Generalisation of nth_element to a range of nths

Johan Lundberg

Document #:	P2375R0
Date:	2021-05-14
Project:	Programming Language C++
Audience:	LEWG
Email:	lundberj@gmail.com

Contents

1 Introduction . . . 1

2 The algorithm . . . 2

3 Before/After . . . 3

4 Applications . . . 4

5 Wording and Synopsis . . . 5

5.1 First wording draft [alg.nth.element] . . . 5

5.2 Synopsis – <algorithm> [algorithm.syn] . . . 5

6 Questions and Answers . . . 6

Acknowledgements . . . 6

References . . . 6

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] and 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."

A possible implementation of the range-of-nths algorithm is provided (section 2). It translates naturally to `std::ranges` versions just like `std::nth_element`, returning `last`.

To clarify what is new, the addition is called `std::nth_elements` throughout the text, but is proposed to go as `nth_element`.

2 The algorithm

The proposal can be implemented with an algorithm that partitions the data on the midpoint of nths using the single-nth `nth_element` and proceeds to recurse into the two partitions. Ref [Alsuwaiyel2001] considers this algorithm and concludes complexity O(N log m) where N = last - first and m is the number of unique elements in nths. The same paper presents a parallel version, building on refs [Akl1984, Akl1989, Shen1997].

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 supports a range-of-nths as proposed here and the implementation[NPImpl] (in C++) uses Introselect[Musser1997] by specification[1].

A possible implementation[2] is shown below. Comparisons and projections can also be fed through in the most natural way.

```
template< class RandomAccessIterator, class RandomAccessIterator2 >
constexpr void nth_elements(RandomAccessIterator first,
  RandomAccessIterator2 nth_first, RandomAccessIterator2 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);
  nth_elements(first, nth_first, nth_mid, at_nth_mid);
  if (at_nth_mid != last){
    const auto nth_left = std::find_if_not(nth_mid, nth_last,
      [at_nth_mid](auto v) {return v == at_nth_mid; });
    nth_elements(at_nth_mid + 1, nth_left, nth_last, last);
  }
}
```

[1] numpy.partition and the in-place version numpy.ndarray.partion do not state complexity in terms of M (the size of nths) or m (the number of unique nths), but 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.

[2] The implementation is not the point of the proposal but it may help explain it. Feedback would be very much appreciated. The algorithm described in [Alsuwaiyel2001] is a little bit simpler since it takes *unique* nths, something that is not reasonable to require here. The algorithms is O(N log m) except for a small N-independent term O(M-m) where M is the size of nths, due to the iterator comparisons and increments in `find_if_not` to skip over doublets in nths. The first if-statement is found in existing single-nth `nth_element` implementations, but here it also prevents some unnecessary bisections of nths.

3 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)

```
nth_elements(v, nths, pred);
```

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

```
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?

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

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

Did we get this right?

Before Alternative 2: Simple but O(N log N):

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

4 Applications

It partitions into any number of partitions as shown in the previous section. Partitioning a bunch of ponies into several age groups, then sort one group by name.

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

Partitioning with interpolation. Quantiles.

`nth_elements` can be used to efficiently *implement* the calculation of a single or a range of quantiles. To do interpolation one may directly partition at two iterators at each 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 = nth_elements(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 = nth_elements(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).

5 Wording and Synopsis

5.1 First wording draft [alg.nth.element]

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 ([swappable.requirements]), and the type of *first meets the Cpp17MoveConstructible (Table 30) and Cpp17MoveAssignable (Table 32) requirements. For the overloads taking a range [nths_first,nths_last), RandomAccessIterator2 is a RandomAccess 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 Preconditions: The elements e of [first, last) are partitioned with respect to the expression bool(invoke(comp, invoke(proj, e), value)).

4 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.

5 Complexity: For the overloads with no ExecutionPolicy, linear on average. For the overloads with an ExecutionPolicy, O(N) applications of the predicate, and O(N log N) swaps, where N = last - first. For overloads taking a range [nths_first,nths_last) but no ExecutionPolicy the complexity is approximately O(N log m) where m is the number of unique elements in [nths_first,nths_last). For overloads taking a range [nths_first,nths_last) and an ExecutionPolicy the complexity is O(N log N).

5.2 Synopsis – <algorithm> [algorithm.syn]

Added signatures:

```
template<class RandomAccessIterator, class RandomAccessIterator2>
constexpr void nth_element(
RandomAccessIterator first,
RandomAccessIterator2 nths_first, RandomAccessIterator2 nths_last,
RandomAccessIterator last);

template<class RandomAccessIterator, class RandomAccessIterator2, class Compare>
constexpr void nth_element(
RandomAccessIterator first,
RandomAccessIterator2 nths_first, RandomAccessIterator2 nths_last,
RandomAccessIterator last, Compare comp);

template<class ExecutionPolicy, class RandomAccessIterator, class RandomAccessIterator2>
void nth_element(ExecutionPolicy&& exec,
RandomAccessIterator first,
RandomAccessIterator2 nths_first, RandomAccessIterator2 nths_last,
RandomAccessIterator last);
```

```
template<class ExecutionPolicy, class RandomAccessIterator,
class RandomAccessIterator2, class Compare>
void nth_element(ExecutionPolicy&& exec,
RandomAccessIterator first,
RandomAccessIterator2 nths_first, RandomAccessIterator2 nths_last,
RandomAccessIterator last, Compare comp);

namespace ranges {
  template<random_access_iterator I, sentinel_for<I> S,
  random_access_range R2, class Comp = ranges::less, class Proj = identity>
  requires sortable<I, Comp, Proj>
  constexpr I nth_element(I first, R2&& nths, Comp comp = {}, Proj proj = {});

  template<random_access_range R,
  random_access_range R2,
  class Comp = ranges::less, class Proj = identity>
  requires sortable<iterator_t<R>, Comp, Proj>
  constexpr safe_iterator_t<R>
  nth_element(R&& r, R2&& nths, Comp comp = {}, Proj proj = {});
}
```

6 Questions and Answers

Q: What's the best name? A: I suggest to reuse `nth_element` for discoverability but could as well be a separate name. Numpy calls both single nth and range-of-nths "`partition`".

Q: What if `nths` or [first,last) is empty? A: `nth_elements` 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.

Acknowledgements

Many thanks to undisclosed proofreaders and to Albin Fredriksson and Marco Rubini for helpful discussions.

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

[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