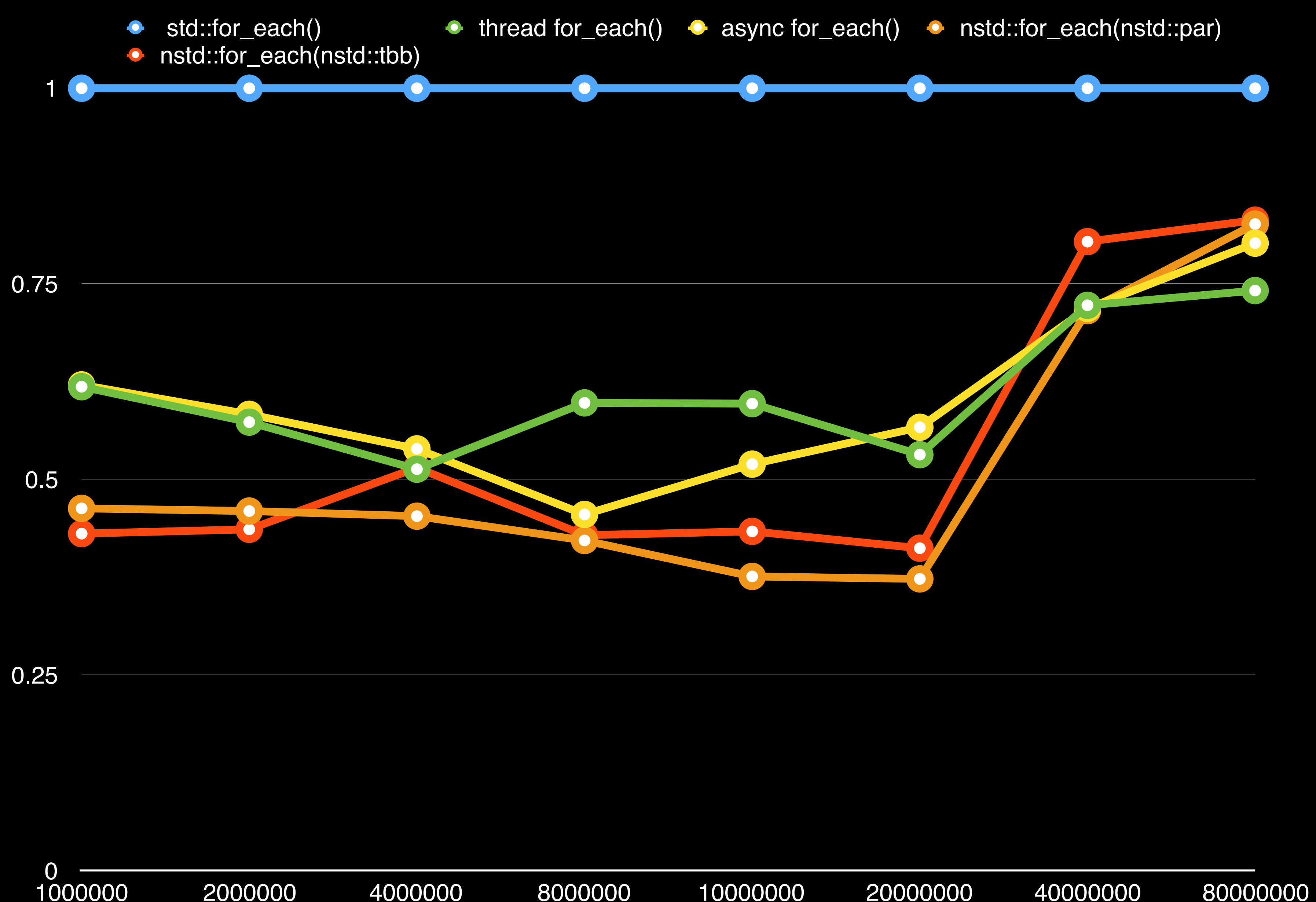


Results: map gcc I7

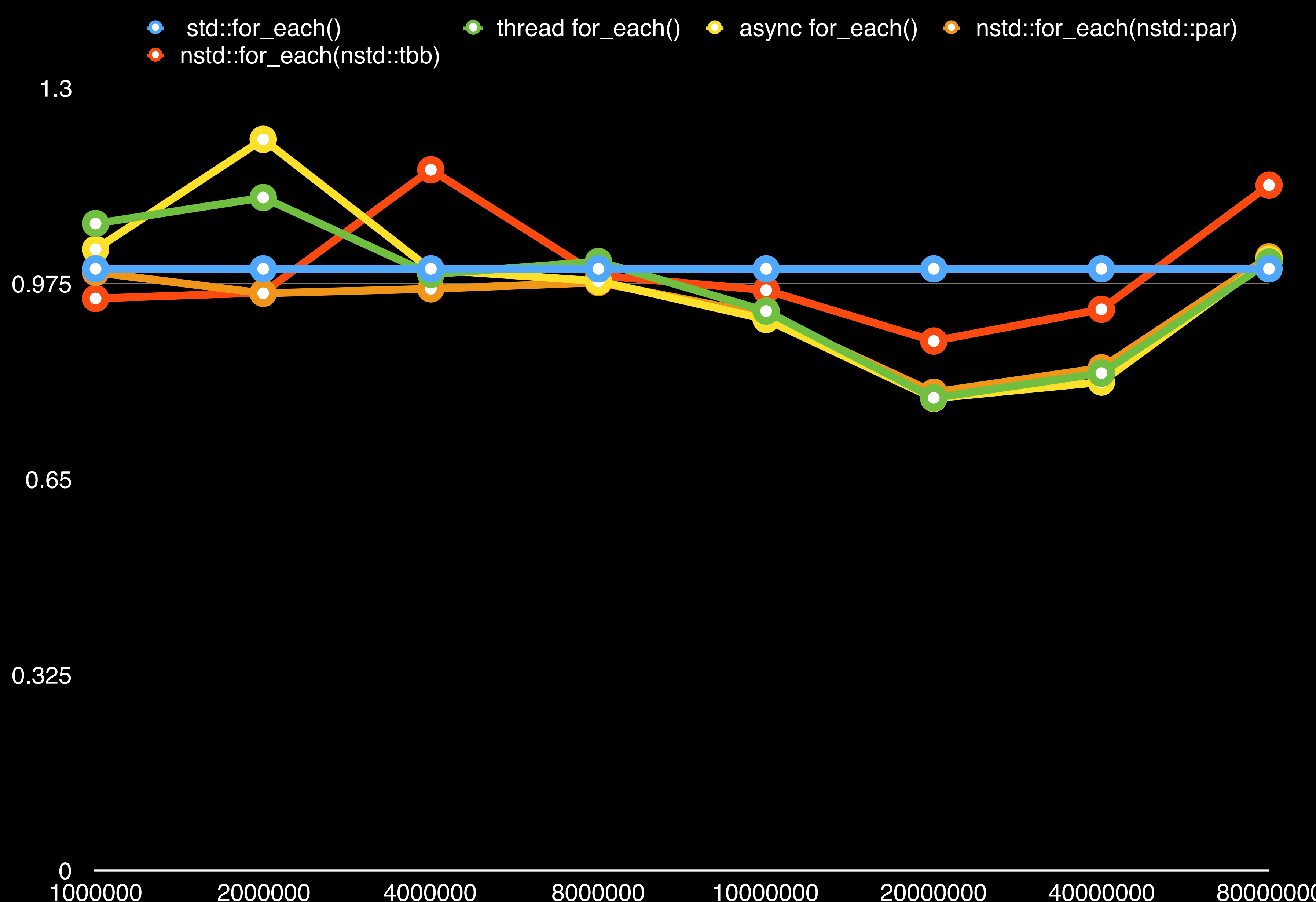


Results: map clang I7

Results: map gcc ARM



Results: map clang ARM

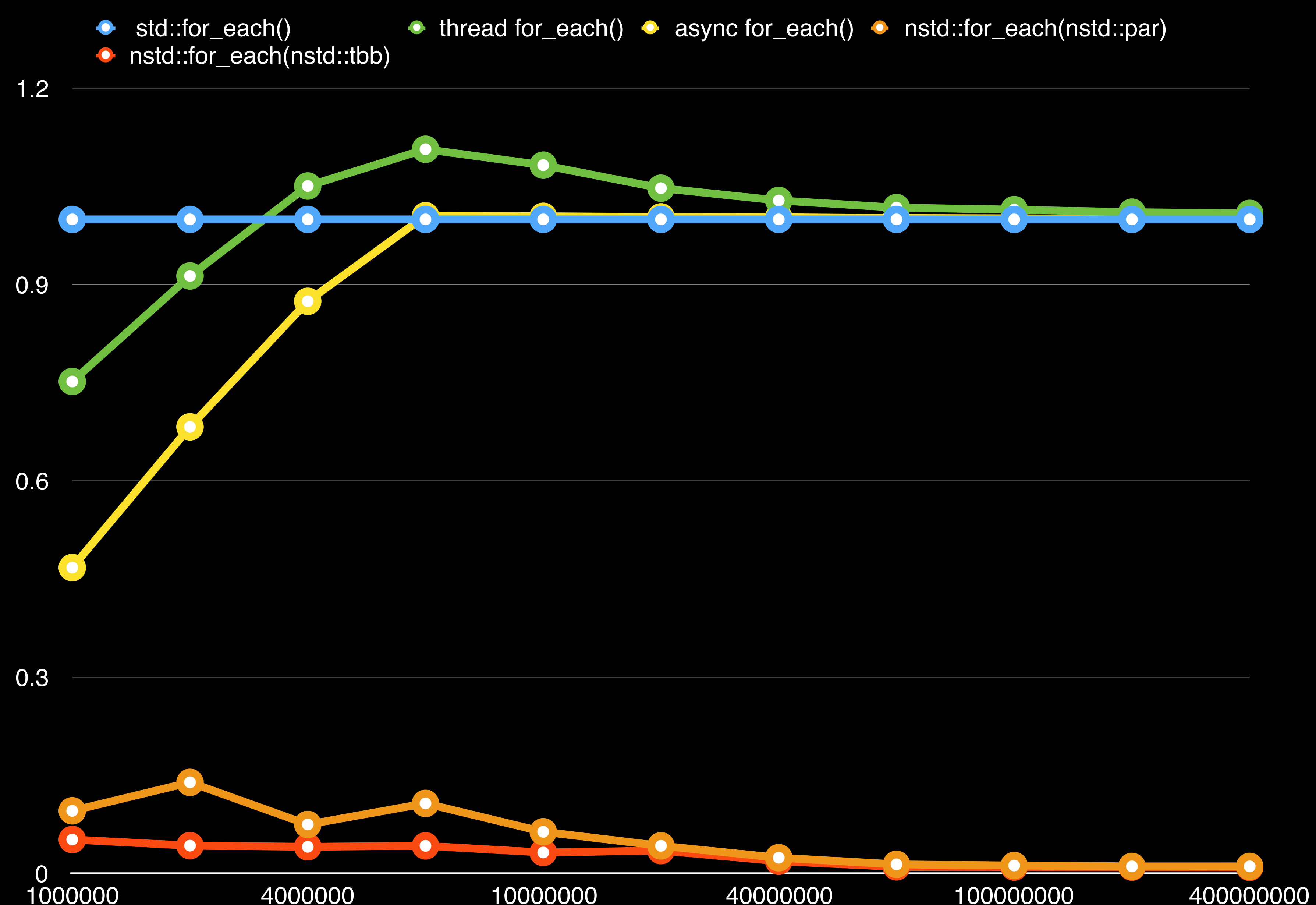


Results: map

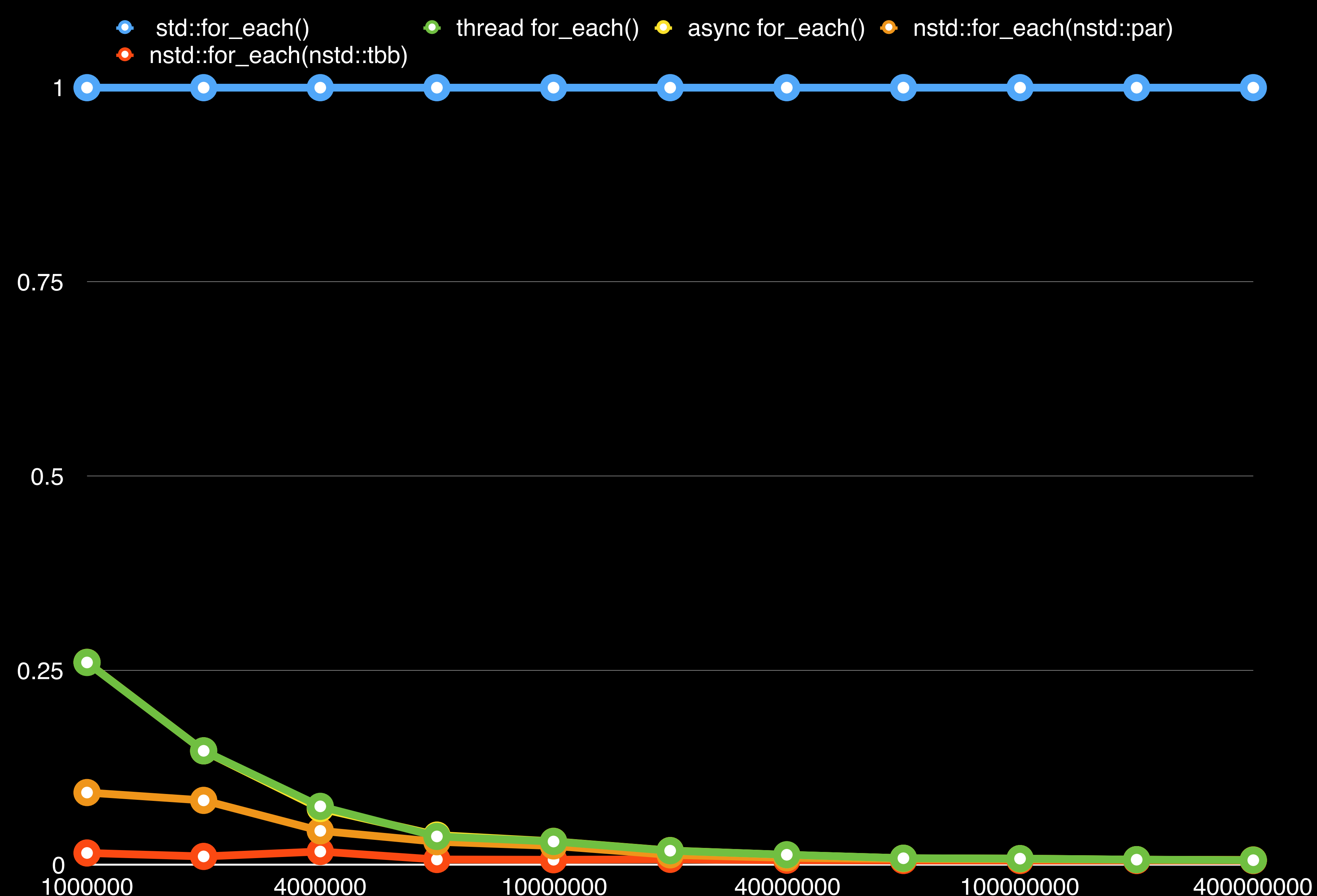
```
for (; it != end; ++it) {
  constexpr int max(2000);
  std::complex<double> p(2.5 * *it / s - 0.5, 0.001);
  int count(0);

  for (std::complex<double> v(p);
        norm(v) < 4.0 && count != max; ++count) {
      v = v * v - p;
  }
  *it = count;
}
```

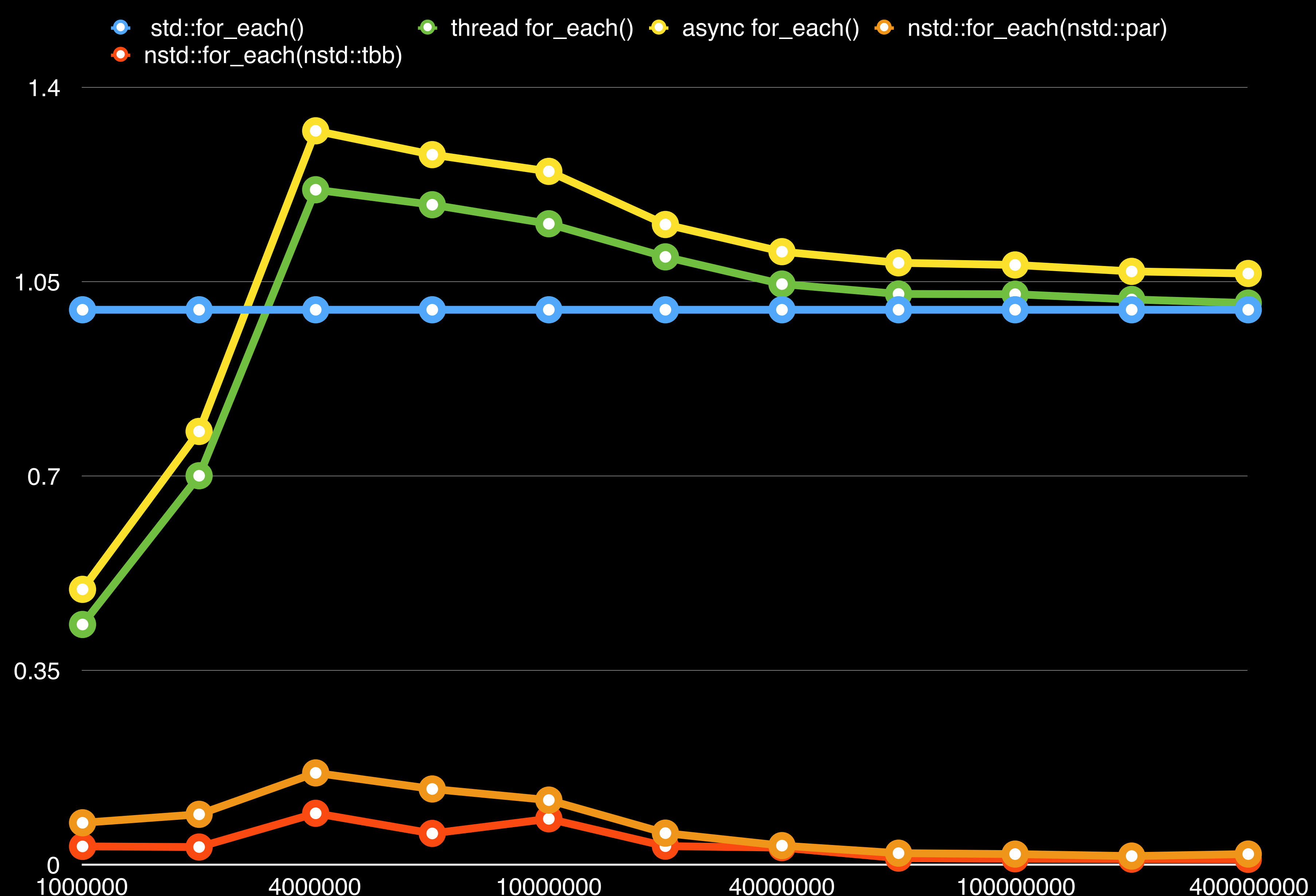
Results: work Intel phi



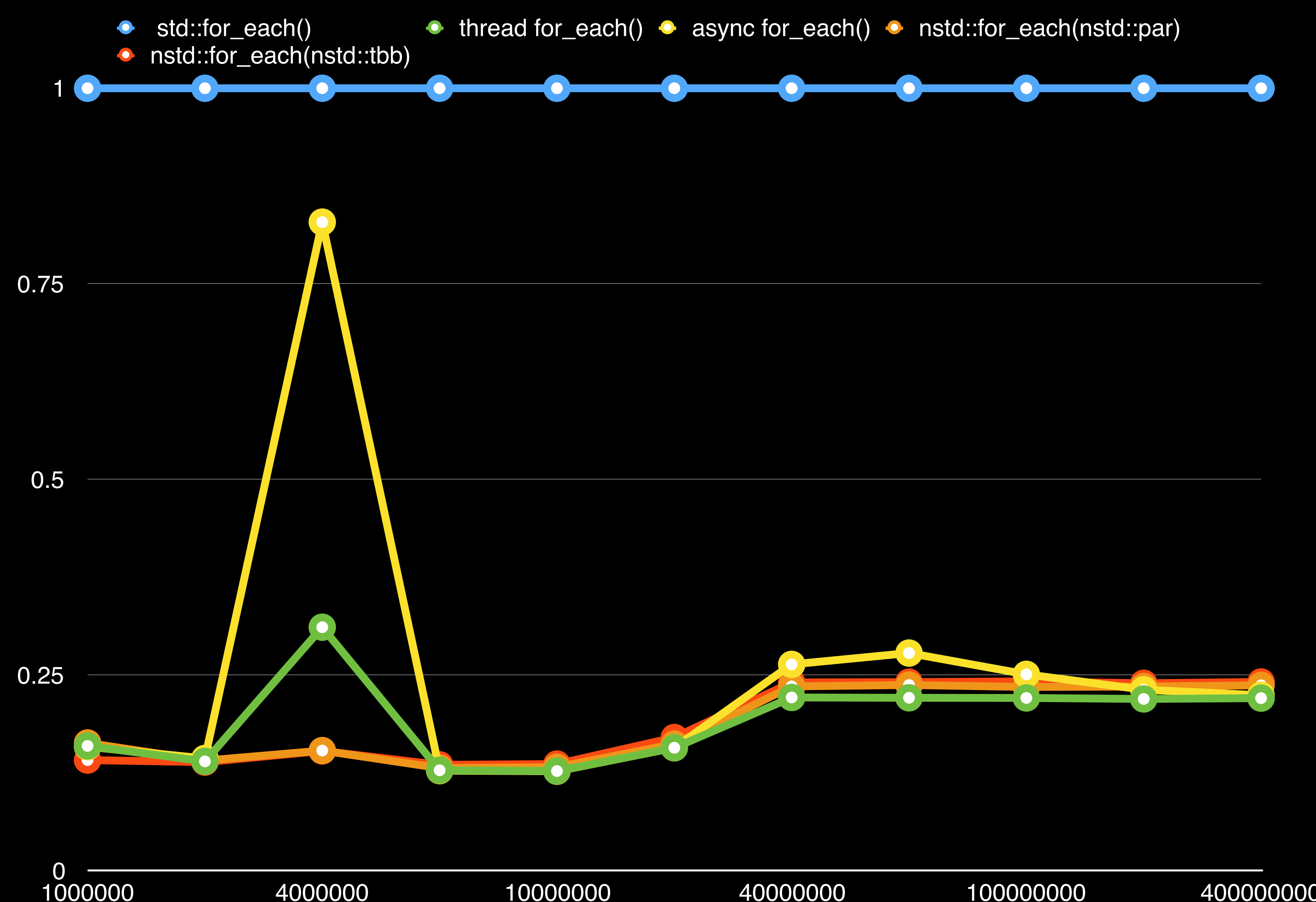
Results: work gcc phi



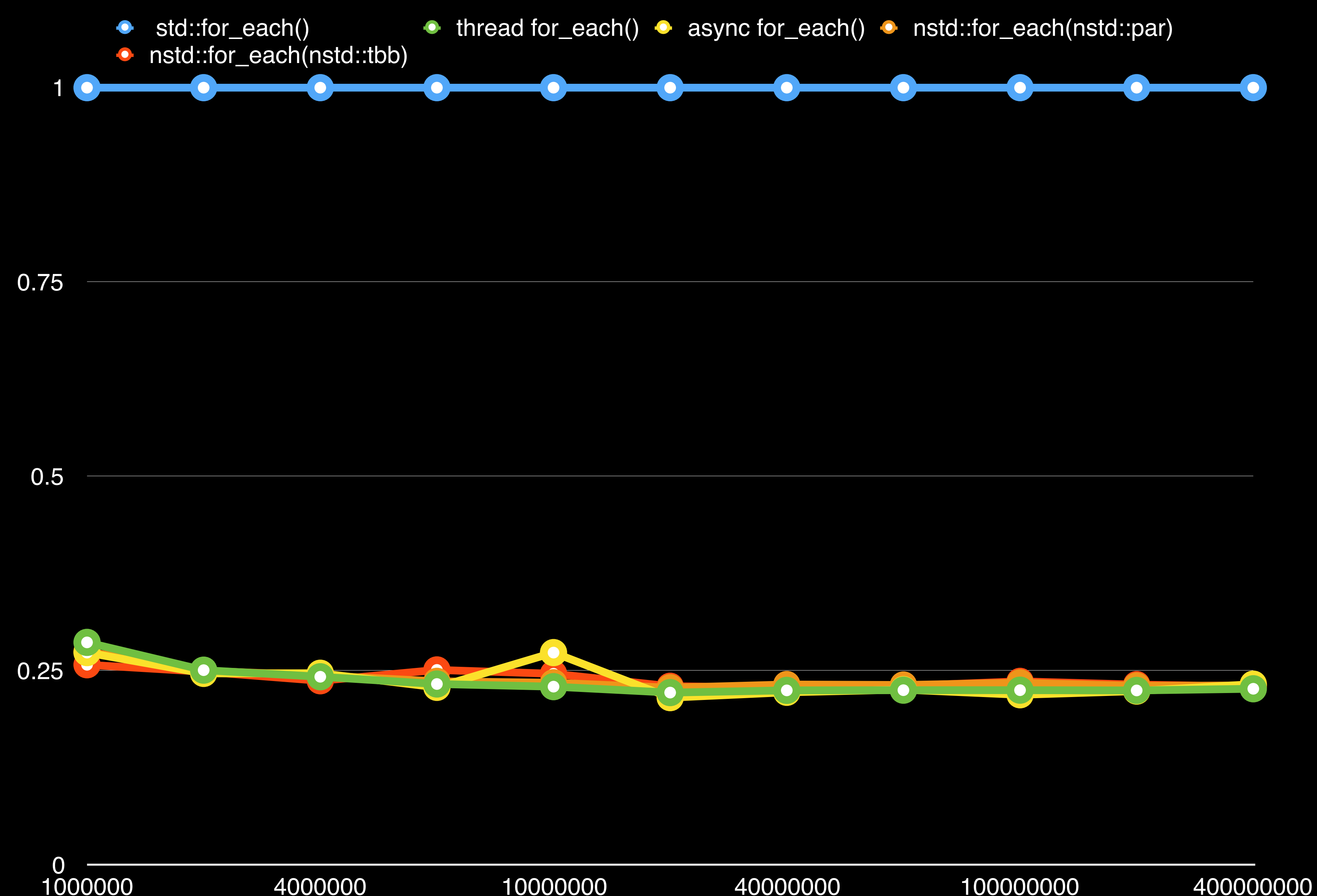
Results: work clang phi



Results: work Intel I7

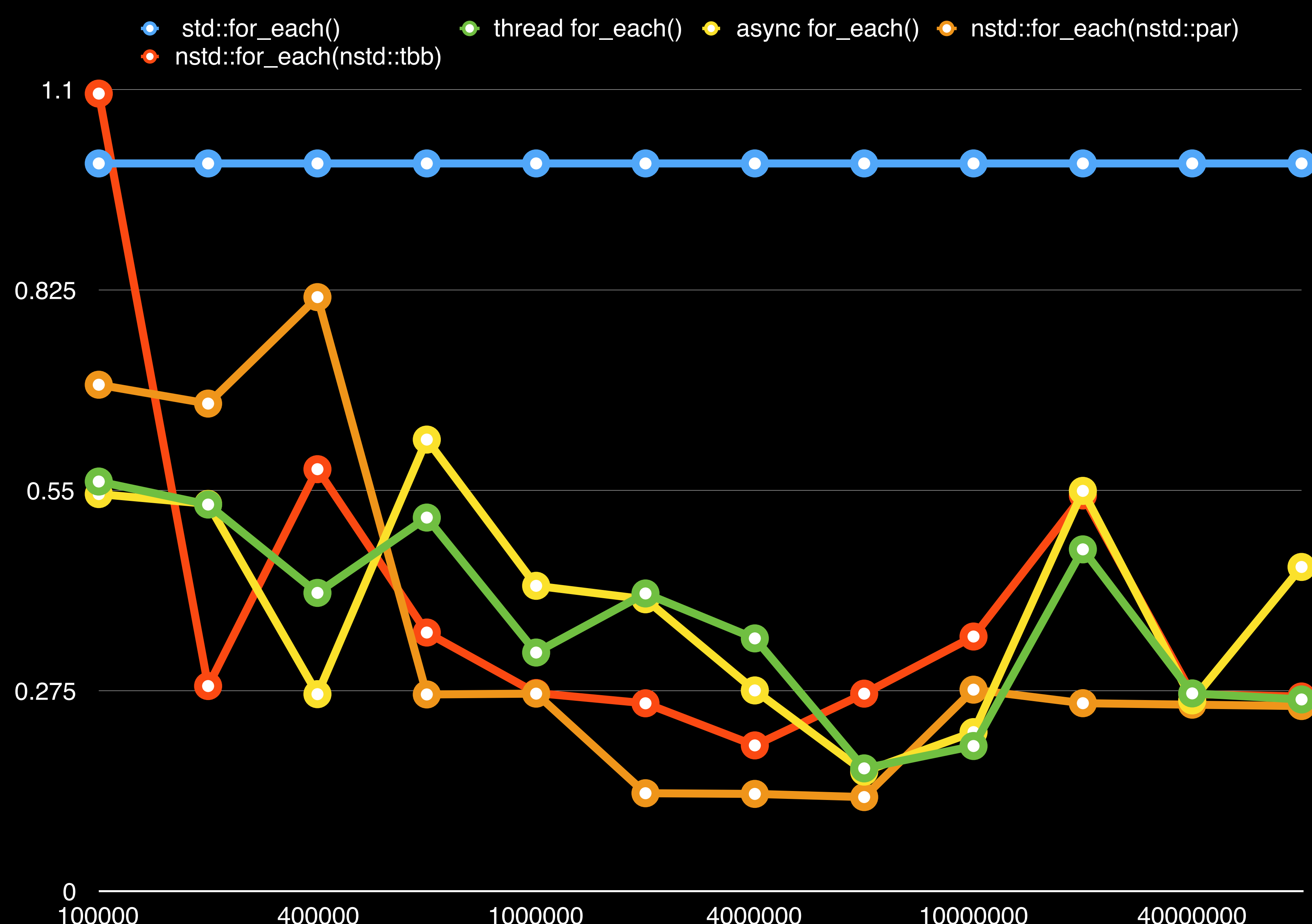


Results: work gcc I7

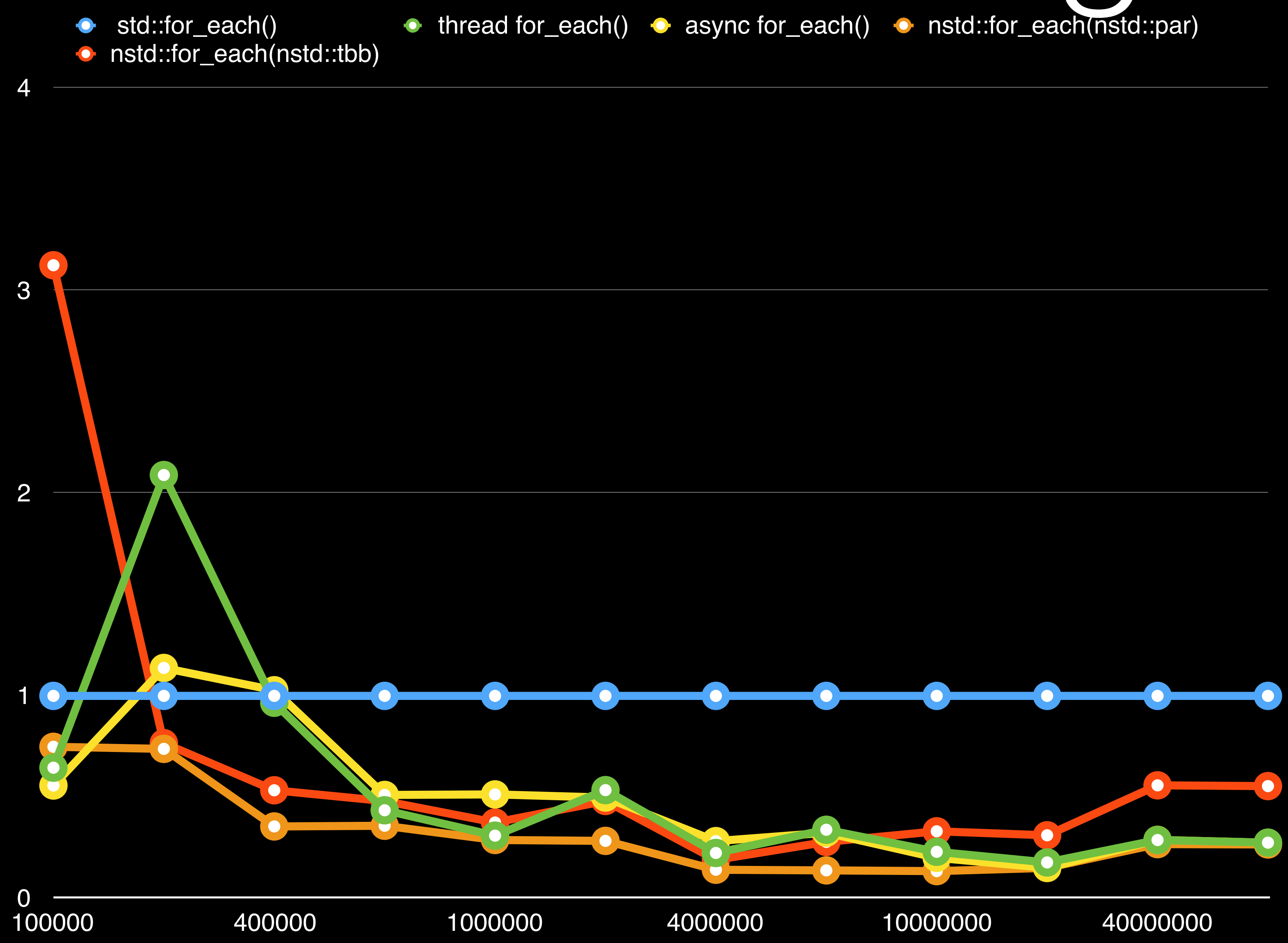


Results: work clang I7

Results: work gcc ARM



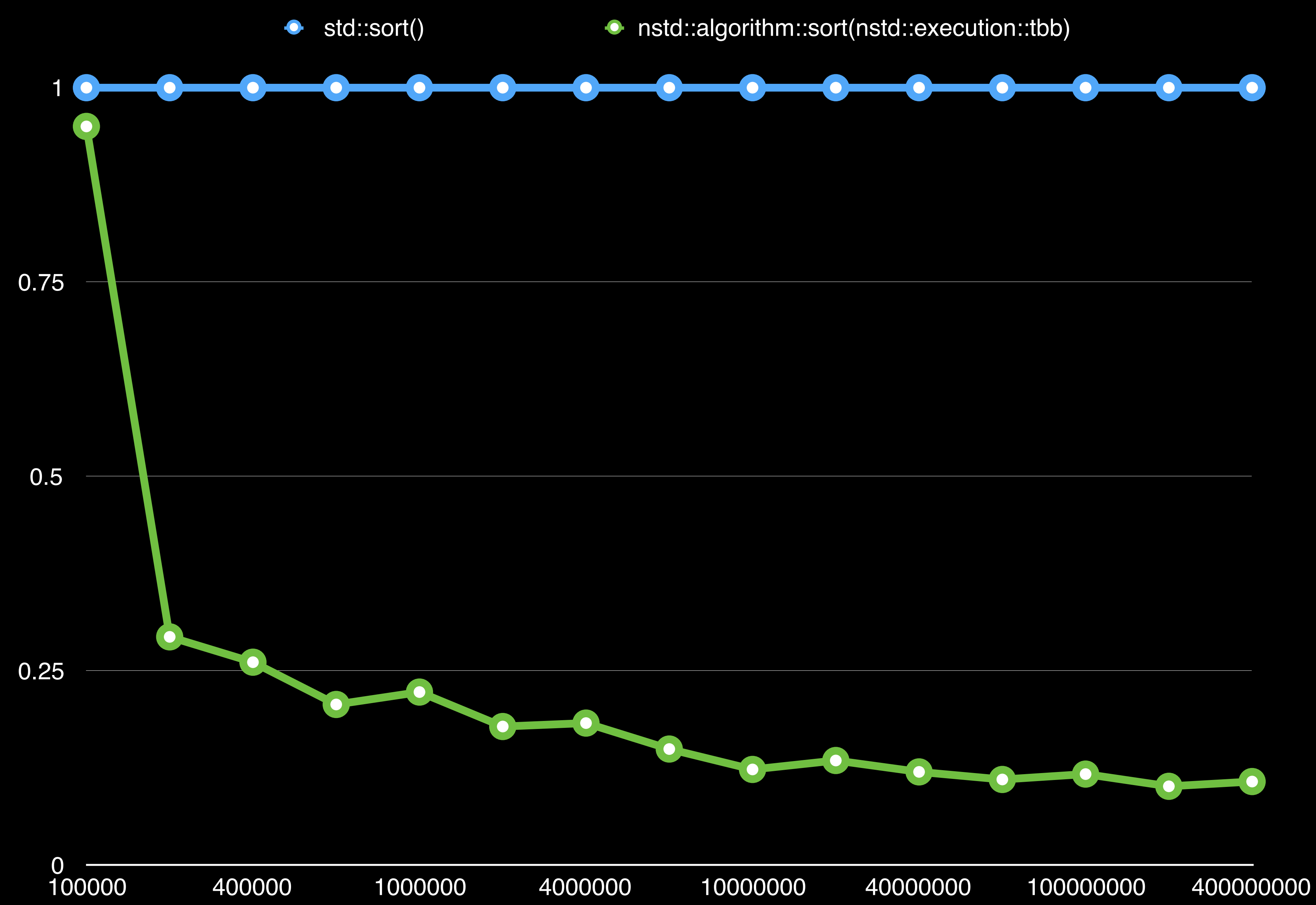
Results: work clang ARM



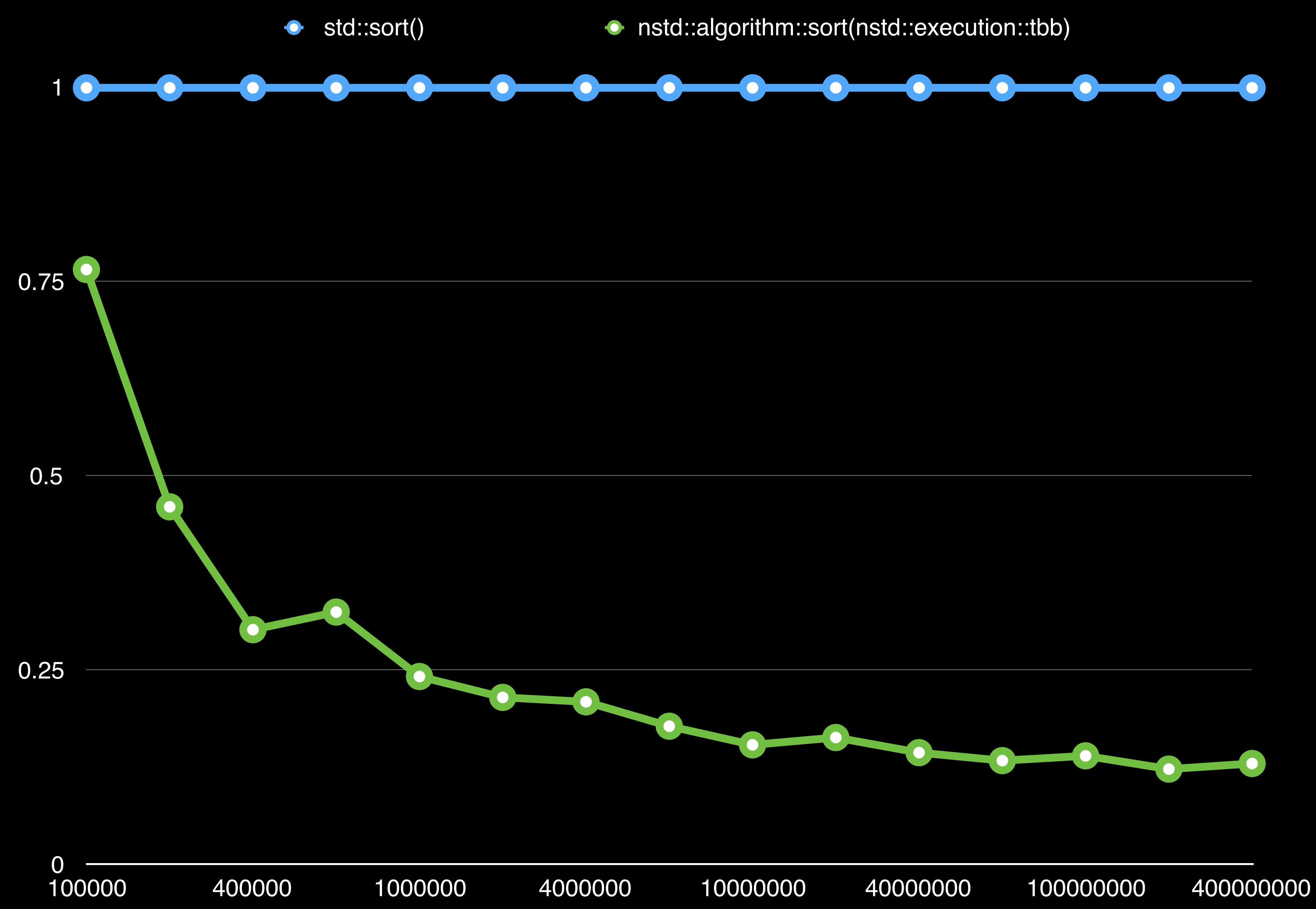
Results: sort

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

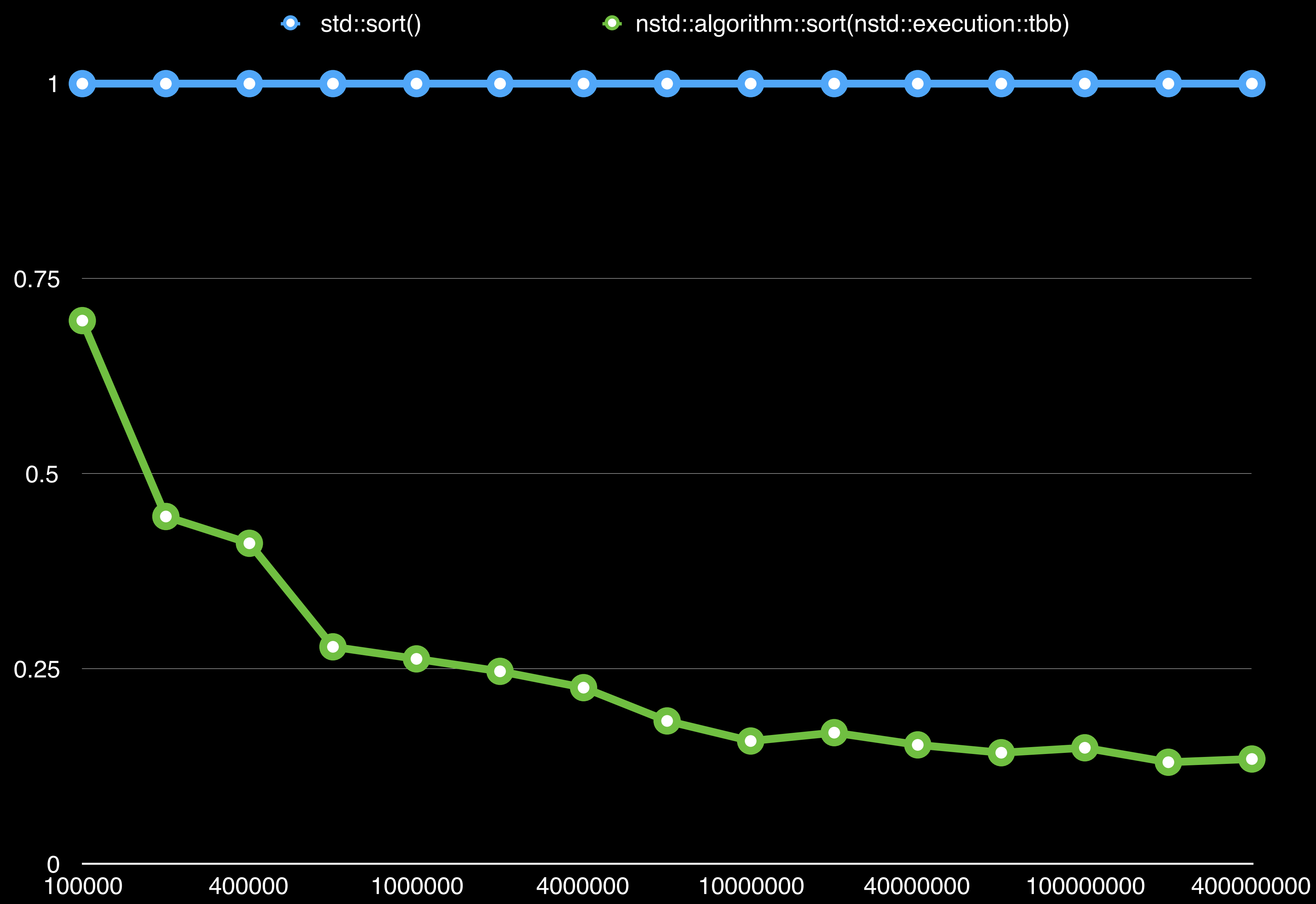
Results: sort Intel phi



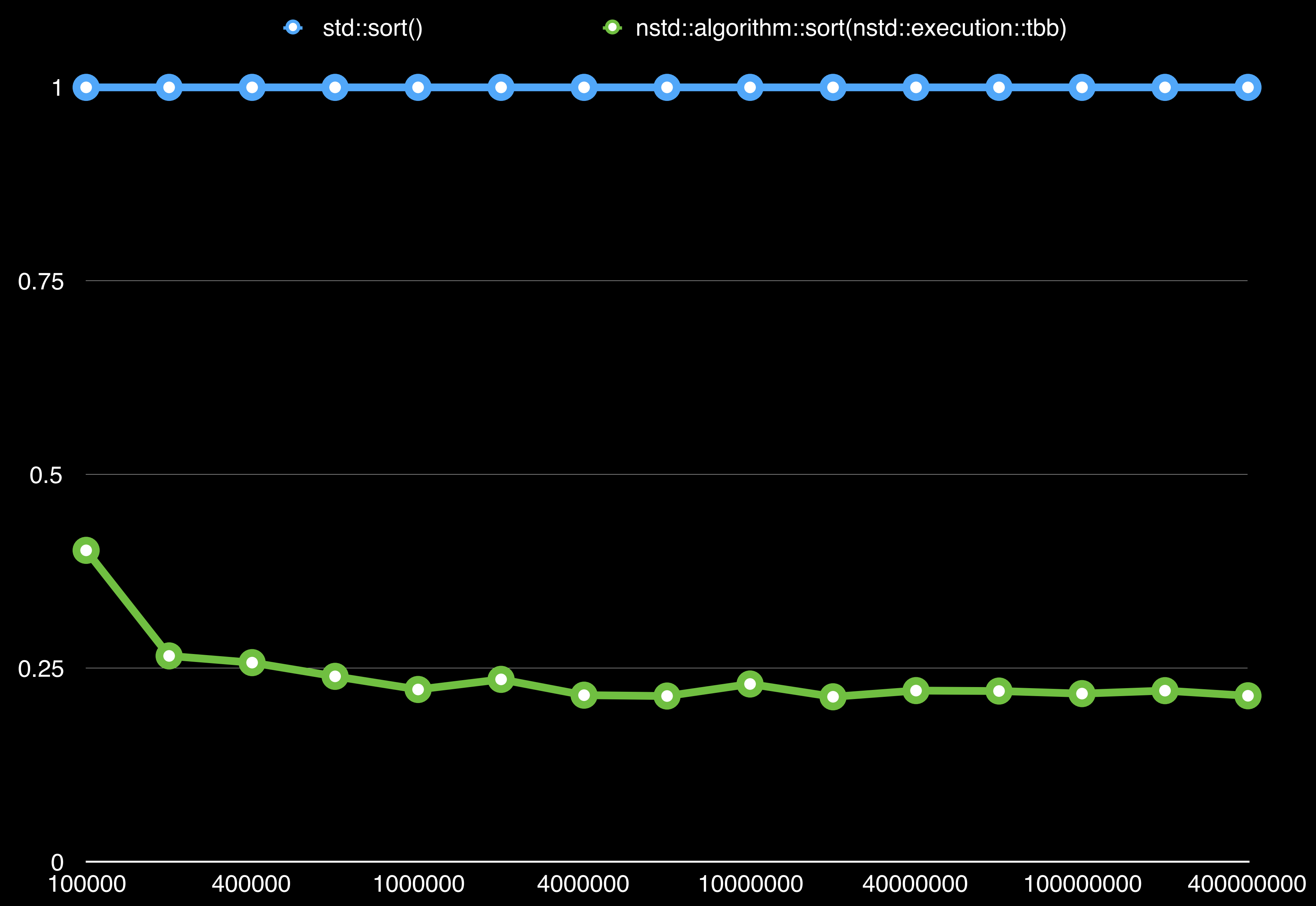
Results: sort gcc phi



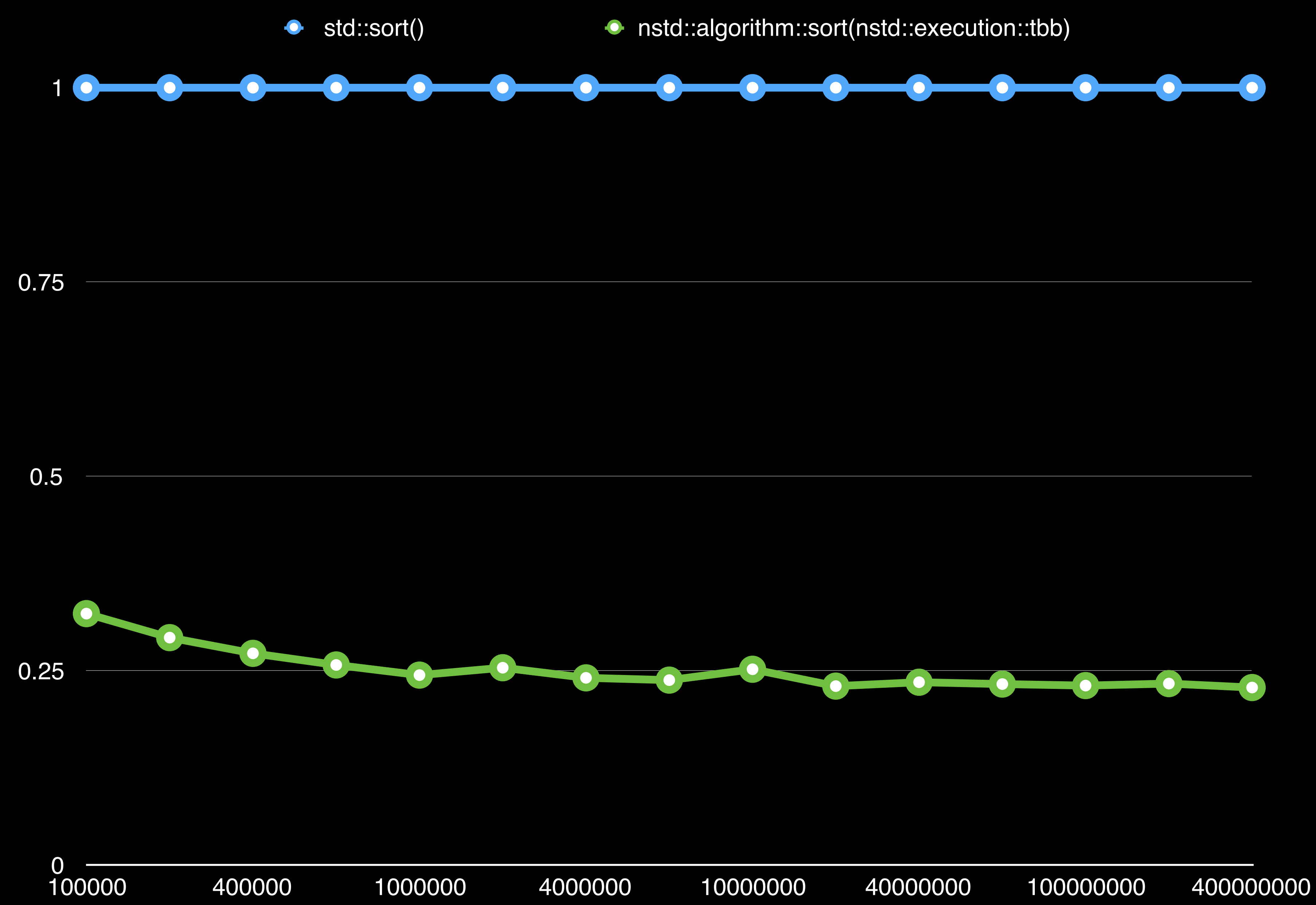
Results: sort clang phi



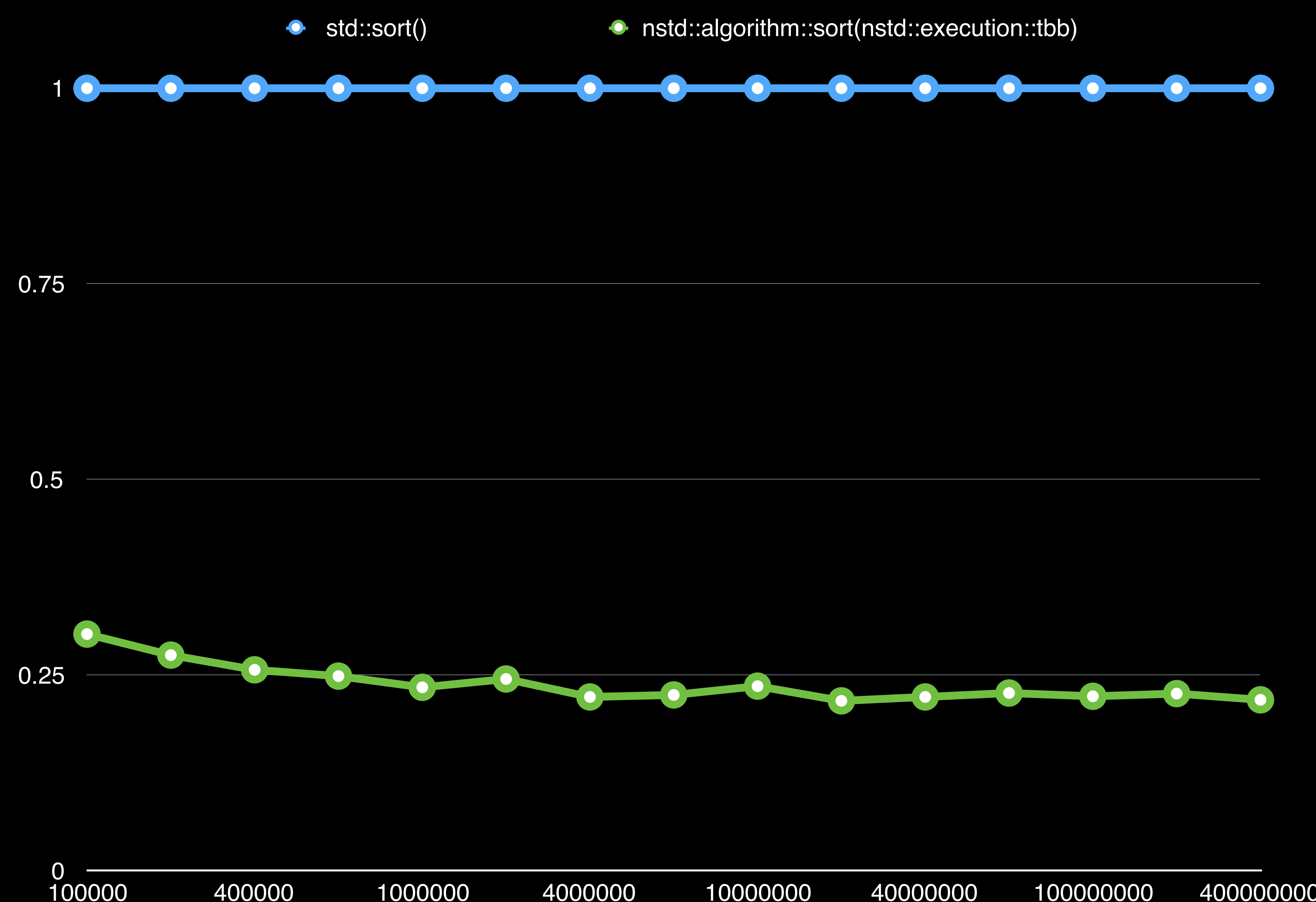
Results: sort Intel I7



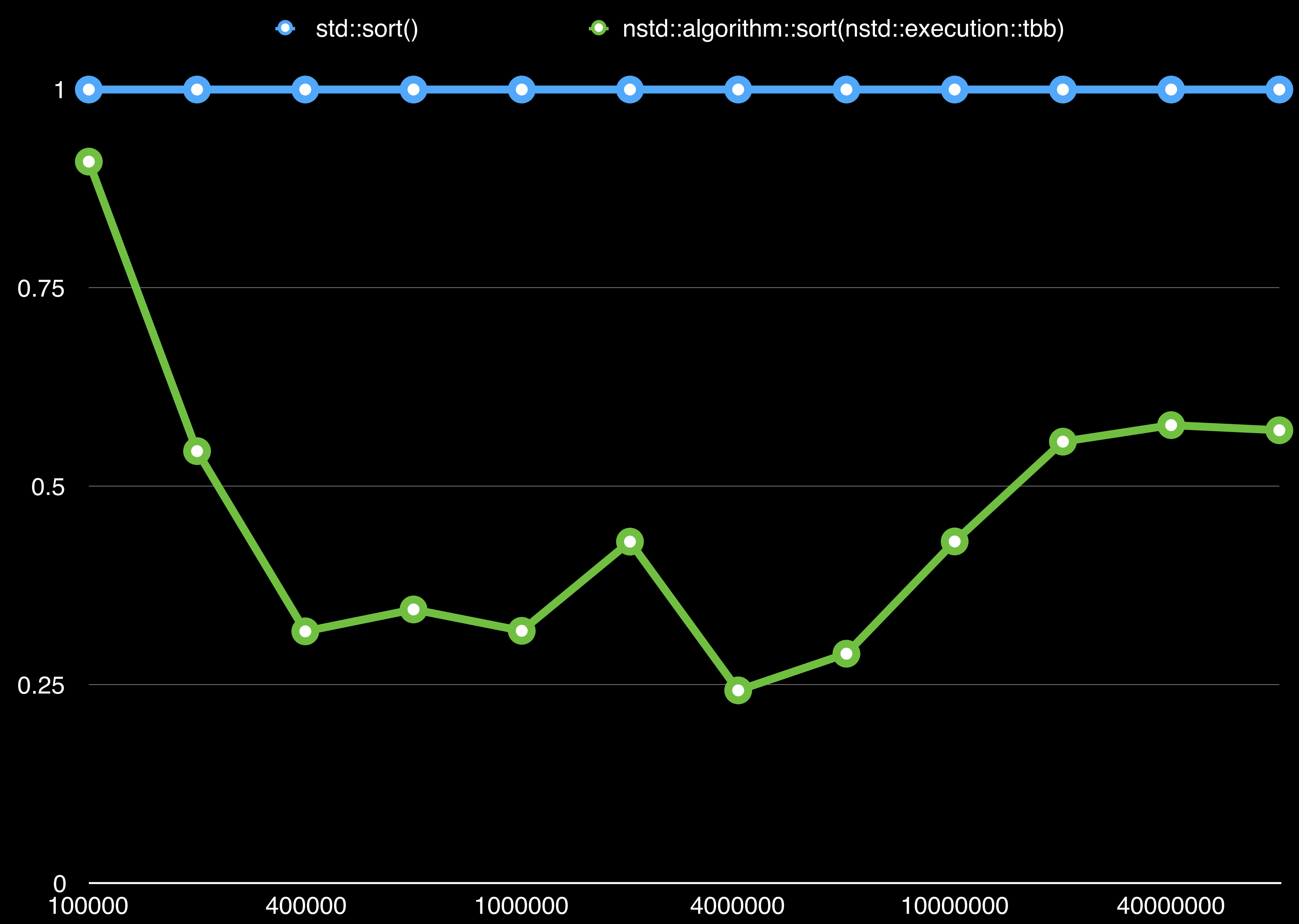
Results: sort gcc I7



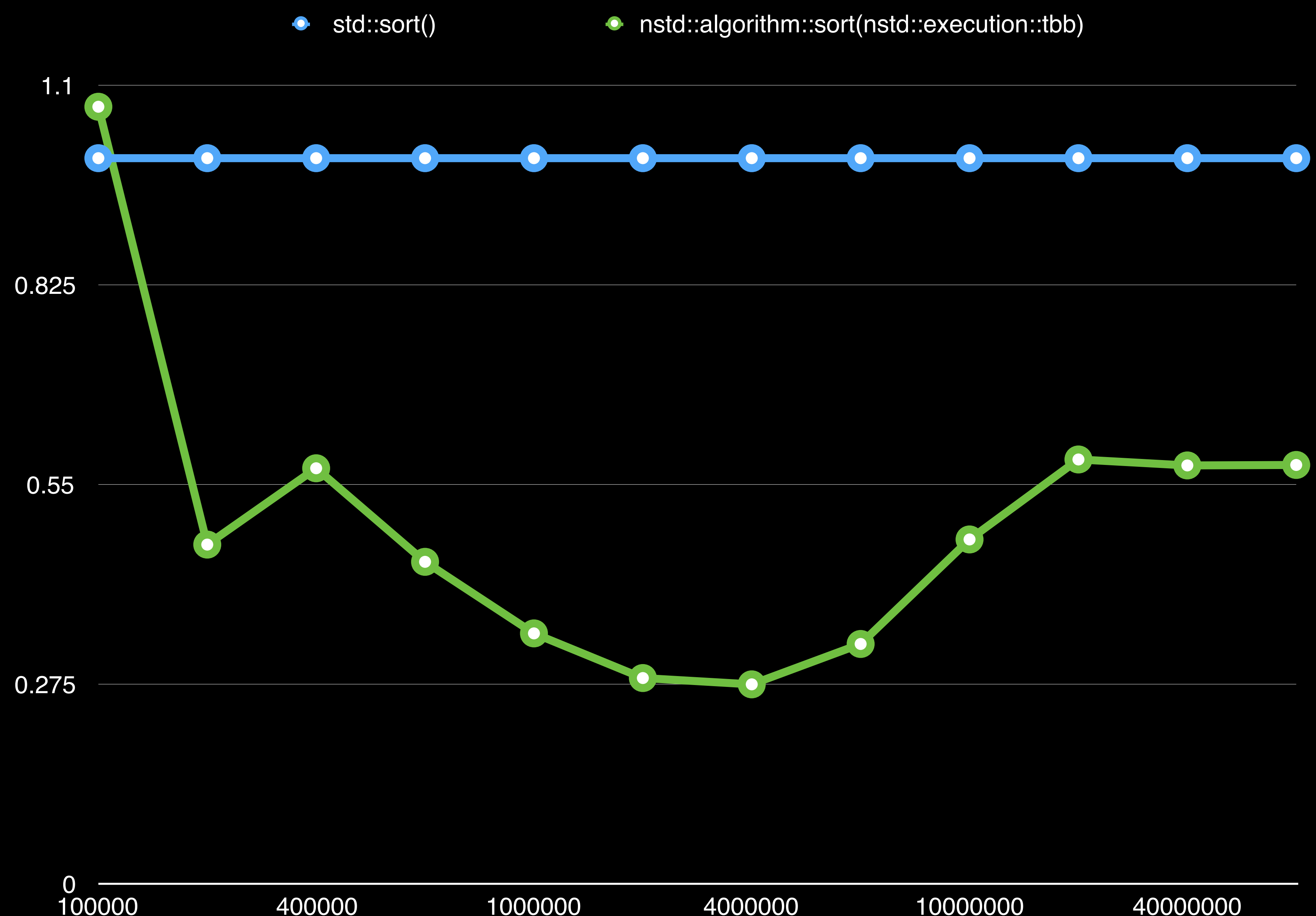
Results: sort clang I7



Results: sort gcc ARM



Results: sort clang ARM



Results: reduce

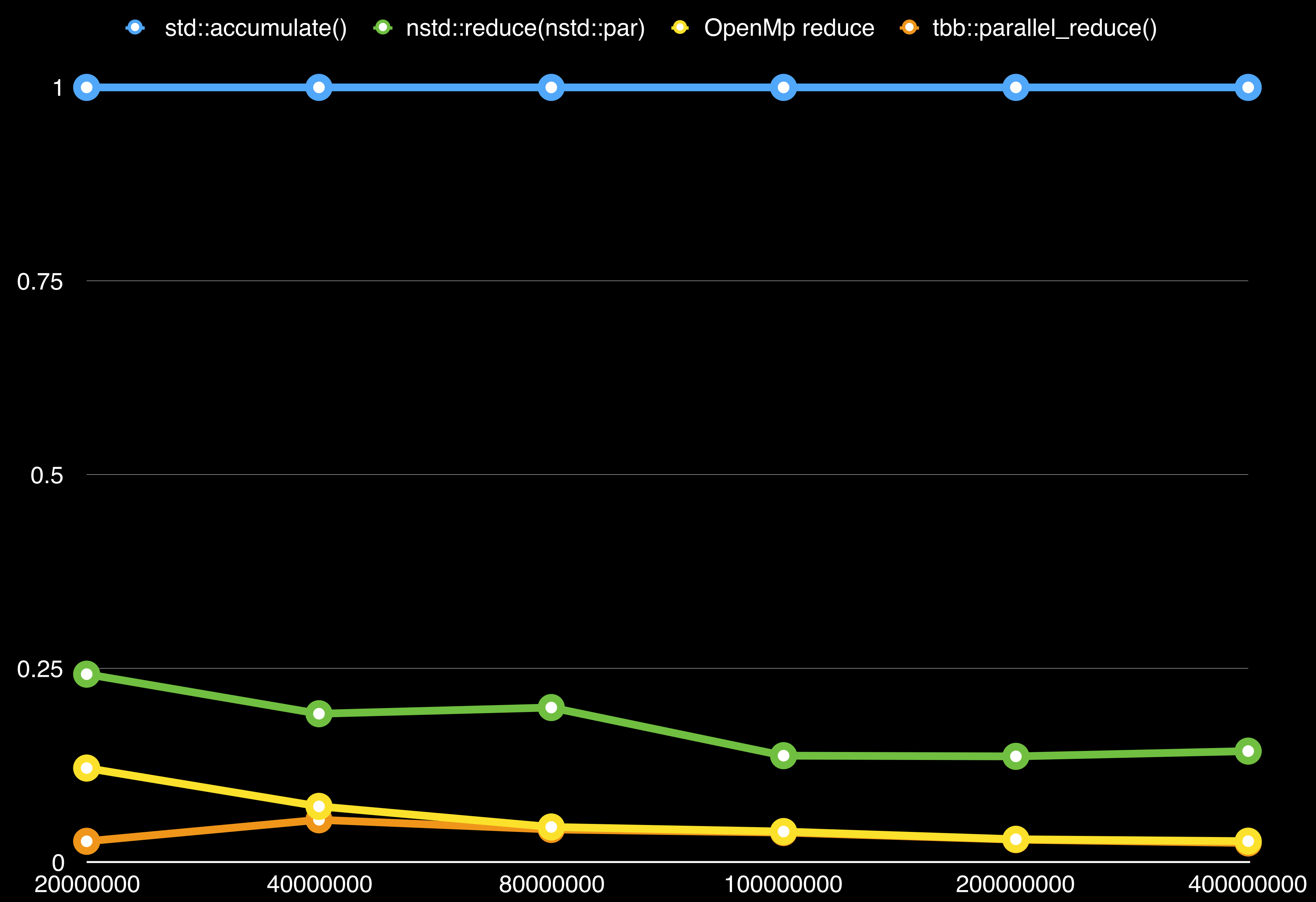
```
accumulate(begin, end);
```

Results: reduce

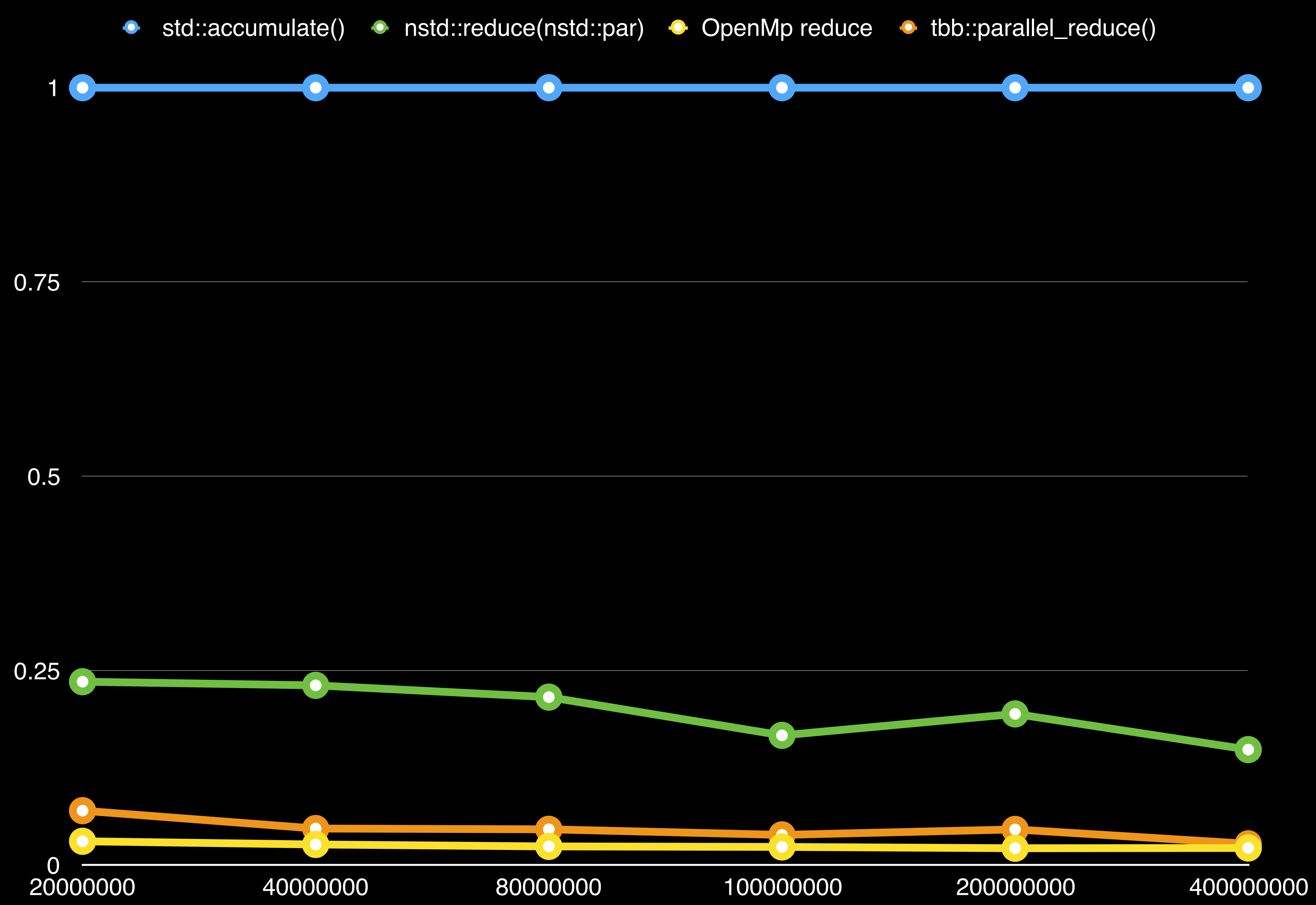
```
reduce(begin, end);
```

Results: redu Intel phi

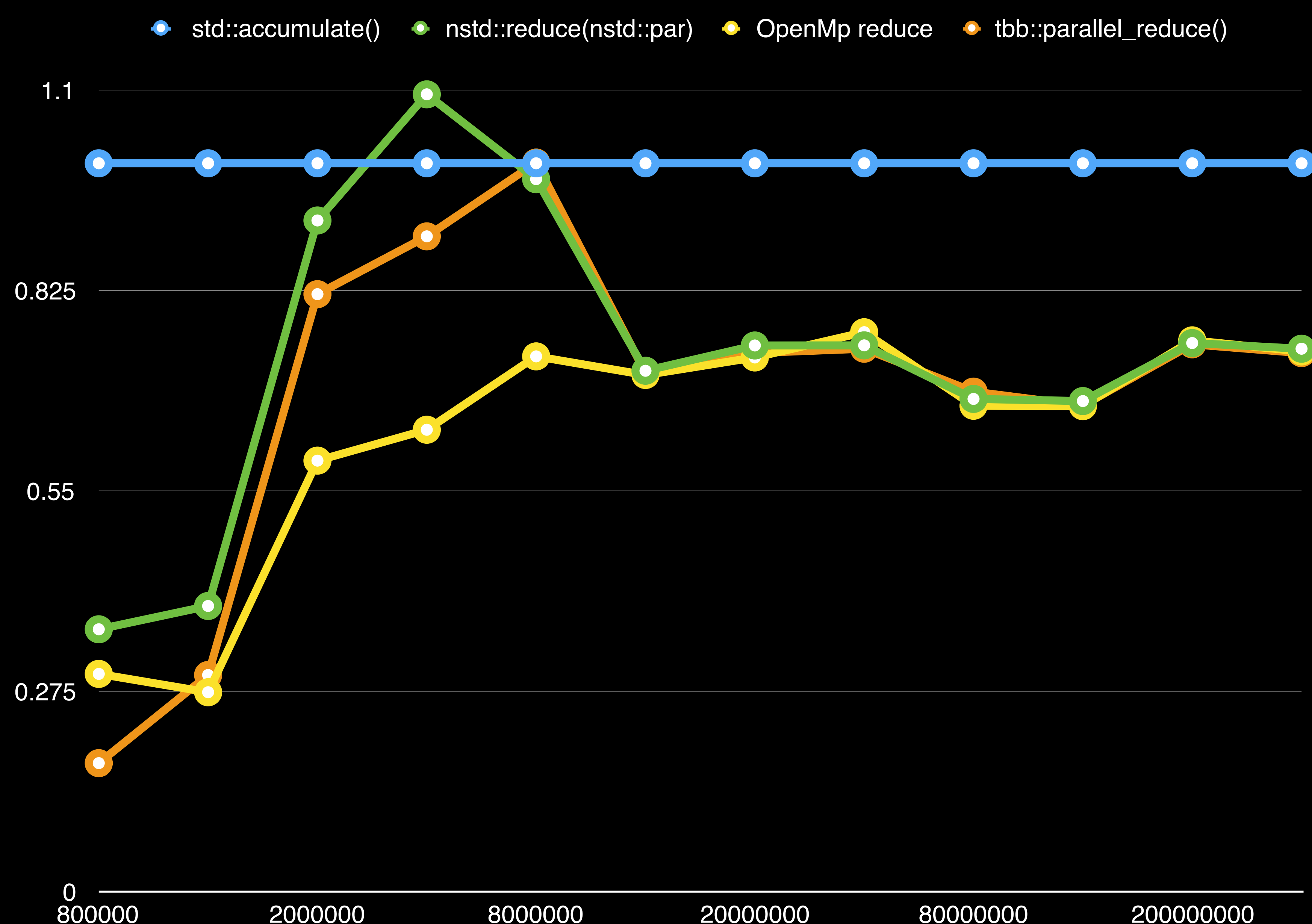
Results: redu gcc phi



Results: redu clang phi



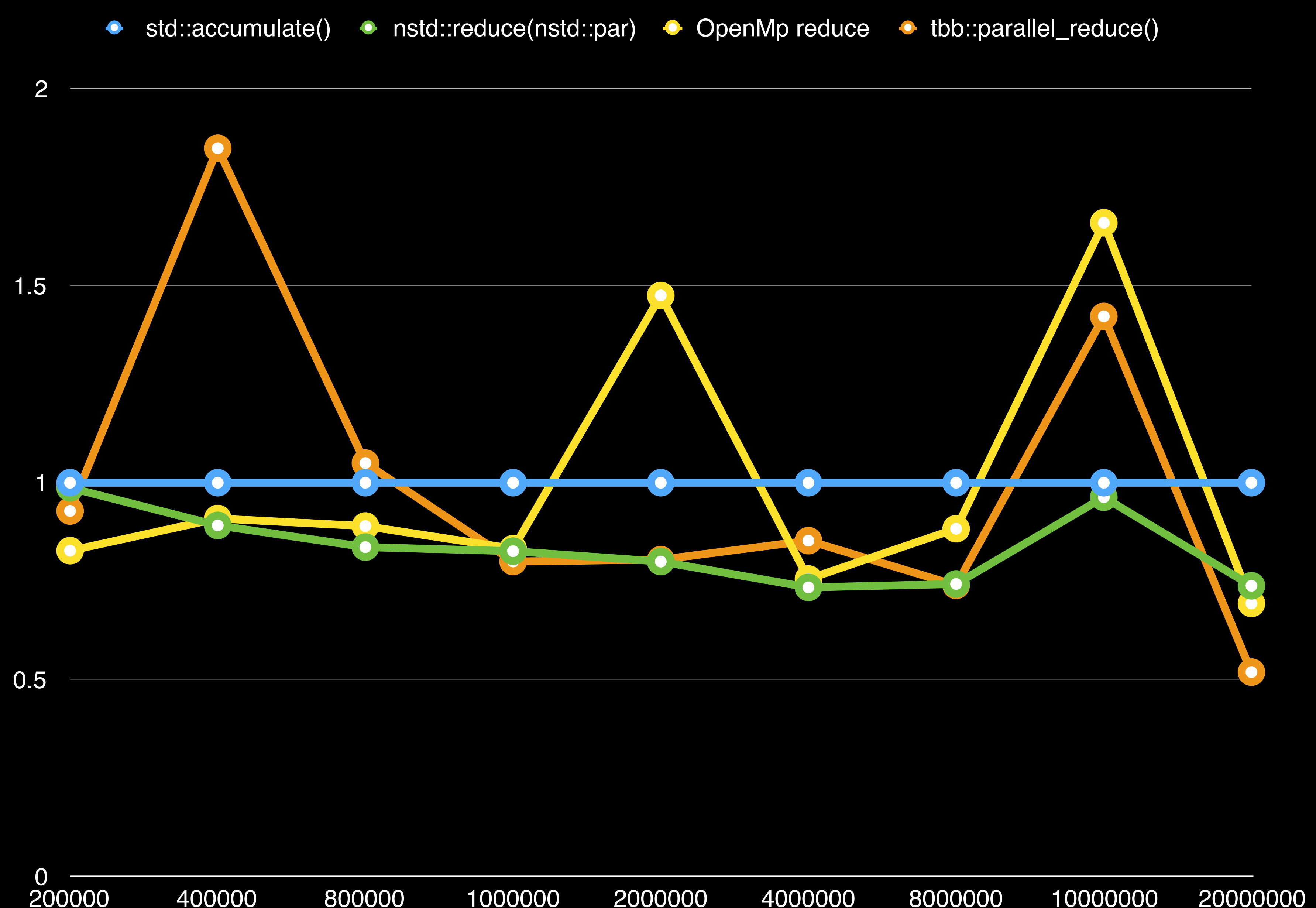
Results: redu Intel I7



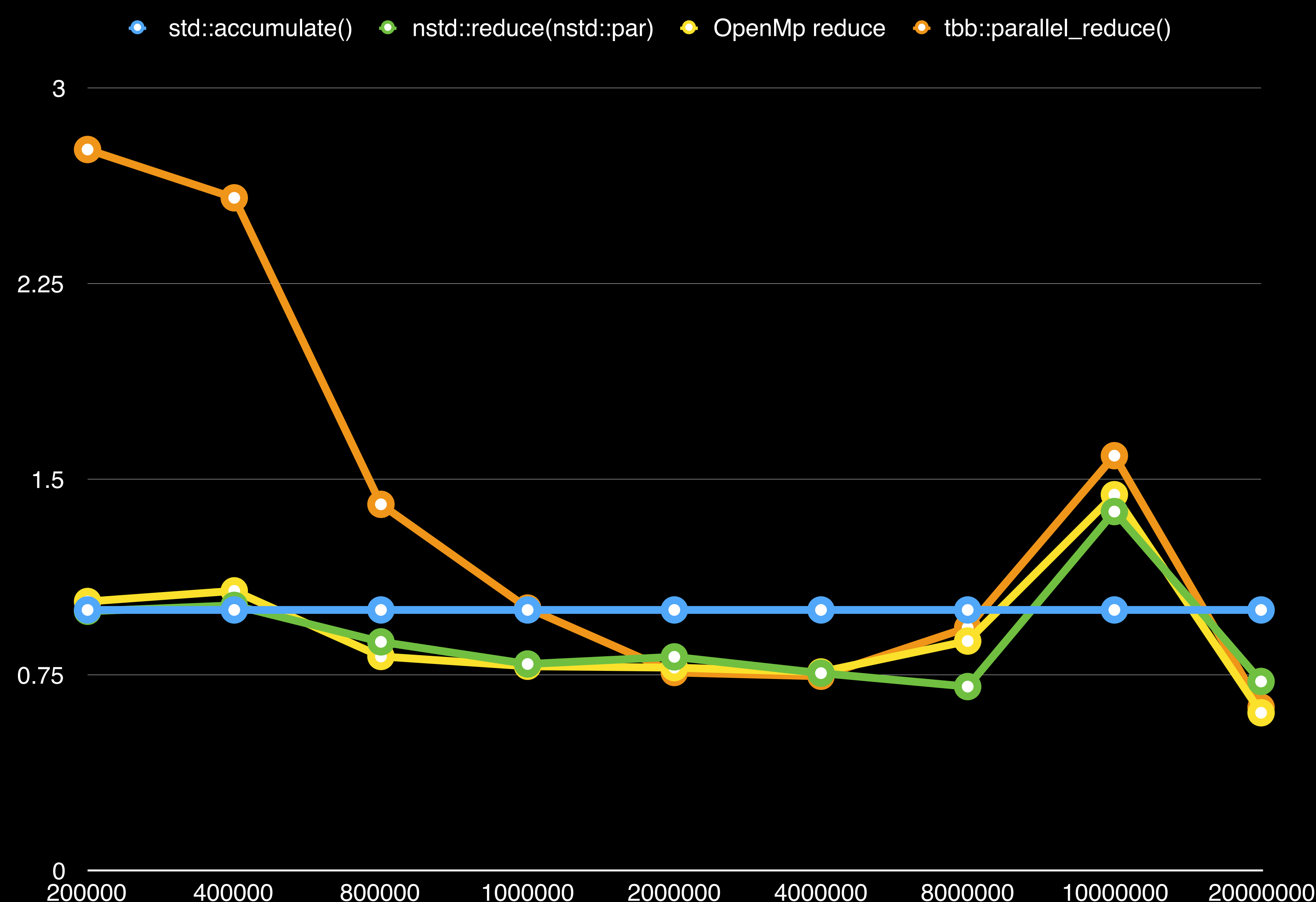
Results: redu gcc I7

Results: redu clang I7

Results: redu gcc ARM



Results: redu clang ARM



Future Directions

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

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

- only a subset of algorithms is implemented
- typically no support for non-random access
- no support for std::par_unseq
- implementations don’t implement a fallback

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

Conclusions

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

Questions ?