Generalisation of nth_element to a range of nths

Johan Lundberg

Document #: P2375R1
Date: 2022-01-14
Project: Programming Language C++
Audience: LEWG, SG19
Email: lundberj@gmail.com

Contents

Revision history vs R0

Added clarifications and more background, explanations, references, summarized performance study, applications, more questions/answers. Updated and rebased wording and synopsis to C++23 draft.

1 Introduction

The paper proposes a generalisation of `std::nth_element`, taking a sorted range of iterators instead of a single `nth` iterator, allowing arbitrary partial sorting of any sortable range.

The use and analysis of such algorithms is widespread and mature[Alsuwaiyel2001, Panh2002, lent1996, Shen1997] under the name *multiple selection* or *multiselect*. It is available to Python programmers as `numpy.partition`[NpPart, NPImpl] since 2014.

The single-nth `nth_element` algorithm has been part of the C++ standard library since the beginning[StepLee95], introduced as "... the element in the position pointed to by nth is the element that would be in that position if the whole range were sorted. Also for any iterator i in the range [first, nth) and any iterator j in the range [nth, last) it holds that !(*i > *j) or comp(*i, *j) == false. It is linear on the average."

This proposal extends the usability of `std::nth_element` to multiple `nths`. That is, the previously stated post condition holds for all nth in `nths`. In other words, at each nth the range is arranged as if sorted, and all elements after each nth are no less than the element at that location.

Just as with the current standard single-nth version of `std::nth_element` the purpose is *faster operation*, but just as with the single-nth version, there is additional *semantic clarity* in performing only the required partitioning. In this specific sense it's in the same category as `std::partial_sort`.

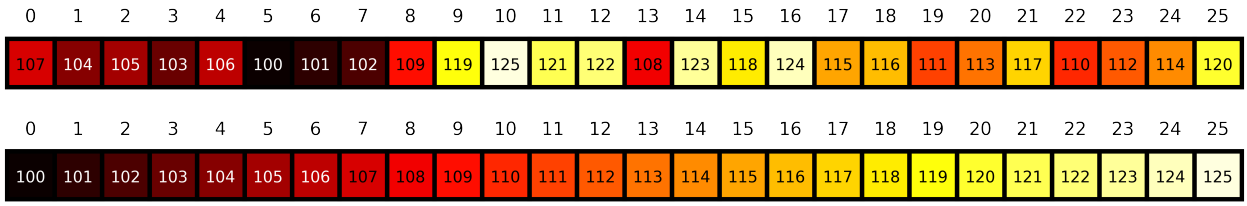
Possible implementations of the range-of-nths algorithm is provided (section 2, and [p2375RefImpl]). It translates naturally to `std::ranges` versions.

Current alternatives are either at least somewhat hard to write correctly and/or less performant.

To clarify what is new, the addition is here called `std::multi_nth_element`, but the proposed wording overloads `std::nth_element`.

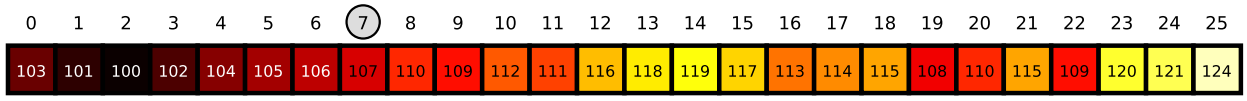
1.1 Visual explanation

As example data, consider the permuted integers [100,126) at index [0,26) and the sorted counterpart:



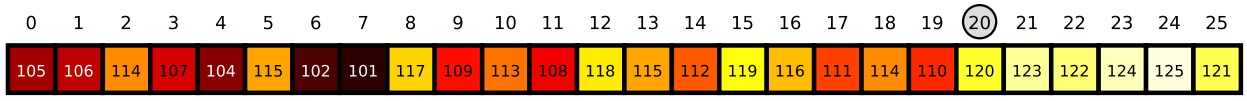
Single-nth `nth_element`

What the current-standard `std::nth_element` does is to rearrange the data in relation to a specified nth position, as described in the previous section. With our concrete example, with nth = begin+7, the effect is that the element at nth is the element (in our case: 107) that would be in that position if the whole range were sorted, and all subsequent values are no less than that value:



,

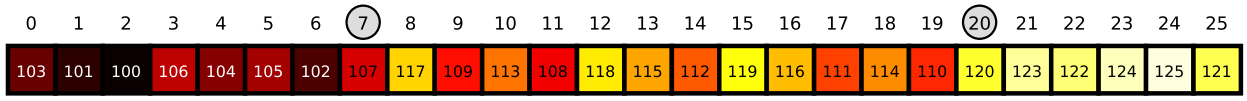
or for `nth = begin+20`:



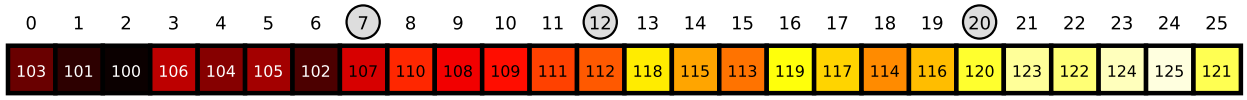
.

Multi-nth `nth_element`

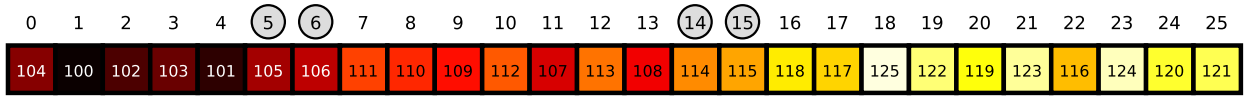
This proposal adds support for multiple selection. That is, the possibility to provide a range of nths, such as `{begin+7, begin+20}` :



or at `{begin+7, begin+12, begin+20}`:



or at `{begin+5, begin+6, begin+14, begin+15}`:



2 Algorithm Implementation

Several implementations are provided[p2375RefImpl]. The first is `multi_quick_select`, which can either be described as the natural extension of quick select, or as a shallow quick sort. The second implementation, here called `bisect_nths`[1] is in a sense simpler, and is shown below. It works by bisecting `nths`, partitions the data using the single-nth `nth_element`, and recurses.

```
// multi_nth_element implemented using bisect_nths
template< class RandomAccessIterator, class RandomAccessIteratorNths >
constexpr void multi_nth_element(RandomAccessIterator first,
  RandomAccessIteratorNths nth_first, RandomAccessIteratorNths nth_last,
  RandomAccessIterator last){
  if (last - first <= 32) { std::sort(first, last); return; }
  const auto nth_dist = nth_last - nth_first;
  if (nth_dist == 0 || *nth_first == last) return;
  const auto nth_mid = nth_first + nth_dist / 2;
  const auto at_nth_mid = *nth_mid;
  nth_element(first, at_nth_mid, last);
  multi_nth_element(first, nth_first, nth_mid, at_nth_mid);
  if (at_nth_mid != last){
    const auto nth_left = std::upper_bound(nth_mid, nth_last, at_nth_mid );
    multi_nth_element(at_nth_mid + 1, nth_left, nth_last, last);
  }
}
```

The complexity for both `multi_quick_select` and `bisect_nths` is O(N log m) on average, where N = last - first and m is the number of unique elements in nths. To be compared with O(N) on average for current `std::nth_element`.

In many applications m is constant and the complexity as function of N alone is naturally linear on average. On the other hand, in the worst case m varies with N as m = N, and the whole container is sorted. For parallel versions (overloads taking an ExecutionPolicy) it is reasonable to leave freedom to implementers to do a full parallel sort and allow O(N log N).

Python has numpy.partition[NpPart] as their incarnation of `ntn_element`. It also support multi selection, and the implementation[NPImpl](in C++) uses Introselect[Musser1997] by specification[2].

Further references on the subject are [Kaligosi2006, Panh2002].

[1] Analyzed in [Alsuwaiyel2001], in context of parallel versions. Builds on refs [Akl1984, Akl1989, Shen1997].

[2] Interestingly, Musser only hints on single element selection as a future extension of introsort. Further: numpy.partition do not state complexity in terms of M (the size of nths) or m (the number of unique nths), but promises *worst case* O(N). It appears to be $\sim$ N log M for reasonable M, to become $\sim$ N $\cdot$ M for large M, such as M>1e4, N=1e6.

3 Tony Tables (Before/After)

Existing alternatives are to sort the whole container or to figure out a series of calls to e.g. `nth_element` and `partial_sort`. The examples below could be the linear time partitioning of messages to be processed into fixed sized priority buckets, keeping or dropping remaining messages. Or finding the fastest 25, 100, and 1000 race participants in linear time. The partitions themselves form half open ranges so it's easy to e.g. sort and print the 100th up to the 1000th fastest runners by name.

Context: partitioning into a fixed number of slots

```
vector<decltype(v)::iterator> nths;
for(size_t slot=1; slot<16 ; ++slot){
        nths.push_back(v.begin()+ min(slot*2048,N));
}
```

or at some other arbitrary iterators in the inclusive range [first,last].

```
auto nths=vector{v.begin()+25,v.begin()+100,v.begin()+1000};
```

After Simple and O(N)

For all examples, the Tony Tables–*After* case is the same:

⋆⋆ **After** ⋆⋆

`multi_nth_element(v, nths, pred);`
 	
or, for the examples using a projection:
`multi_nth_element(v, nths, pred, proj);`

Alternative 1a: Hand-wired bisection for nths of known size 3. O(N) but messy

Before

`nth_element(v.begin(), nths[1], v.end(), pred);`
`nth_element(v.begin(), nths[0], nths[1], pred);`
`nth_element(nths[1]+1, nths[2], v.end(), pred);`

Did we get this right? Is it correct for repeated nths or empty v? This is an attempt at manually figuring out and hand-inlining *this* proposal for a know nths size equal to 3.

If we are using a projection, we must use sub-ranges or something to the same effect (and with the same risk of corner-case bugs, eg missed +1 at the right places):

Before

`ranges::nth_element(v, nths[1], pred, proj);`
`ranges::nth_element(sub_range(v.begin(),nths[1]), nths[0], pred, proj);`
`ranges::nth_element(sub_range(nths[1]+1,v.end()), nths[2], pred, proj);`

Alternative 1b: Hand-wired bisection for nths of known size 5. O(N) but messy

Before

```
nth_element(v.begin(), nths[5/2-1], v.end(), pred);
nth_element(v.begin(), nths[(5/2)/2-1], nths[5/2-1], pred);
nth_element(nths[(5/2)/2-1]+1, nths[5/2-1], v.end(), pred);
nth_element(nths[(5/2)-1]+1, nths[(5/2)/2+5/2-1], nths[5-1]), pred);
nth_element(nths[(5/2)/2+5/2-1]+1, nths[5/2-1], v.end(), pred);
```

Did we get this right?

Alternative 1c: Hand-wired for size 3. O(N · M)

Hand-wired simpler alternative. Easier to figure out, but O(N · M):

Before

```
nth_element(v.begin(), nths[0],v.end(), pred);
nth_element(nths[0]+1, nths[1], v.end(), pred);
nth_element(nths[1]+1, nths[2], v.end(), pred);
```

Did we get this right?

Alternative 2: Simple but O(N log N)

Before

```
sort(v,pred);
```

4 Examples and Applications

It partitions into any number of partitions as shown in the previous section. Further examples and applications follow.

4.1 `multi_nth_element` and sort

Partitioning a bunch of ponies into several age groups, then sort one group by name.

```
struct Pony{
  double littleness;
  chrono::duration age;
  string name;
};
auto end=multi_nth_element(v, nths, std::greater{}, Pony::age);
std::sort(nths[3], nths[4], std::less{}, Pony::name);
```

4.2 `multi_nth_element` into slots

Context: partitioning into a fixed number of slots

```
vector<decltype(v)::iterator> nths;
for(size_t slot=1; slot<16 ; ++slot){
        nths.push_back(v.begin()+ min(slot*2048,N));
}
```

or at some other arbitrary iterators in the inclusive range [first,last].

```
auto nths=vector{v.begin()+25,v.begin()+100,v.begin()+1000};
```

4.3 Outlier filtering

With two partitioning points, the lowest a, and highest b elements are excluded from a range in constant time with a single call to the proposed extension.

4.4 Pagination and sorted subset

A small sorted windows into a large data set can be selected as if sorted by partitioning at two points. For example, if `j` items fit on a display page, we can jump to page `k`, that is, into the range from `a=v.begin()+j*k` to `b=v.begin()+j*(k+1)`:

```
ranges::multi_nth_element(v, vector{a,b});
processPage(a,b); // May also now continue and sort the small subrange with std::sort(a,b);
```

An option is to pre-partition into all pages, exactly as the *slots* example above.

4.5 Partitioning with interpolation. Quantiles, Percentiles.

`multi_nth_element` can be used to efficiently *implement* the calculation of a single or a range of quantiles.

The current standard single-nth `std::nth_element` is actually not generally enough to calculate even a single quantile point, such as the median in the way that is often preferred: For example the [median](https://en.wikipedia.org/wiki/Median) of an even number of elements is typically taken to be the mean of the two central elements. With `multi_nth_element`, single or multiple quantiles can be calculated efficiently.

It's also a common situation to calculate more than one quantile, such as min, 25%, 50% (median), 75%, max. This requires 5 to 8 partition points depending on the size of the data. With *this* proposal this can be done in O(N).

Also note wikipedia on [Percentiles](https://en.wikipedia.org/wiki/Percentile), and [Estimating quantiles from a sample](https://en.wikipedia.org/wiki/Quantile#Estimating_quantiles_from_a_sample).

To do interpolation around each requested quantile (such as the median of even N or a percentile that does not divide N) one may directly partition at two iterators at each requested quantile point. For example, partitioning N elements at a single quantile specified as a divisor d (where d=2 would be median and d=100 would mark the first percentile).

```
auto n = N == 0?0:N-1;
auto [q,r] = div(n, d);
auto nths=vector{first+q, first+q+(r>0)};

auto last = multi_nth_element(v, nths, std::less{}, Pony::littleness);
if (nths[0]!=last){
  cout << nths[0]->name << " " << nths[1]->name ;
  auto intrp_littleness = lerp(nths[0]->littleness, nths[1]->littleness, r*1.0/d);
}
```

In the above we did floating point based interpolation, but one may stay in integer arithmetic[3] for example when working with chrono durations, iterators and indices. Any type the user knows how to interpolate.

```
auto last = multi_nth_element(v, nths, std::less{}, Pony::age);
if (nths[0]!=last){
  auto intrp_dur = i_lerp(nths[0]->age, nths[1]->age, r, d);
}
```

In Python, `numpy.quantile`[4] takes a range of floating point quantile points in [0.0,1.0] and uses the previously mentioned multi-nth version of `numpy.partition`

[3] `i_lerp(auto a,auto b,auto r,auto d){return a+(r*(b-a))/d;}`. Yes, there are other ways to express this depending on type, e.g. extra work to avoid overflow.

[4] It defaults to the above division by N-1 to do linear interpolation but there's a plethora of variations (nine different modes supported by many statistical libraries and tools) on which indices to use, rounding and interpolation/selection and handling of edge cases. A good overview is found in P2119R0 commenting on the paper P1708R4 "Simple Statistical Functions" which proposes user-facing median and quantile similar to `numpy.partition`, returning by value (not via iterators).

4.6 Histogram equalization and bin selection. Application to image equalization

Image equalization, or data equalization in general is a data processing technique used to enhanced contrast.[5] Equalization makes use of the Cumulative distribution function, CDF[6] of the image (or data in general). The cumulative distribution can be estimated by taking the cumulant of a *histogram* of the data. Finding good bins is itself done by histogram equalization, which itself relies on an estimated CDF.

An interesting alternative to cumulant-of-histogram, is to obtain the CDF from ordering the data directly. If this is done with a full sort, and exact descriptive CDF is obtained, resulting in a loss-less equalization (modulo floating point and image format aspects).[7]

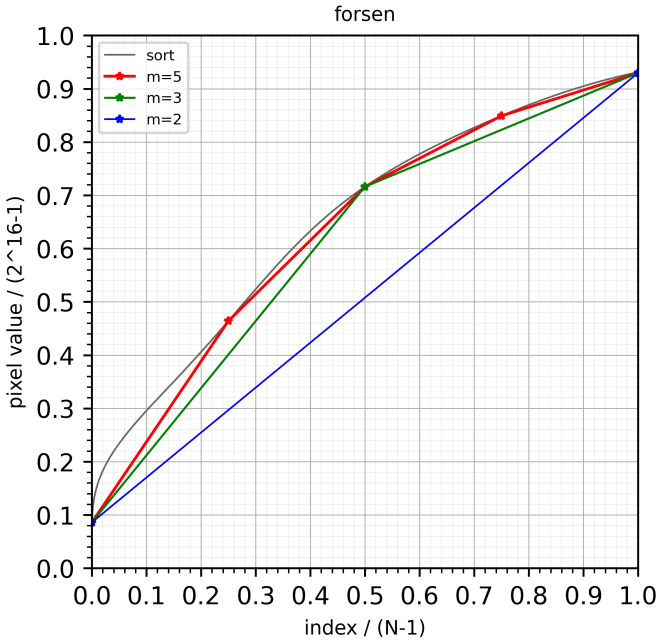
As a performance optimization we can avoid a full sort, and instead use `multi_nth_element` and ensure exact results at specific cumulant values (partition points) in the CDF, such as at 0.25, 0.5, 0.75. That is, we use a multiselection (aka `multi_nth_element)` approximate the full sorted data. Worked through examples are found at [p2375RefImpl]. In summary, consider the following image of Tännforsen (Sweden's greatest waterfall), with three different equalization methods.

<table>
<tr><td>Original:
  </td><td>Equalized using full sort
  </td></tr>
<tr><td>Equalized using multi_nth_element: m=2:
  </td><td>Equalized using multi_nth_element: m=5
  </td></tr>
</table>

[5]The subject is described for example at `https://en.wikipedia.org/wiki/Histogram_equalization` and `https://www.tutorialspoint.com/dip/histogram_equalization.htm`

[6]CDF: `https://en.wikipedia.org/wiki/Cumulative_distribution_function`

[7]A more detailed description of sort-based image equalization: Image enhancement using sorted histogram specification and POCS postprocessing - Il-Lyong Jung and Chang-Su Kim, 2007 IEEE International Conference on Image Processing, ICIP 2007 Proceedings.



Above, the exact CDF (using sort) of the original image is shown in gray, along with approximations using *multi_nth_element*. At m=2, the equalization algorithm is equivalent to mapping the image min,max to grayscale values zero and 1.

5 Wording and Synopsis

5.1 [alg.nth.element]

Underlined green text marks additions with respect to C++23 draft (2021-12-27).

1 Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

2 *Preconditions:* `[first, nth)` and `[nth, last)` are valid ranges. For the overloads in namespace `std`, `RandomAccessIterator` meets the *Cpp17ValueSwappable* requirements ref:*swappable.requirements*, and the type of `*first` meets the *Cpp17MoveConstructible* (ref:*cpp17.moveconstructible*) and *Cpp17MoveAssignable* (ref:*cpp17.moveassignable*) requirements. For the overloads taking a range [nths_first,nths_last), RandomAccessIteratorNths is a *Cpp17RandomAccess* iterator, and *nths_first is convertible to RandomAccessIterator. For every iterator i and j in the range [nths_first,nths_last), it holds that if i<j then !(*j<*i).

3 *Effects:* After `nth_element` the element in the position pointed to by `nth` is the element that would be in that position if the whole range were sorted with respect to `comp` and `proj`, unless `nth == last`. Also for every iterator `i` in the range `[first, nth)` and every iterator `j` in the range `[nth, last)` it holds that: `bool(invoke(comp, invoke(proj, *j), invoke(proj, *i)))` is `false`. For the overloads taking a range of nths, this holds for all nth in nths.

4 *Returns:* `last` for the overload in namespace `ranges`.

5 *Complexity:* For the overloads with no `ExecutionPolicy`, linear on average. For the overloads with an `ExecutionPolicy`, $\mathcal{O}(N)$ applications of the predicate, and $\mathcal{O}(N \log N)$ swaps, where $N =$ `last - first`. For overloads taking a range of nths but no ExecutionPolicy, $\mathcal{O}(N\ log\ m)$ on average, where m is the number of unique elements in nths. For overloads taking a range of nths and an ExecutionPolicy, approximately $\mathcal{O}(N\ log\ N)$.

5.2 Synopsis – `<algorithm>` [algorithm.syn]

Added signatures to `std::`

```
template<class RandomAccessIterator, class RandomAccessIteratorNths>
constexpr void nth_element( RandomAccessIterator first,
RandomAccessIteratorNths nths_first, RandomAccessIteratorNths nths_last,
RandomAccessIterator last);

template<class RandomAccessIterator, class RandomAccessIteratorNths, class Compare>
constexpr void nth_element( RandomAccessIterator first,
RandomAccessIteratorNths nths_first, RandomAccessIteratorNths nths_last,
RandomAccessIterator last, Compare comp);
```

```
template<class ExecutionPolicy, class RandomAccessIterator, class RandomAccessIteratorNths>
void nth_element(ExecutionPolicy&& exec, RandomAccessIterator first,
RandomAccessIteratorNths nths_first, RandomAccessIteratorNths nths_last,
RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator,
class RandomAccessIteratorNths, class Compare>
void nth_element(ExecutionPolicy&& exec, RandomAccessIterator first,
RandomAccessIteratorNths nths_first, RandomAccessIteratorNths nths_last,
RandomAccessIterator last, Compare comp);
```

Added signatures to `std::ranges::`

```
namespace ranges {
  template<random_access_iterator I, sentinel_for<I> S,
  random_access_range Nths, class Comp = ranges::less, class Proj = identity>
  requires sortable<I, Comp, Proj>
  && convertible_to<iter_reference_t<iterator_t<Nths>>, I>
  constexpr I nth_element(I first, Nths&& nths, S last, Comp comp = {}, Proj proj = {});

  template<random_access_range R,
  random_access_range Nths, class Comp = ranges::less, class Proj = identity>
  requires sortable<iterator_t<R>, Comp, Proj>
  && convertible_to<iter_reference_t<iterator_t<Nths>>, iterator_t<R>>
  constexpr borrowed_iterator_t<R>
  nth_element(R&& r, Nths&& nths, Comp comp = {}, Proj proj = {});
}
```

6 Questions and Answers

6.1 Q: What's the best name?

A: I suggest to reuse `nth_element` for discoverability but it could as well be a separate name. The name `std::nth_element` is established since pre-standard STL times (~1994?). In the literature[Alsuwaiyel2001, Kaligosi2006, Panh2002, lent1996, Shen1997], it is known as *select(ion)*, and the proposed algorithm is known as *multiselect(ion)*.

As an alternative to overloading `std::nth_element`, I find it quite reasonable (and not *too* creative) to follow the same pattern, with the name `std::multi_nth_element`.

Numpy overloads the name "`partition`" for both select and multiselect.

6.2 Q: What about corner cases?

Q: What if `nths` or [first,last) is empty? A: `multi_nth_element` does nothing.

Q: What if some elements of `nths` are equal to last. A: As with `nth_element`, not a problem.

Q: What if some elements of `nths` are equal to each other A: By specification, not a problem.

6.3 Q: Is there's a need to require `nths` be `sized_range`?

A: Not sure, I don't think so. The example implementation does not use .size().

6.4 Q: How should the `nths` be provided?

The current-standard single-nth version uses a single iterator `nth` to designate the location in the range. A range of iterators seems to me the most natural way to designate multiple arbitrary locations in a range.

This seem *least-surprise* and offers flexibility as well as natural combination with other operations (such as seen in the examples here and in the proposal). Python uses indices rather than iterators to express locations in lists and arrays, and its incarnation of the proposed algorithm uses range-of-indices (or a single value) to specify the partition point(s)[NpPart].

6.5 Q: Benefits and performance beyond Ordo.

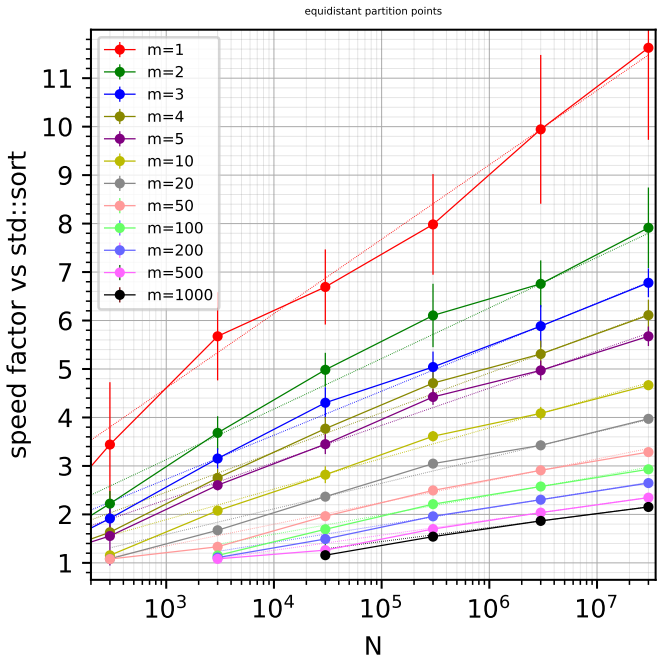
A: The comment is appreciated. Expanded the paper introduction and Examples and Application section. Added performance study, section 7.

An interesting discussion of `std::sort` vs single-nth `std::nth_element` and `std::partial_sort` is found at [CppCon 2018: Fred Tingaud "A Little Order: Delving into the STL sorting algorithms"](https://www.youtube.com/watch?v=-0tO3Eni2uo)

7 Reference implementation and practical performance

Several reference implementations and a performance study, including the code, is found at [p2375RefImpl] and is summarized here.

Time vs `std::sort` for equidistant partitioning points



The above image shows the execution speed of a `multi_quick_select` implementation compared to `std::sort` for various vector sizes N. Each data point is the mean of many repetitions, excluding outliers. Error bars show the spread of all repetitions and is influenced by the randomness of the test data. Individual data points do fluctuate between initial random seeds of the example, but the overall trends are very stable.

The `bisect_nths` algorithm (similar to [Alsuwaiyel2001]) shown in section 2 is about 10% slower for $m > 5$, and about 20% slower for $m > 50$ but has the same asymptotic complexity. The data to sort here consists of randomly shuffled doubles, and `multi_nth_element` was given different numbers m of evenly spaced partitioning points in the vector. Dotted lines show lines $k_i \cdot log(N)$ fitted to pass though each curve at $N = 3e6$.

The image shows that the speed curves approximately follow the expected $log(N)$ shape, with different factors for different m.

The following table shows a few speedup factors for a number of unique partitioning points m and vector sizes N.

m	speedup factor	at N
1	11	$N = 3e7$
5	5.7	$N = 3e7$
10	4.7	$N = 3e7$
500	2.3	$N = 3e7$
1000	2.2	$N = 3e7$
10	3.6	$N = 3e5$
50	2.5	$N = 3e5$
5	2.6	$N = 3e3$

Clearly, the benefit compared to `std::sort` is the greatest for smaller m. In these tests of `multi_quick_select`, the performance are still not worse than `std::sort` even for $m \sim N$. The benefit over `sort` grows with N as $logN$. Further slices and ways to plot the same performance data is found at [p2375RefImpl]. It also shows the study repeated for uniformly random (unique) partition points with very similar conclusion, but slightly better performance.

Acknowledgements

Many thanks to undisclosed proofreaders and to Albin Fredriksson and Marco Rubini for helpful discussions. Many thanks to all who gave comments and feedback.

References

[StepLee95] Alexander Stepanov and Meng Lee: The Standard Template Library.
HP Laboratories Technical Report 95-11(R.1), November 14, 1995
http://stepanovpapers.com/STL/DOC.PDF

[Alsuwaiyel2001] Muhammad H. Alsuwaiyel: An optimal parallel algorithm for the multiselection problem. Parallel Computing Volume 27, Issue 6, May 2001, Pages 861-865
https://doi.org/10.1016/S0167-8191(00)00095-8

[Kaligosi2006] Towards Optimal Multiple Selection
Kanela Kaligos,Kurt Mehlhorn, J. Ian Munro, Peter Sanders – July 2005 Lecture Notes in Computer Science 3580:103-114 – DOI:10.1007/11523468_9

[Akl1984] S. G. Akl, Optimal parallel algorithms for computing convex hulls and for sorting, Computing, 33 (1984), 1-11.

[Akl1989] S. G. Akl, The Design and Analysis of Parallel Algorithms (PrenticeHall, Englewood Cliffs, New Jersey, 1989).

[Shen1997] H. Shen, Optimal parallel multiselection on EREW PRAM, Parallel Computing, 23(1997), 1987-1992.

[NpPart] Python numpy.partition
https://numpy.org/doc/stable/reference/generated/numpy.partition.html

[NPImpl] The implementation of partition (multiple and single nth version) is found at https://github.com/numpy/numpy/blob/v1.20.2/numpy/core/src/multiarray/item_selection.c#L1023

[Musser1997] David R. Musser, Introspective Sorting and Selection Algorithms
Software–Practice and Experience, (8): 983-993 (1997))
https://www.cs.rpi.edu/~musser/gp/algorithms.html

[Panh2002] Alois Panholzer – Analysis of multiple quickselect variants
Theoretical Computer Science Volume 302, Issues 1–3, 13 June 2003, Pages 45-91
https://doi.org/10.1016/S0304-3975(02)00729-6

[lent1996] Janice Lent, Hosam M.Mahmoud
Average-case analysis of multiple Quickselect: An algorithm for finding order statistics
Statistics & Probability Letters
Volume 28, Issue 4, August 1996, Pages 299-310 https://doi.org/10.1016/0167-7152(95)00139-5

[p2375RefImpl] Reference Implementation, Performance study and usage examples on *this* proposal.
https://github.com/jmlundberg/nth_element_material

[p2375src] Document source and status page for *this* proposal.
https://github.com/jmlundberg/p2375