From Algorithm to Generic, Parallel Code

Dietmar Kühl
Bloomberg L.P.

Bloomberg

Overview

- Introduce inclusive scan
- Explain the algorithm based on a version from a text book
- Implement versions using OpenMP and Threading Building Blocks (TBB)
- Implement a version using an executor

Inclusive Scan

- Compute all of the sums of initial subsequences:
- For all i compute $y_i = \sum_{j=0}^{i} x_j$
- Trivial to do sequentially
- Example:
  - Input:

1	2	3	4	5	6	7	8	9

  - Output:

1	3	6	10	15	21	28	36	45

Inclusive Scan

- Compute all of the sums of initial subsequences:
- For all i compute $y_i = \sum_{j=0}^{i} x_j$
- Trivial to do sequentially
- Example:
  - Input:

1	2	3	4	5	6	7	8	9

  - Output:

Inclusive Scan

- Compute all of the sums of initial subsequences:
- For all i compute $y_i = \sum_{j=0}^{i} x_j$
- Trivial to do sequentially
- Example:
    - Input:

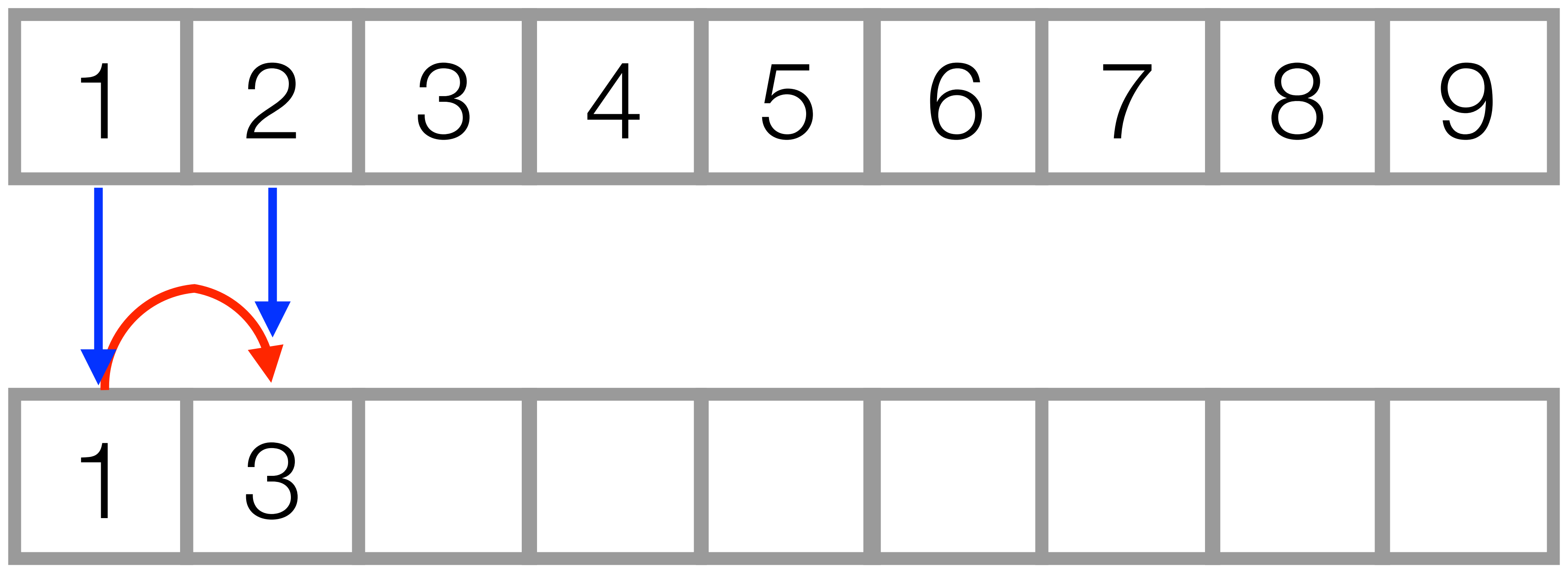
1	2	3	4	5	6	7	8	9

    - Output:

1								

Inclusive Scan

- Compute all of the sums of initial subsequences:
- For all i compute $y_i = \sum_{j=0}^{i} x_j$
- Trivial to do sequentially
- Example:
  - Input:
  - Output:

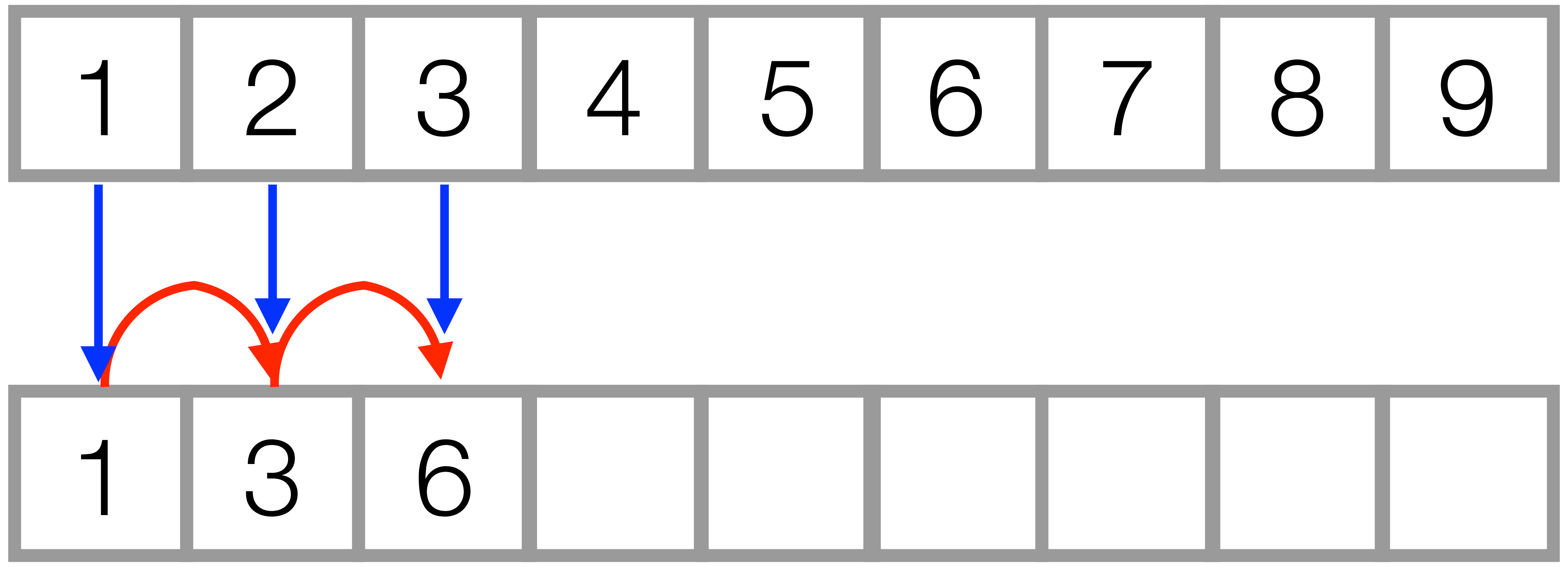


TechAtBloomberg.com

Bloomberg

Inclusive Scan

- Compute all of the sums of initial subsequences:
- For all i compute $y_i = \sum_{j=0}^{i} x_j$
- Trivial to do sequentially
- Example:
  - Input:
  - Output:

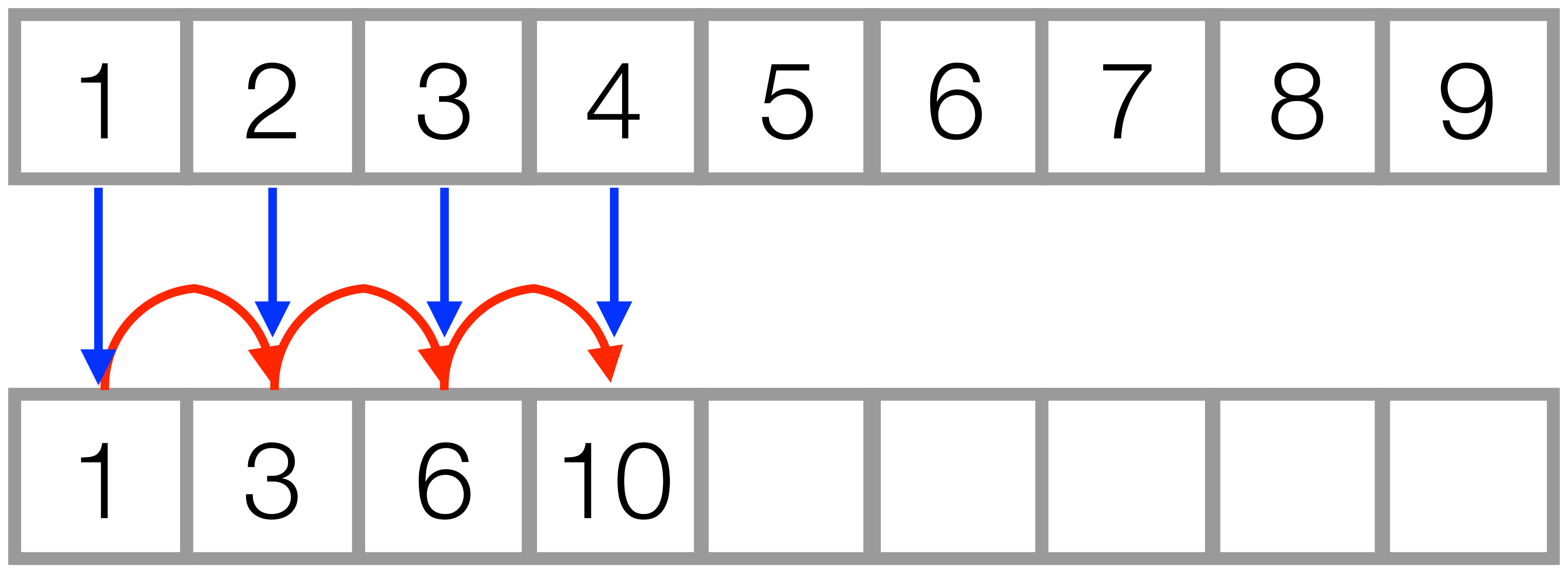


TechAtBloomberg.com

Bloomberg

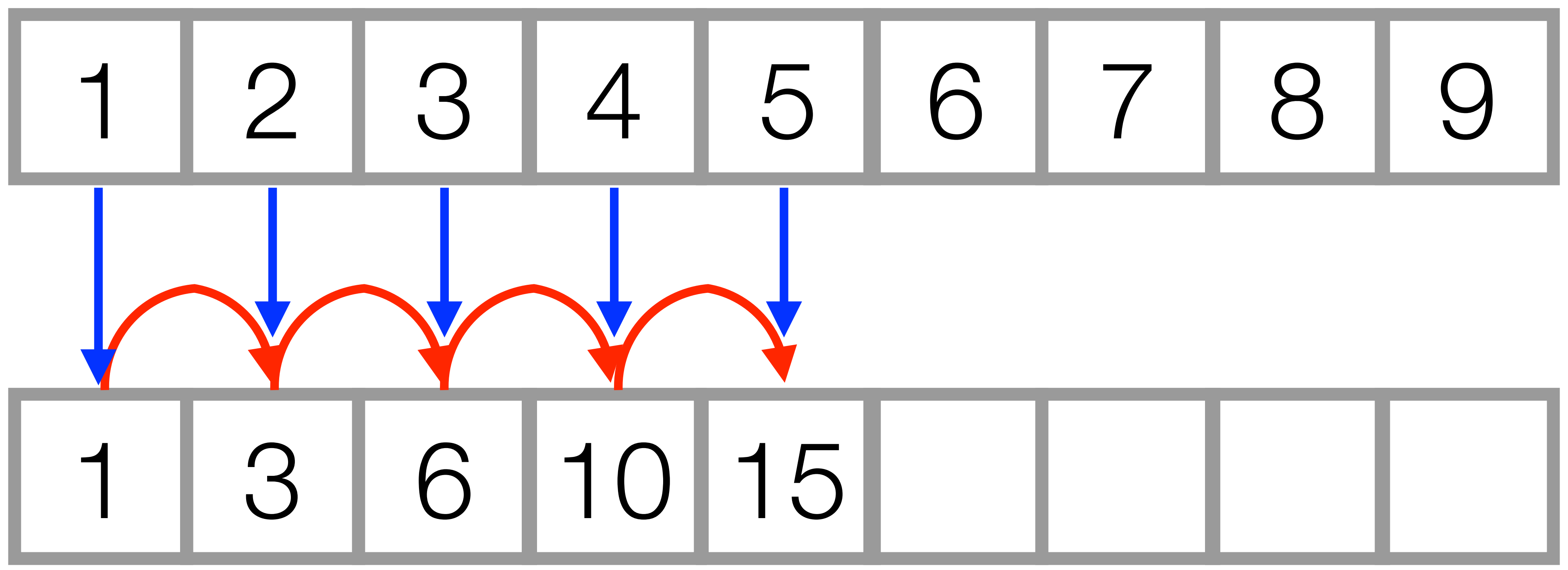
Inclusive Scan

- Compute all of the sums of initial subsequences:
- For all i compute $y_i = \sum_{j=0}^{i} x_j$
- Trivial to do sequentially
- Example:
  - Input:
  - Output:



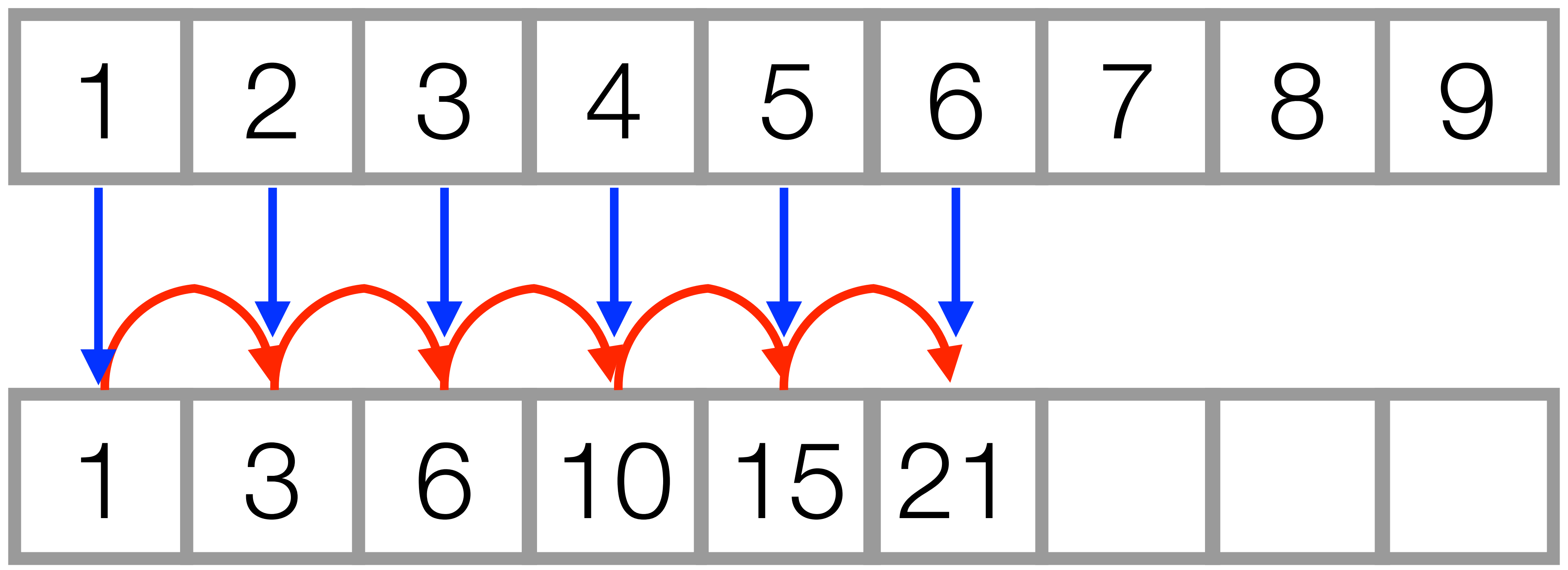
Inclusive Scan

- Compute all of the sums of initial subsequences:
- For all i compute $y_i = \sum_{j=0}^{i} x_j$
- Trivial to do sequentially
- Example:
  - Input:
  - Output:



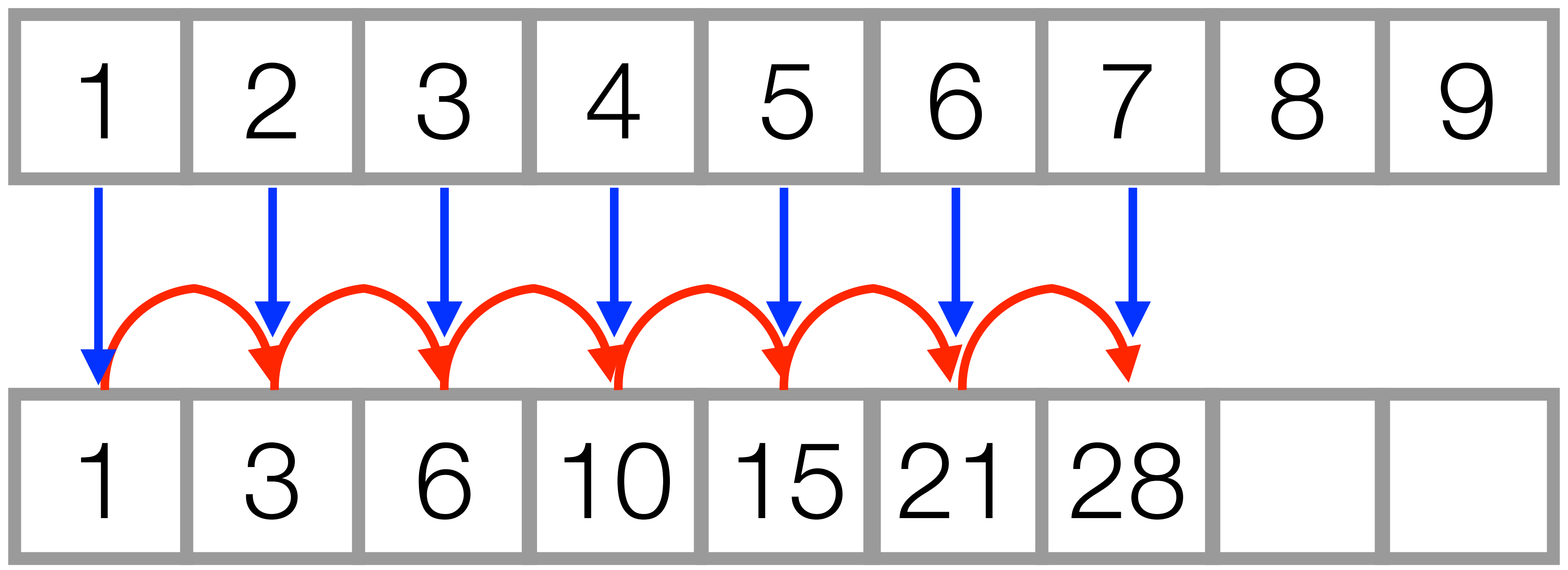
Inclusive Scan

- Compute all of the sums of initial subsequences:
- For all i compute $y_i = \sum_{j=0}^{i} x_j$
- Trivial to do sequentially
- Example:
  - Input:
  - Output:



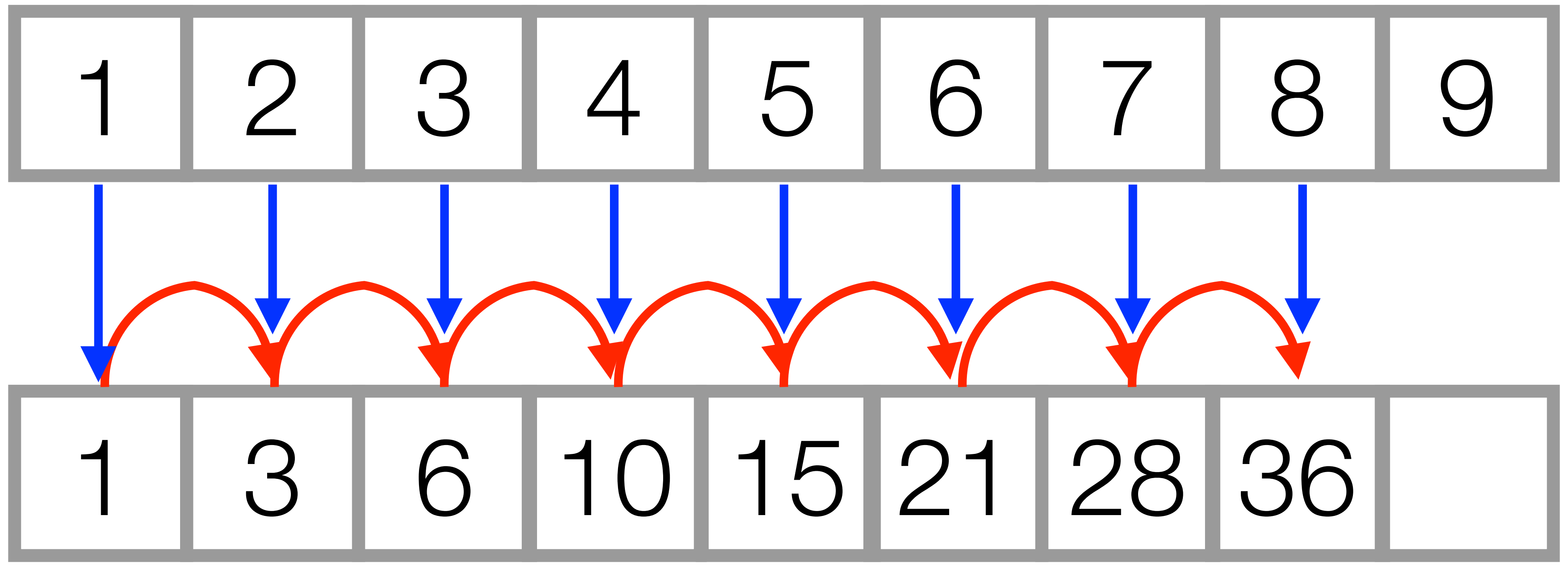
Inclusive Scan

- Compute all of the sums of initial subsequences:
- For all i compute $y_i = \sum_{j=0}^{i} x_j$
- Trivial to do sequentially
- Example:
  - Input:
  - Output:



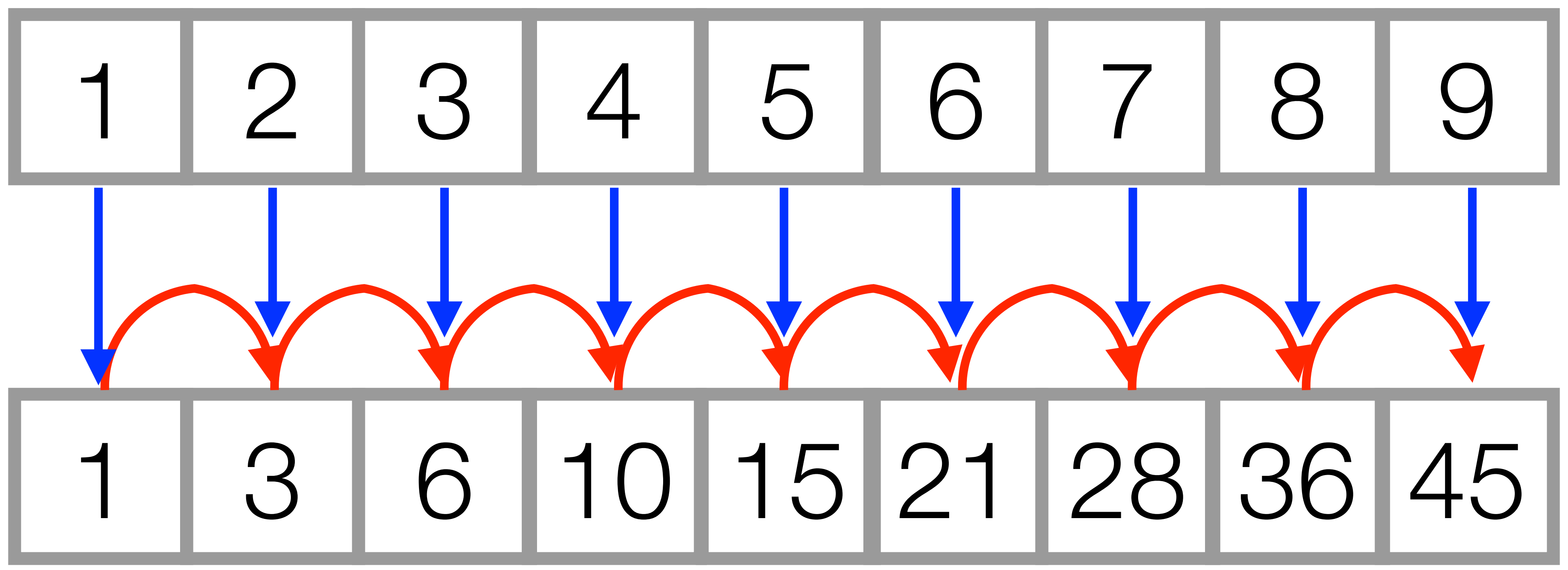
Inclusive Scan

- Compute all of the sums of initial subsequences:
- For all i compute $y_i = \sum_{j=0}^{i} x_j$
- Trivial to do sequentially
- Example:
  - Input:
  - Output:



Inclusive Scan

- Compute all of the sums of initial subsequences:
- For all i compute $y_i = \sum_{j=0}^{i} x_j$
- Trivial to do sequentially
- Example:
  - Input:
  - Output:



Sequential Implementation

```
template <typename In, typename Out, typename Op = std::plus<>>
Out inclusive_scan(In it, In end, Out to, Op op) {
    if (it != end) {
        auto value(*it);
        *to++ = value;
        for (; ++it != end; ++to)
            *to = value = op(value, *it);
    }
    return to;
}
```

Sequential Implementation

```
template <typename In, typename Out, typename Op = std::plus<>>
Out inclusive_scan(In it, In end, Out to, Op op) {
    return it != end? inclusive_scan(it + 1, end, to + 1, op, *to = *it): to;
}

template <typename In, typename Out, typename Op, typename Value>
Out inclusive_scan(In it, In end, Out to, Op op, Value value) {
    for (; it != end; ++it, ++to)
        *to = value = op(value, *it);
    return to;
}
```

Sequential Implementation

```
template <typename In, typename Out, typename Op = std::plus<>>
Out inclusive_scan(In it, In end, Out to, Op op) {
    return it != end? inclusive_scan(it + 1, end, to + 1, op, *to = *it): to;
}

template <typename In, typename Out, typename Op, typename Value>
Out inclusive_scan(In it, In end, Out to, Op op, Value value) {
    for (; it != end; ++it, ++to)
        *to = value = op(value, *it);
    return to;
}
```

Sequential Implementation

```
template <typename In, typename Out, typename Op = std::plus<>>
Out inclusive_scan(In it, In end, Out to, Op op) {
    return it != end? inclusive_scan(it + 1, end, to + 1, op, *to = *it): to;
}

template <typename In, typename Out, typename Op, typename Value>
Out inclusive_scan(In it, In end, Out to, Op op, Value value) {
    for (; it != end; ++it, ++to)
        *to = value = op(value, *it);
    return to;
}
```

Sequential Implementation

```
template <typename In, typename Out, typename Op = std::plus<>>
Out inclusive_scan(In it, In end, Out to, Op op) {
    return it != end? inclusive_scan(it + 1, end, to + 1, op, *to = *it): to;
}

template <typename In, typename Out, typename Op, typename Value>
Out inclusive_scan(In it, In end, Out to, Op op, Value value) {
    for (; it != end; ++it, ++to)
        *to = value = op(value, *it);
    return to;
}
```

TechAtBloomberg.com

Bloomberg

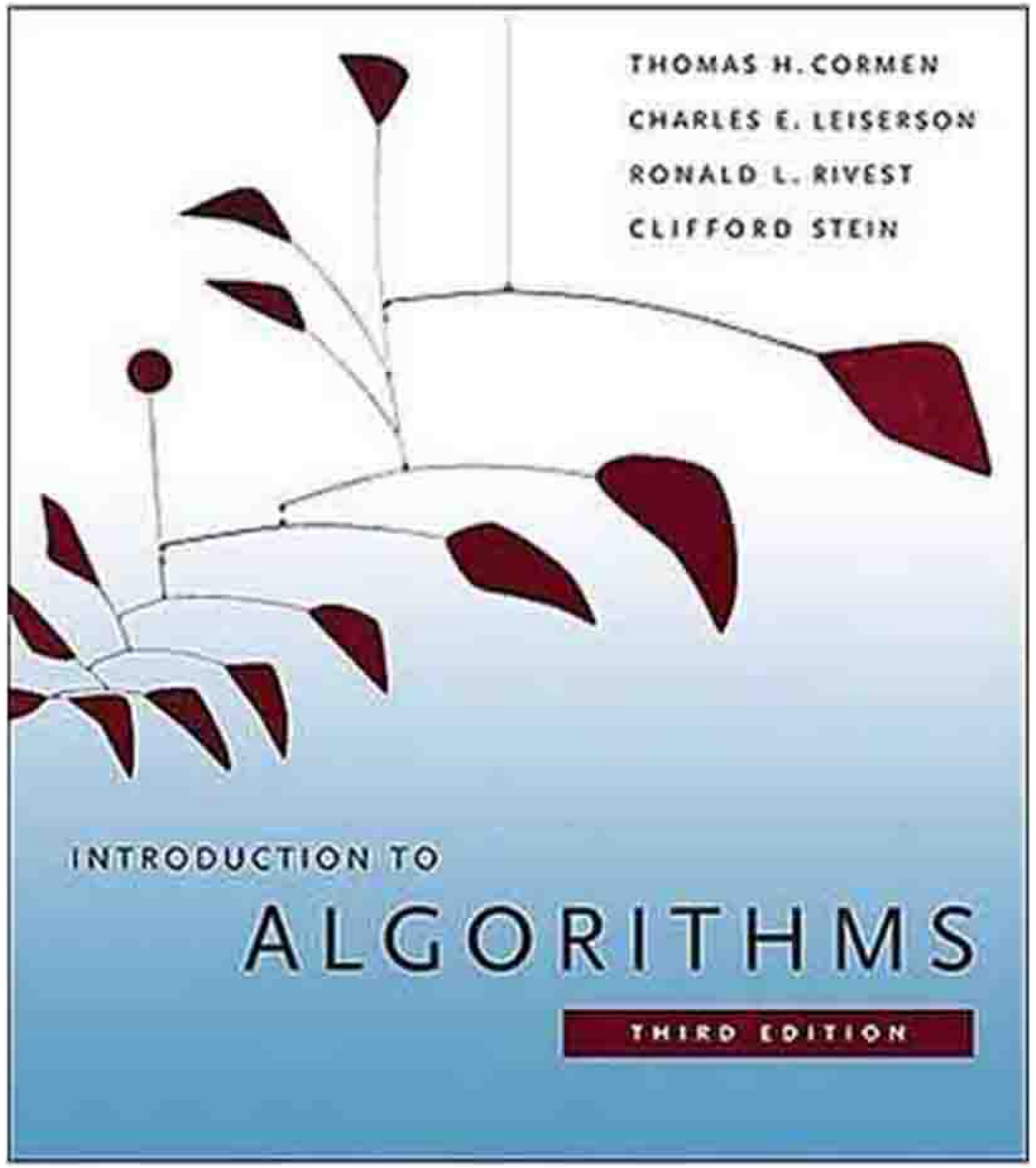
Parallelization

- It doesn't look suitable to parallelize inclusive scan:
  Later values in the sequence depend on all earlier ones
- By doing more work the expected time to completion (span) can be reduced:
  - Recursively decompose the sequence into first/second halves
  - Compute the sums of the halves
  - Compute the partial sums based on the halves
  - Twice as many operations (work) but only log(n) of the time (span)

Basic Idea of the Algorithm

- With the middle sum of each half, these can be populated concurrently
  - Recursively decomposing the populating yields an O(log N) span for this
- Computing the sum of a range can be recursively decomposed:
  - Compute the sum of each half and just add them
  - Doing so actually yields the middle sums needed for populating
  - Recursively decomposing this computation also has an O(log N) span

Parallel Algorithm (from “Introduction to Algorithms”, Cormen et al)



Parallel Algorithm (from "Introduction to Algorithms", Cormen et al)

- P-Scan-Up computes the sum of halves
- P-Scan-Down computes the partial sums based on the halves

```
P-Scan-3(x)
    n = x.length
    let y[1..n] and t[1..n] be new arrays
    y[1] = x[1]
    if n > 1
        P-Scan-Up(x, t, 2, n)
        P-Scan-Down(x[1], x, t, y, 2, n)
    return y
```

Parallel Algorithm: STL interface version

```
template <typename In, typename Out, typename Op>
Out p_scan(In begin, In end, Out to, Op op) {
    auto n = std::distance(begin, end);
    if (0 < n) {
        to[0] = begin[0];
        if (1 < n) {
            std::vector<std::decay_t<decltype(*begin)>> t(n);
            p_scan_up(begin + 1, end, t.begin(), op);
            p_scan_down(begin[0], begin + 1, end, t.begin(), to + 1, op);
        }
    }
    return to + n;
}
```

P-Scan-Up: Compute Auxiliary Sums of Sub-Ranges

```
P-Scan-Up(x, t, i, j)
  if i == j
      return x[i]
  else
      k = ⌊(i + j) / 2⌋

      t[k] = spawn P-Scan-Up(x, t, i, k)
      right = P-Scan-Up(x, t, k + 1, j)
      sync
      return t[k] ⊗ right
```

P-Scan-Up: STL interface version

```
template <typename In, typename Tmp, typename Op>
auto p_scan_up(In b, In end, Tmp tmp, Op op) {
    auto n = std::distance(b, end);
    if (1 == n) { return *b; }
    else {
        auto k = n / 2;
        auto fut = std::async([&]{ tmp[k] = p_scan_up(b, b + k, tmp, op); });
        auto right = p_scan_up(b + k, end, tmp + k, op);
        fut.wait();
        return op(tmp[k], right);
    }
}
```

P-Scan-Down: Compute the Actual Result

```
P-Scan-Down(v, x, t, y, i, j)
    if i == j
        y[i] = v ⊗ x[i]
    else
        k = ⌊(i + j) / 2⌋

        spawn P-Scan-Down(v, x, t, y, i, k)
        P-Scan-Down(v + t[k], x, t, y, k + 1, j)
        sync
```

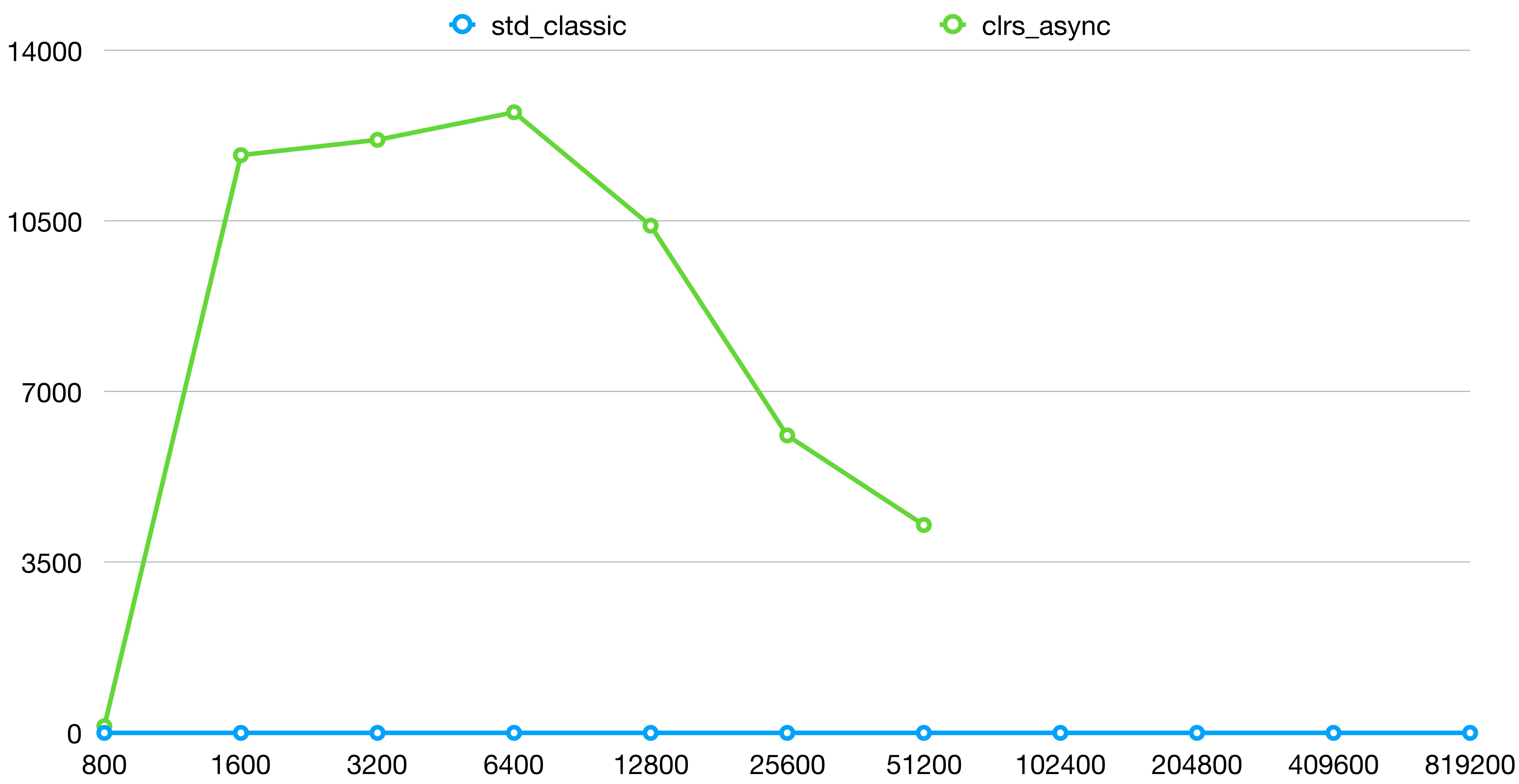
P-Scan-Down: STL interface version

```
template <typename V, typename In, typename T, typename To, typename Op>
void p_scan_down(V v, In b, In end, T tmp, To to, Op op) {
    auto n = std::distance(b, end);
    if (1 == n) { *to = op(v, *b); }
    else {
        auto k = n / 2;
        auto fut = std::async([&]{ p_scan_down(v, b, b + k, tmp, to, op); });
        p_scan_down(op(v, tmp[k]), b + k, end, tmp + k, to + k, op);
        fut.wait();
    }
}
```

The Benchmarks

- All benchmarks are run on a Knights Landing/Xeon Phi machine
  - 64 Intel Atom cores at 1.2GHz, 4 times hyper-threaded
  - 16GB fast access memory, 96GB memory
- The work load is multiplying random generated 4x4 matrices
  - Use the same data a few times to make it not entirely memory bound
- The curves time taken relative to a sequential implementation: **low ⇒ good**

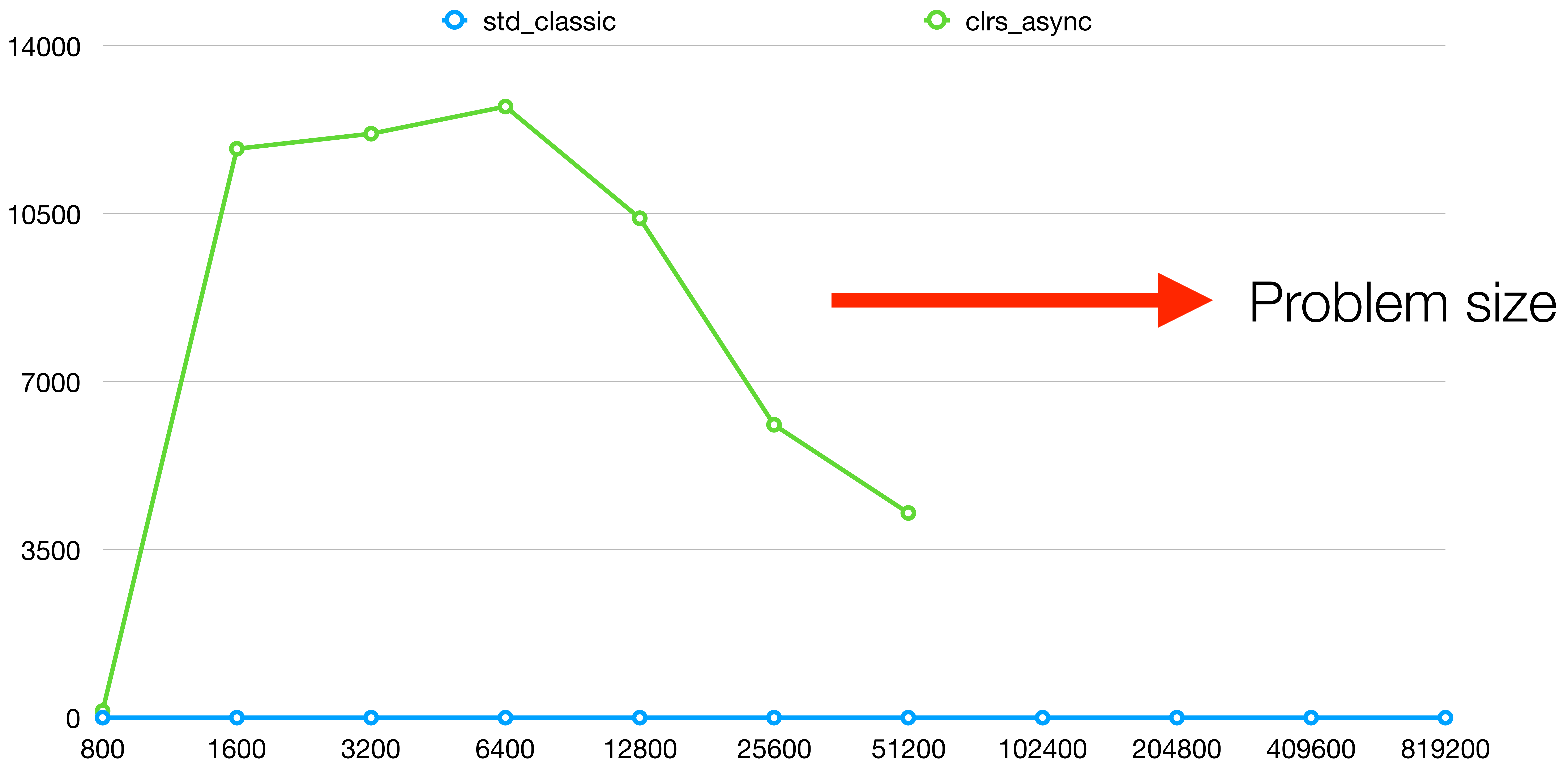
P-Scan: Performance using std::async()



TechAtBloomberg.com

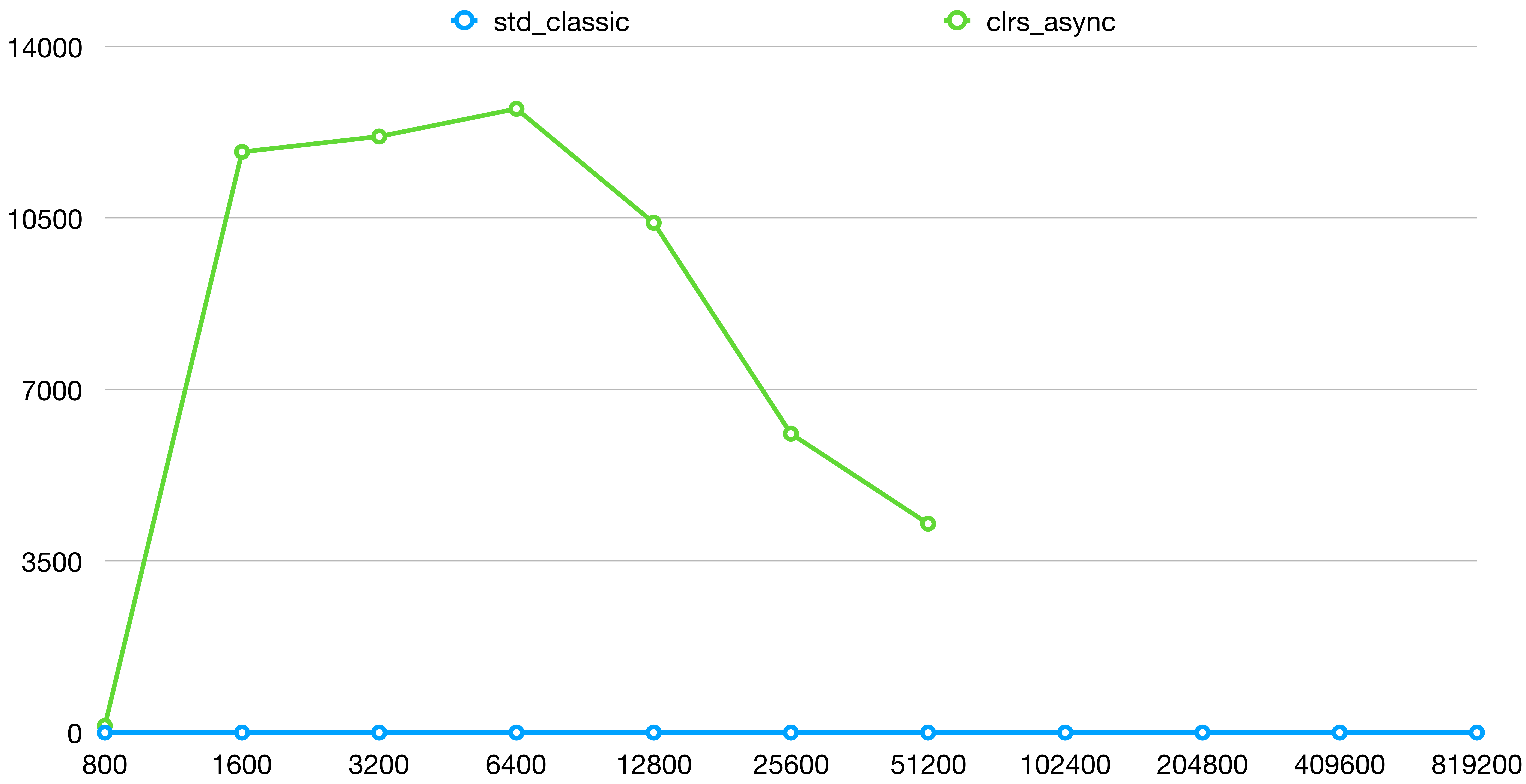
Bloomberg

P-Scan: Performance using std::async()

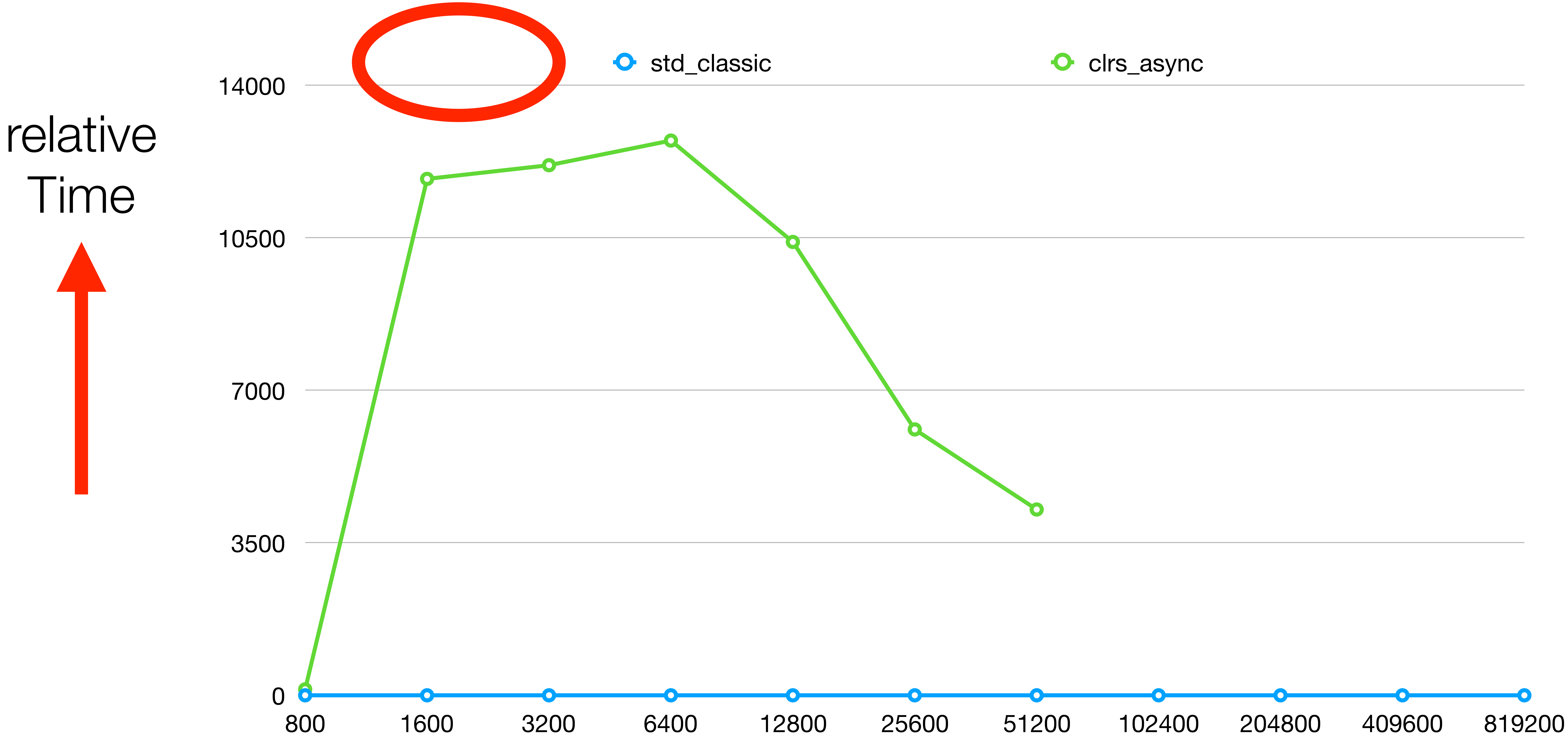


P-Scan: Performance using std::async()

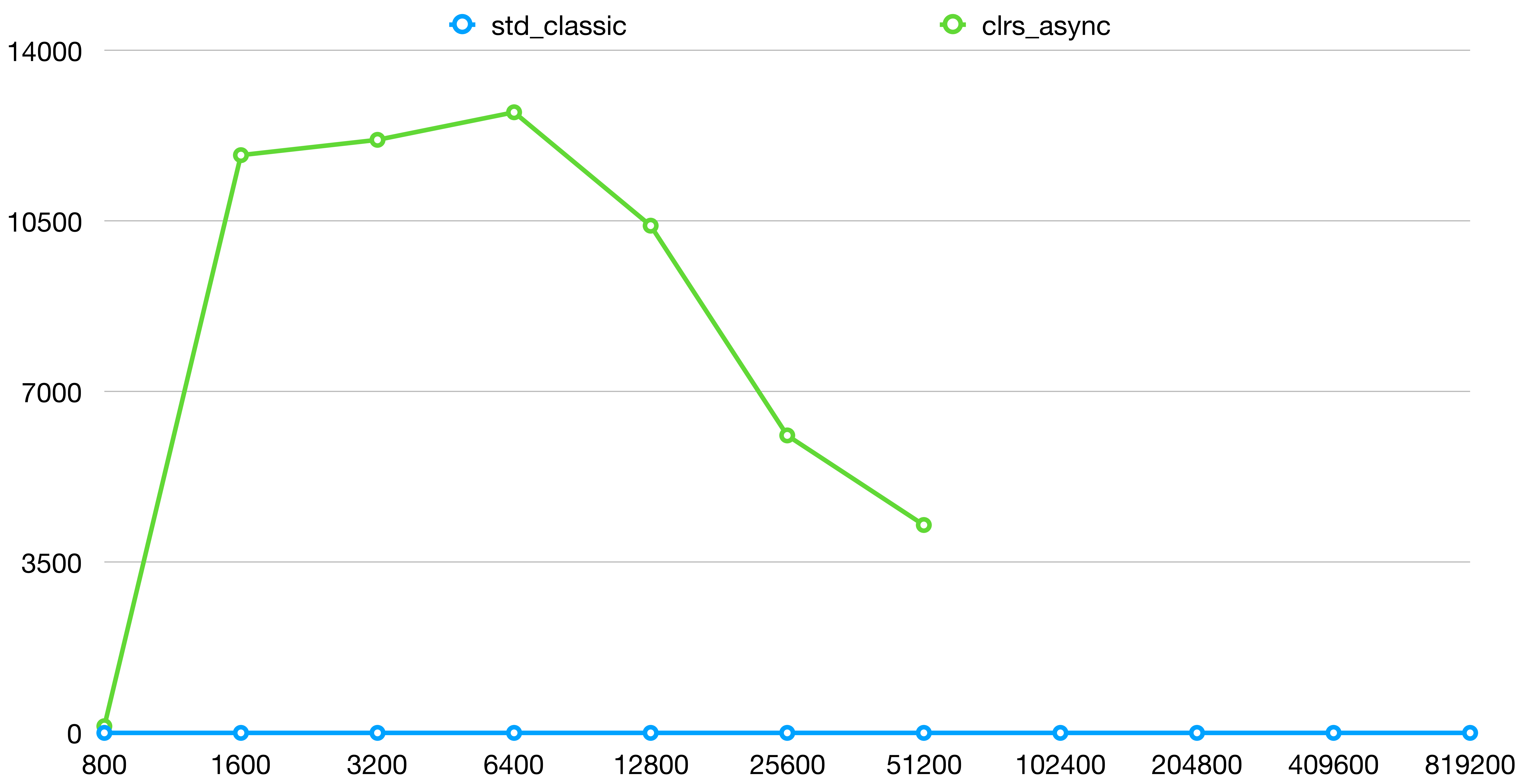
relative
Time



P-Scan: Performance using std::async()



P-Scan: Performance using std::async()



TechAtBloomberg.com

Bloomberg

No Thread Pool in std::async()

- There is no promise that a task started with std::async() doesn't block.
- The example work-load actually does block:

```
auto fut = std::async([&]{ p_scan_down(v, b, b + k, tmp, to, op); });
p_scan_down(op(v, tmp[k]), b + k, end, tmp + k, to + k, op);
fut.wait();
```

- Thus, each task is run on its own thread.

Scheduling Tasks Isn't Free

- The algorithm uses lots of tiny tasks (unless the operation is huge)
- In theory that is fine, in practice it is too expensive:
  - The overhead of managing and scheduling tasks matters
  - There aren't infinite processors to take advantage of many tasks
- Processing the input in blocks improves the performance

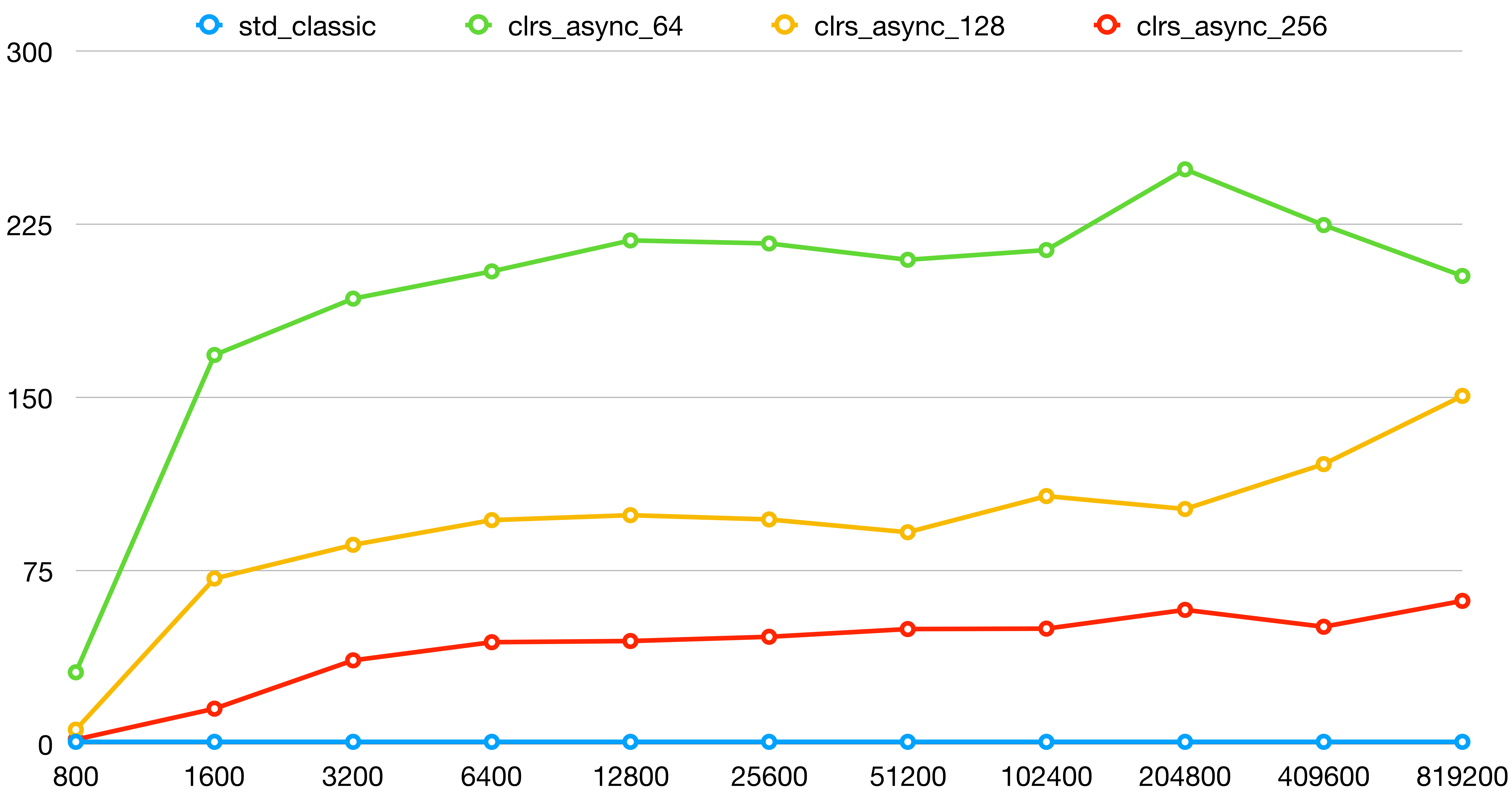
P-Scan-Up: processing data in blocks

```
template <typename In, typename Tmp, typename Op>
auto p_scan_up(In b, In end, Tmp tmp, Op op) {
    auto n = std::distance(b, end);
    if (n < MinSize) { return std::accumulate(begin + 1, end, *begin, op); }
    else {
        auto k = n / 2;
        auto fut = std::async([&]{ tmp[k] = p_scan_up(b, b + k, tmp, op); });
        auto right = p_scan_up(b + k, end, tmp + k, op);
        fut.wait();
        return op(tmp[k], right);
    }
}
```

P-Scan-Down: processing data in blocks

```
template <typename V, typename In, typename T, typename To, typename Op>
void p_scan_down(V v, In b, In end, T tmp, To to, Op op) {
    auto n = std::distance(b, end);
    if (n < MinSize) { inclusive_scan(b, end, to, op, v); }
    else {
        auto k = n / 2;
        auto fut = std::async([&]{ p_scan_down(v, b, b + k, tmp, to, op); });
        p_scan_down(op(v, tmp[k]), b + k, end, tmp + k, to + k, op);
        fut.wait();
    }
}
```

P-Scan: Performance using std::async()



Bloomberg

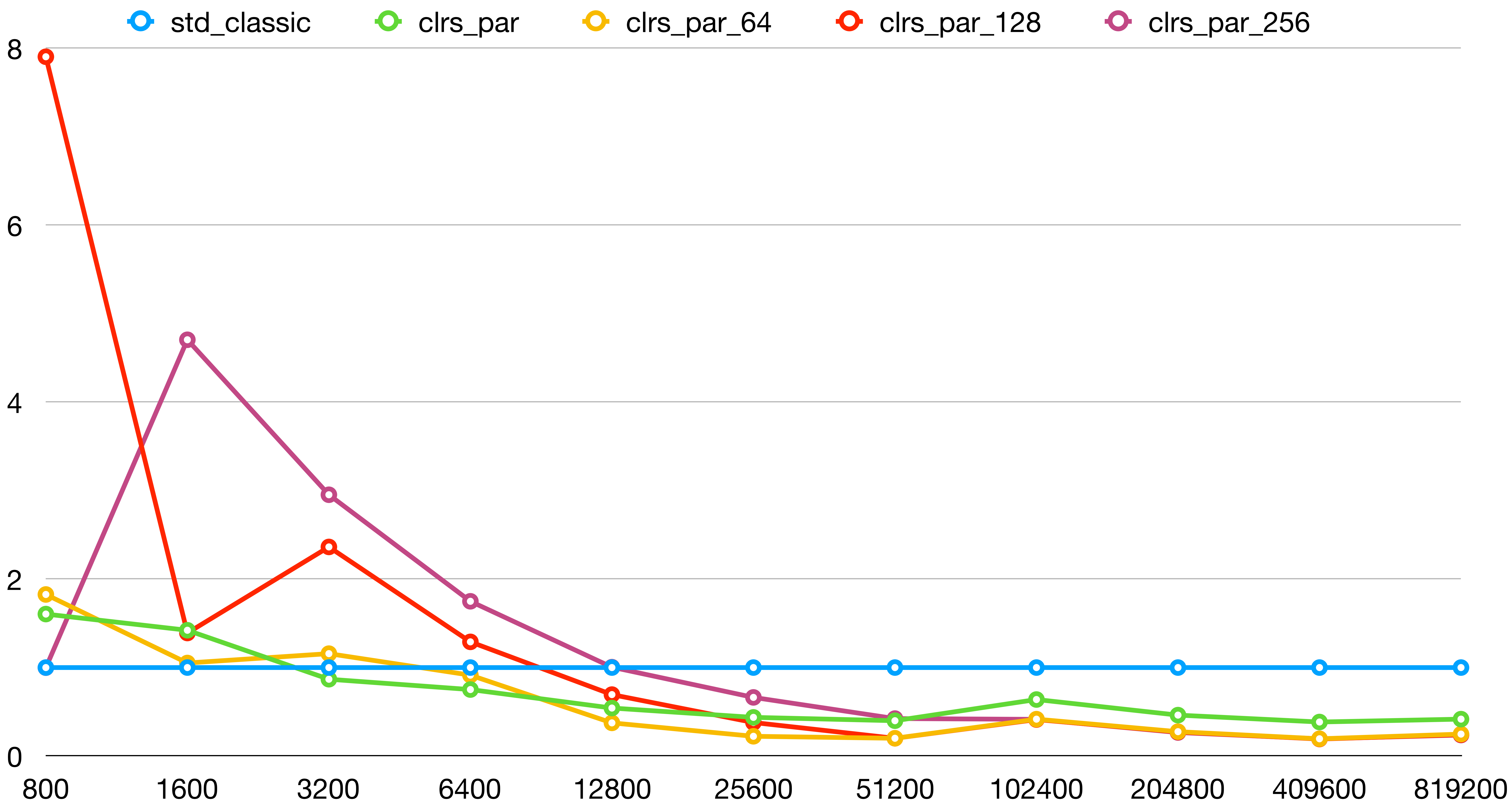
P-Scan-Up: processing data in blocks using tab::task_group

```
template <typename In, typename Tmp, typename Op>
auto p_scan_up(In b, In end, Tmp tmp, Op op) {
  auto n = std::distance(b, end);
  if (n < MinSize) { return std::accumulate(begin + 1, end, *begin, op); }
  else {
    auto k = n / 2;
    tbb::task_group g; g.run([&]{ tmp[k] = p_scan_up(b, b + k, tmp, op); });
    auto right = p_scan_up(b + k, end, tmp + k, op);
    g.wait();
    return op(tmp[k], right);
  }
}
```

P-Scan-Down: processing data in blocks using tab::task_group

```
template <typename V, typename In, typename T, typename To, typename Op>
void p_scan_down(V v, In b, In end, T tmp, To to, Op op) {
    auto n = std::distance(b, end);
    if (n < MinSize) { inclusive_scan(b, end, to, op, v); }
    else {
        auto k = n / 2;
        tbb::task_group g; g.run([&]{ p_scan_down(v, b, b + k, tmp, to, op); });
        p_scan_down(op(v, tmp[k]), b + k, end, tmp + k, to + k, op);
        g.wait();
    }
}
```

P-Scan: Performance using tbb::task_group



Parallel Algorithm: reformulate to be more friendly to parallelization

- Recursive decomposition is a natural match for divide and conquer
- Using an iteration makes the problem accessible to parallel tools:
  - OpenMP
  - Thread Building Blocks (TBB)
  - Standard library algorithms - to some extent

Parallel Algorithm (Overview)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
    ⊞ Auxiliary set up (types and variables)
    ⊞ Compute sums of subsequences (parallel)
    ⊞ Compute results for elements on boundaries
    ⊞ Compute the complete values (parallel)
}
```

TechAtBloomberg.com

Bloomberg

Parallel Algorithm (Overview)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
    ⊞ Auxiliary set up (types and variables)
    ⊞ Compute sums of subsequences (parallel)
    ⊞ Compute results for elements on boundaries
    ⊞ Compute the complete values (parallel)
}
```

Parallel Algorithm (Setup)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
  ⊟ Auxiliary set up (types and variables)
  using ptrdiff_t  = typename std::iterator_traits<InIt>::difference_type;
  using value_type = std::decay_t<decltype(op(*begin, *begin))>;
  ptrdiff_t const size = std::distance(begin, end);
  ptrdiff_t const count = (size + chunk - 1) / chunk;
  std::vector<std::optional<value_type>> tmp(count);

  ⊞ Compute sums of subsequences (parallel)
  ⊞ Compute results for elements on boundaries
  ⊞ Compute the complete values (parallel)
```

Parallel Algorithm (Setup)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
    ⊟ Auxiliary set up (types and variables)
    using ptrdiff_t  = typename std::iterator_traits<InIt>::difference_type;
    using value_type = std::decay_t<decltype(op(*begin, *begin))>;
    ptrdiff_t const size = std::distance(begin, end);
    ptrdiff_t const count = (size + chunk - 1) / chunk;
    std::vector<std::optional<value_type>> tmp(count);

    ⊞ Compute sums of subsequences (parallel)
    ⊞ Compute results for elements on boundaries
    ⊞ Compute the complete values (parallel)
```

Parallel Algorithm (Overview)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
    ⊞ Auxiliary set up (types and variables)
    ⊞ Compute sums of subsequences (parallel)
    ⊞ Compute results for elements on boundaries
    ⊞ Compute the complete values (parallel)
}
```

Parallel Algorithm (Overview)

template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {

- ⊞ Auxiliary set up (types and variables)
- ⊞ **Compute sums of subsequences (parallel)**
- ⊞ Compute results for elements on boundaries
- ⊞ Compute the complete values (parallel)

}

Parallel Algorithm (Pre Processing, Sequential)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
  ⊞ Auxiliary set up (types and variables)
  ⊟ Compute sums of subsequences (parallel)

  for (ptrdiff_t i = 0; i < count - 1; ++i) {
    tmp[i] = std::accumulate(begin + chunk * i + 1, begin + chunk * (i + 1),
                             begin[chunk * i], op);
  }
  ⊞ Compute results for elements on boundaries
  ⊞ Compute the complete values (parallel)
```

Parallel Algorithm (Pre Processing, OpenMP)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
  ⊞ Auxiliary set up (types and variables)
  ⊟ Compute sums of subsequences (parallel)
  #pragma omp parallel for
  for (ptrdiff_t i = 0; i < count - 1; ++i) {
    tmp[i] = std::accumulate(begin + chunk * i + 1, begin + chunk * (i + 1),
                             begin[chunk * i], op);
  }
  ⊞ Compute results for elements on boundaries
  ⊞ Compute the complete values (parallel)
```

TechAtBloomberg.com

Bloomberg

Parallel Algorithm (Pre Processing, tbb::task_group)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
    ⊞ Auxiliary set up (types and variables)
    ⊟ Compute sums of subsequences (parallel)
    tbb::task_group group;
    for (ptrdiff_t i = 0; i < count - 1; ++i) group.run([&, i]{
        tmp[i] = std::accumulate(begin + chunk * i + 1, begin + chunk * (i + 1),
                                 begin[chunk * i], op);
    }); group.wait();
    ⊞ Compute results for elements on boundaries
    ⊞ Compute the complete values (parallel)
```

Parallel Algorithm (Pre Processing, tbb::parallel_for)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
  ⊞ Auxiliary set up (types and variables)
  ⊟ Compute sums of subsequences (parallel)

  tbb::parallel_for(ptrdiff_t(), count - 1, [&](auto i){
    tmp[i] = std::accumulate(begin + chunk * i + 1, begin + chunk * (i + 1),
                             begin[chunk * i], op);
  });
  ⊞ Compute results for elements on boundaries
  ⊞ Compute the complete values (parallel)
```

Parallel Algorithm (Pre Processing, use Executor)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
  ⊞ Auxiliary set up (types and variables)
  ⊟ Compute sums of subsequences (parallel)

  for_each_subrange(exec, chunk, begin, end,
    [&](auto i, auto b, auto e){
      tmp[i + 1] = std::accumulate(b + 1, e, *b, op);
  });
  ⊞ Compute results for elements on boundaries
  ⊞ Compute the complete values (parallel)
```

TechAtBloomberg.com

Bloomberg

Parallel Algorithm (Overview)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
    ⊞ Auxiliary set up (types and variables)
    ⊞ Compute sums of subsequences (parallel)
    ⊞ Compute results for elements on boundaries
    ⊞ Compute the complete values (parallel)
}
```

Parallel Algorithm (Overview)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
    ⊞ Auxiliary set up (types and variables)
    ⊞ Compute sums of subsequences (parallel)
    ⊞ Compute results for elements on boundaries
    ⊞ Compute the complete values (parallel)
}
```

Parallel Algorithm (Intermediate Procesing)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
  ⊞ Auxiliary set up (types and variables)
  ⊞ Compute sums of subsequences (parallel)
  ⊟ Compute results for elements on boundaries
  partial_sum(tmp.begin(), tmp.end(), tmp.begin(),
              [op](auto& a, auto& b){ return op(*a, *b); });

  ⊞ Compute the complete values (parallel)
}
```

Parallel Algorithm (Overview)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
    ⊞ Auxiliary set up (types and variables)
    ⊞ Compute sums of subsequences (parallel)
    ⊞ Compute results for elements on boundaries
    ⊞ Compute the complete values (parallel)
}
```

Parallel Algorithm (Overview)

template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {

- ⊞ Auxiliary set up (types and variables)
- ⊞ Compute sums of subsequences (parallel)
- ⊞ Compute results for elements on boundaries
- ⊞ **Compute the complete values (parallel)**

}

Parallel Algorithm (Post Processing)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
  ⊞ Auxiliary set up (types and variables)
  ⊞ Compute sums of subsequences (parallel)
  ⊞ Compute results for elements on boundaries
  ⊟ Compute the complete values (parallel)

  for (ptrdiff_t i = 0; i < count; ++i) {
    auto it = begin + chunk * i, ce = it + std::min(end - it, chunk);
    auto value = i? op(*tmp[i - 1], *it): *it;
    inclusive_scan(it + 1, ce, to + (it - begin) + 1, op, to[it - begin] = value);
  }
```

TechAtBloomberg.com

Bloomberg

Parallel Algorithm (Post Processing, OpenMP)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
  ⊞ Auxiliary set up (types and variables)
  ⊞ Compute sums of subsequences (parallel)
  ⊞ Compute results for elements on boundaries
  ⊟ Compute the complete values (parallel)
  #pragma omp parallel for
  for (ptrdiff_t i = 0; i < count; ++i) {
    auto it = begin + chunk * i, ce = it + std::min(end - it, chunk);
    auto value = i? op(*tmp[i - 1], *it): *it;
    inclusive_scan(it + 1, ce, to + (it - begin) + 1, op, to[it - begin] = value);
  }
```

Parallel Algorithm (Post Processing, tbb::task_group)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
  ⊞ Auxiliary set up (types and variables)
  ⊞ Compute sums of subsequences (parallel)
  ⊞ Compute results for elements on boundaries
  ⊟ Compute the complete values (parallel)
  tbb::task_group group;
  for (ptrdiff_t i = 0; i < count; ++i) group.run([&, i]{
    auto it = begin + chunk * i, ce = it + std::min(end - it, chunk);
    auto value = i? op(*tmp[i - 1], *it): *it;
    inclusive_scan(it + 1, ce, to + (it - begin) + 1, op, to[it - begin] = value);
  }); group.wait();
```

Parallel Algorithm (Post Processing, tbb::parallel_for)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
  ⊞ Auxiliary set up (types and variables)
  ⊞ Compute sums of subsequences (parallel)
  ⊞ Compute results for elements on boundaries
  ⊟ Compute the complete values (parallel)

  tbb::parallel_for(ptrdiff_t(), count, [&](auto i){
    auto it = begin + chunk * i, ce = it + std::min(end - it, chunk);
    auto value = i? op(*tmp[i - 1], *it): *it;
    inclusive_scan(it + 1, ce, to + (it - begin) + 1, op, to[it - begin] = value);
  });
```

TechAtBloomberg.com

Bloomberg

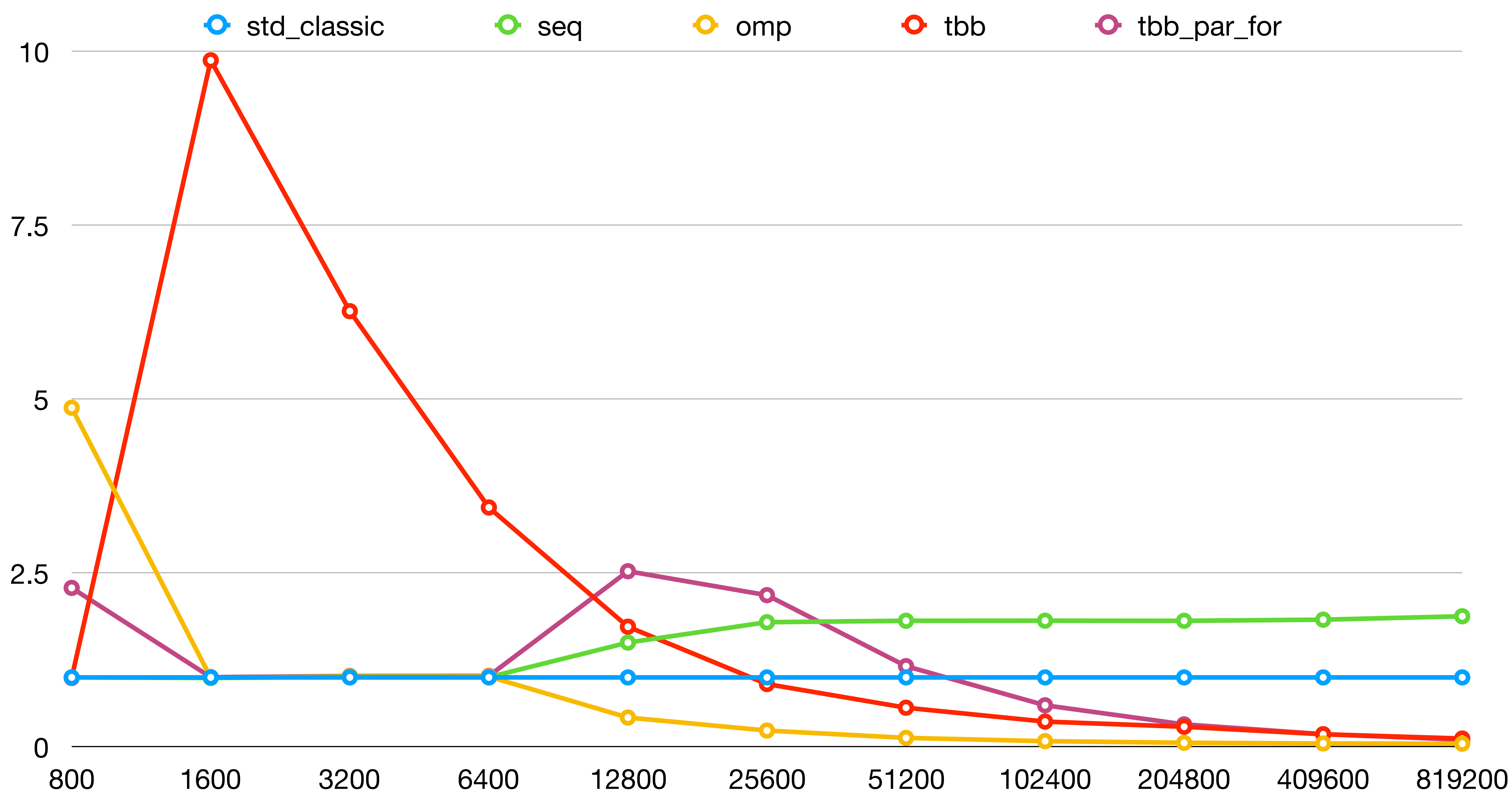
Parallel Algorithm (Post Processing, use Executor)

```
template <class InIt, class OutIt, class Op>
OutIt inclusive_scan(InIt begin, InIt end, OutIt to, Op op) {
  ⊞ Auxiliary set up (types and variables)
  ⊞ Compute sums of subsequences (parallel)
  ⊞ Compute results for elements on boundaries
  ⊟ Compute the complete values (parallel)
  for_each_subrange(exec, chunk, begin, end,
     [&](auto i,  auto sbegin, auto send) {
           auto value = i? op(*tmp[i], *sbegin): *sbegin;
           auto off(std::distance(begin, sbegin));
           inclusive_scan(std::next(sbegin), send, to + off + 1, op, to[off] = value);
  });
```

TechAtBloomberg.com

Bloomberg

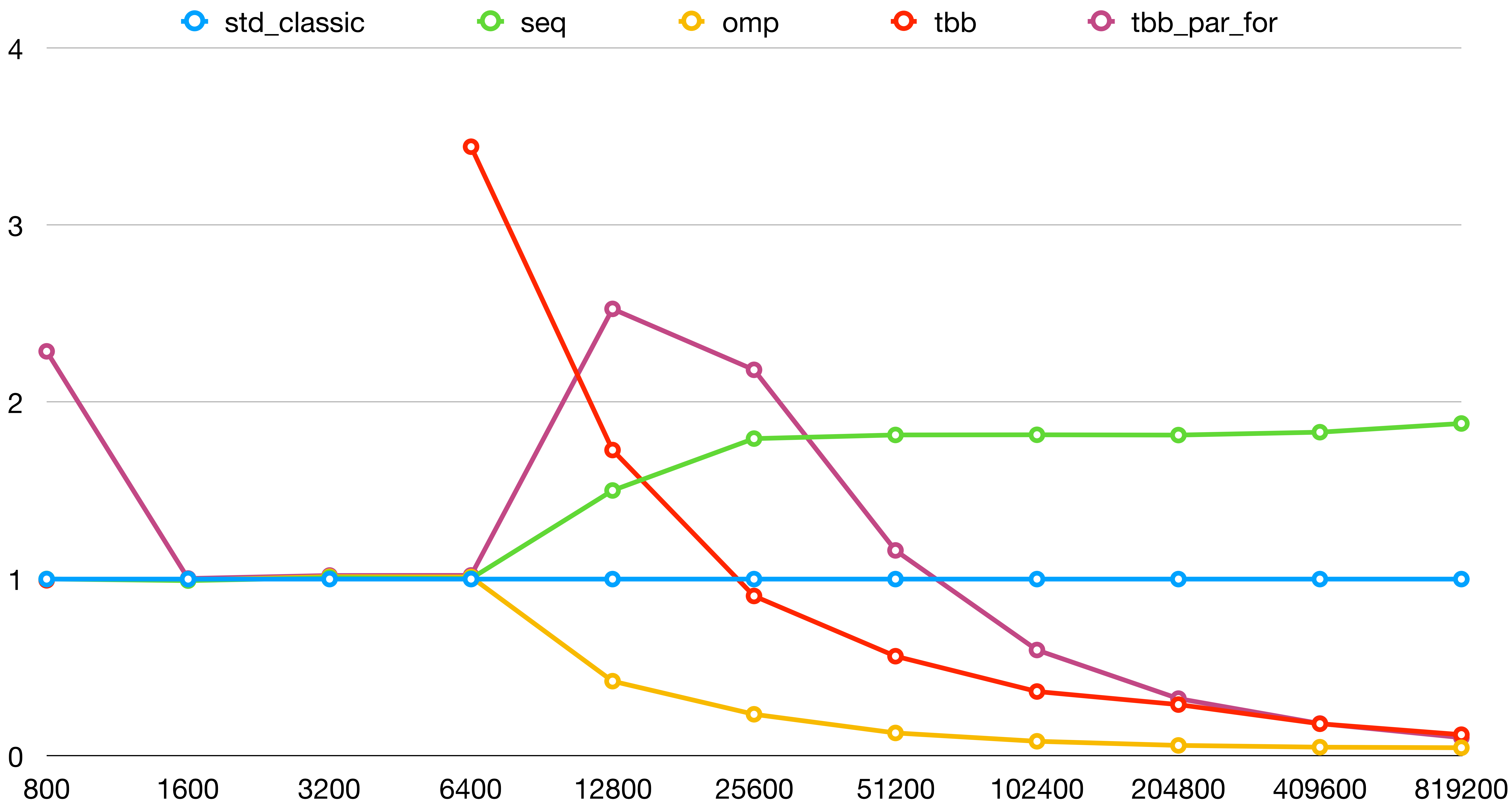
Performance using iterative algorithms



TechAtBloomberg.com

Bloomberg

Performance using iterative algorithms



TechAtBloomberg.com

Bloomberg

for_each_range()

```
template <typename Executor, typename Size, typename InIt, typename Fun>
void for_each_subrange(Executor exec, Size chunk, InIt it, InIt end, Fun fun) {
    if (it == end) { return; }
    ptrdiff_t const s = std::distance(it, end), count = (s + chunk - 1) / chunk;
    latch l(count);
    for (ptrdiff_t i(0); i != count - 1; ++i) {
        auto cend(std::next(it, chunk));
        exec.execute([&, i, it, cend, a = latch_arriver(l)]{ fun(i, it, cend); });
        it = cend;
    }
    exec.execute([&, it, a = latch_arriver(l)]{ fun(count - 1, it, end); });
    l.wait();
}
```

The Missing Bits

- OpenMP and TBB use a task scheduler
- The task schedule is backed by a thread pool:
    - Multiple threads processing queued jobs
    - TBB uses a *job stealing* thread pool

Executor: on the same thread

```
class immediate_executor {
public:
    explicit  immediate_executor(int =  0) {}
    template <typename Fun>
    void execute(Fun&& fun) {
        fun();
    }
};
```

Executor: thread pool (interface)

```
class thread_pool {
    bool                                    d_flag{true};
    std::mutex                              d_mutex;
    std::condition_variable                 d_condition;
    std::deque<function<void()>>            d_queue;
    std::deque<jthread>                     d_pool;
    void work();
public:
    thread_pool(int count);
    ~thread_pool() { this->stop(); } void stop();
    template <typename Fun> void execute(Fun&& fun);
};
```

Executor: thread pool (starting threads)

```
thread_pool::thread_pool(int count) {
    std::lock_guard cerberus(this->d_mutex);
    count = std::max(1, count);
    for (int i{0}; i != count; ++i) {
        this->d_pool.emplace_back(&thread_pool::work, this);
    }
}
```

TechAtBloomberg.com

Bloomberg

Executor: thread pool (enqueue a job)

```
template <typename Fun>
void thread_pool::execute(Fun&& fun) {
    {
        std::lock_guard cerberus(this->d_mutex);
        this->d_queue.emplace_back(std::forward<Fun>(fun));
    }
    this->d_condition.notify_one();
}
```

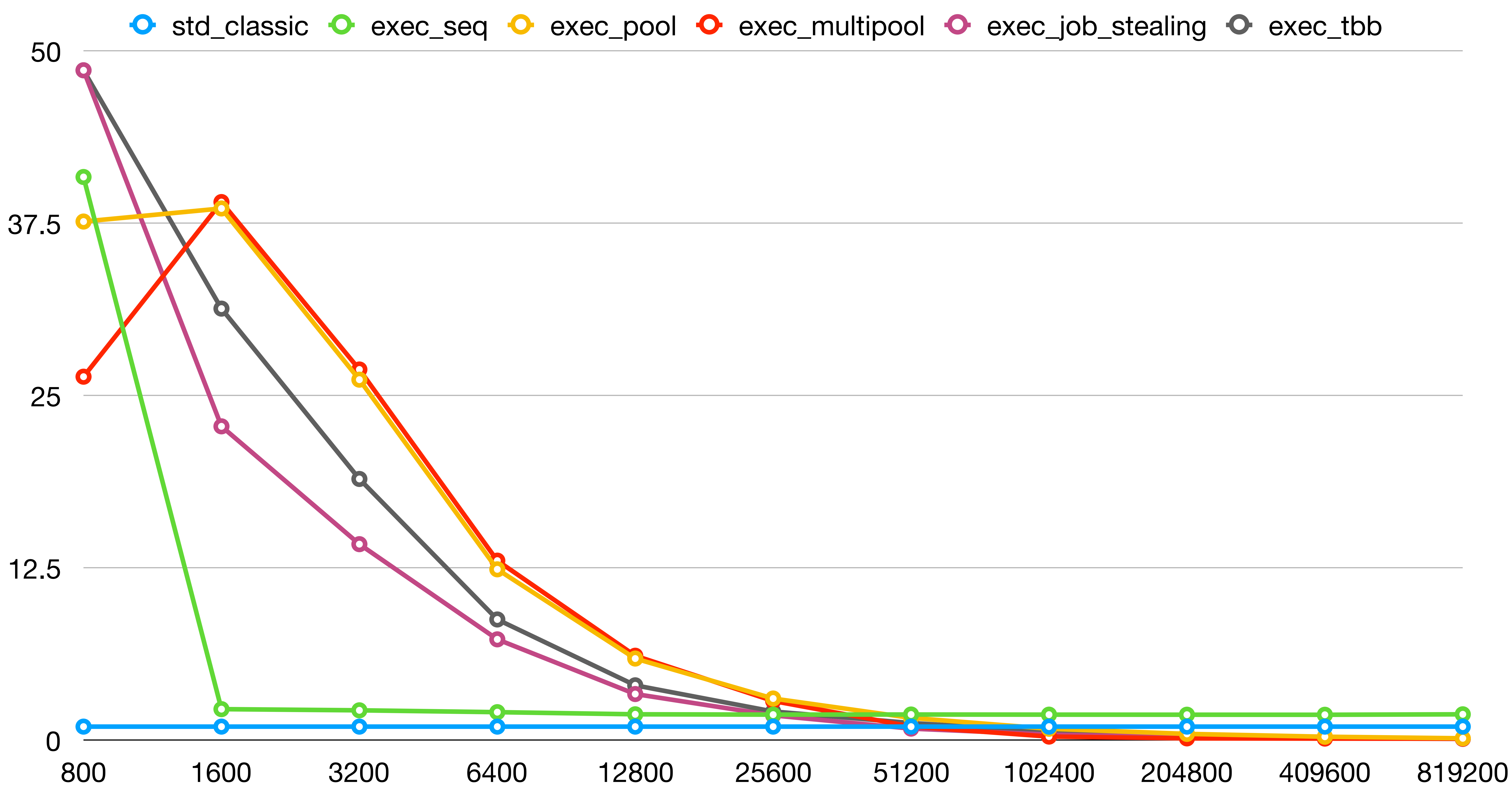
Executor: thread pool (execute work)

```
void thread_pool::work() {
    while (true) {
        std::unique_lock cerberus(this->d_mutex);
        this->d_condition.wait(cerberus,
            [this]{ return !this->d_queue.empty() || !this->d_flag; });
        if (this->d_queue.empty()) { break; }
        auto fun(std::move(this->d_queue.front()));
        this->d_queue.pop_front();
        cerberus.unlock();
        fun();
    }
}
```

Executor: pool executor

```
class pool_executor {
  std::shared_ptr<thread_pool> d_pool;
public:
    explicit pool_executor(int count = std::thread::hardware_concurrency())
        : d_pool(std::make_shared<thread_pool>(count)) {
    }
    void stop() { this->d_pool->stop(); }
    template <typename Fun>
    void execute(Fun&& fun) {
        this->d_pool->execute(std::forward<Fun>(fun)); }
};
```

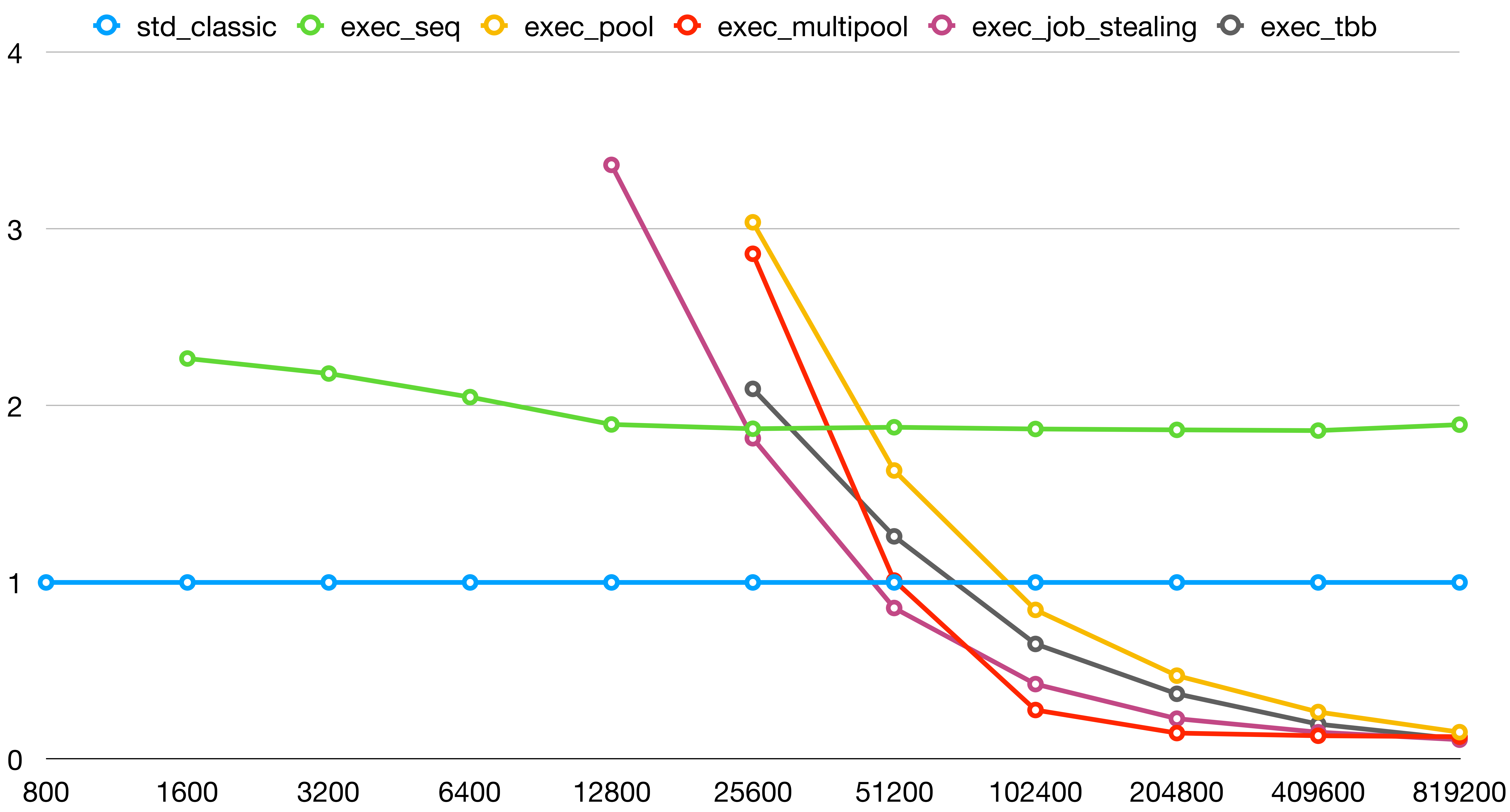
Performance using iterative algorithms with executors



TechAtBloomberg.com

Bloomberg

Performance using iterative algorithms with executors



TechAtBloomberg.com

Bloomberg

Proper Way to Do Inclusive Scan

- The algorithm can be implemented, but it is already in the standard C++ library

  to_end = std::inclusive_scan(std::execution::par, begin, end, to, op);

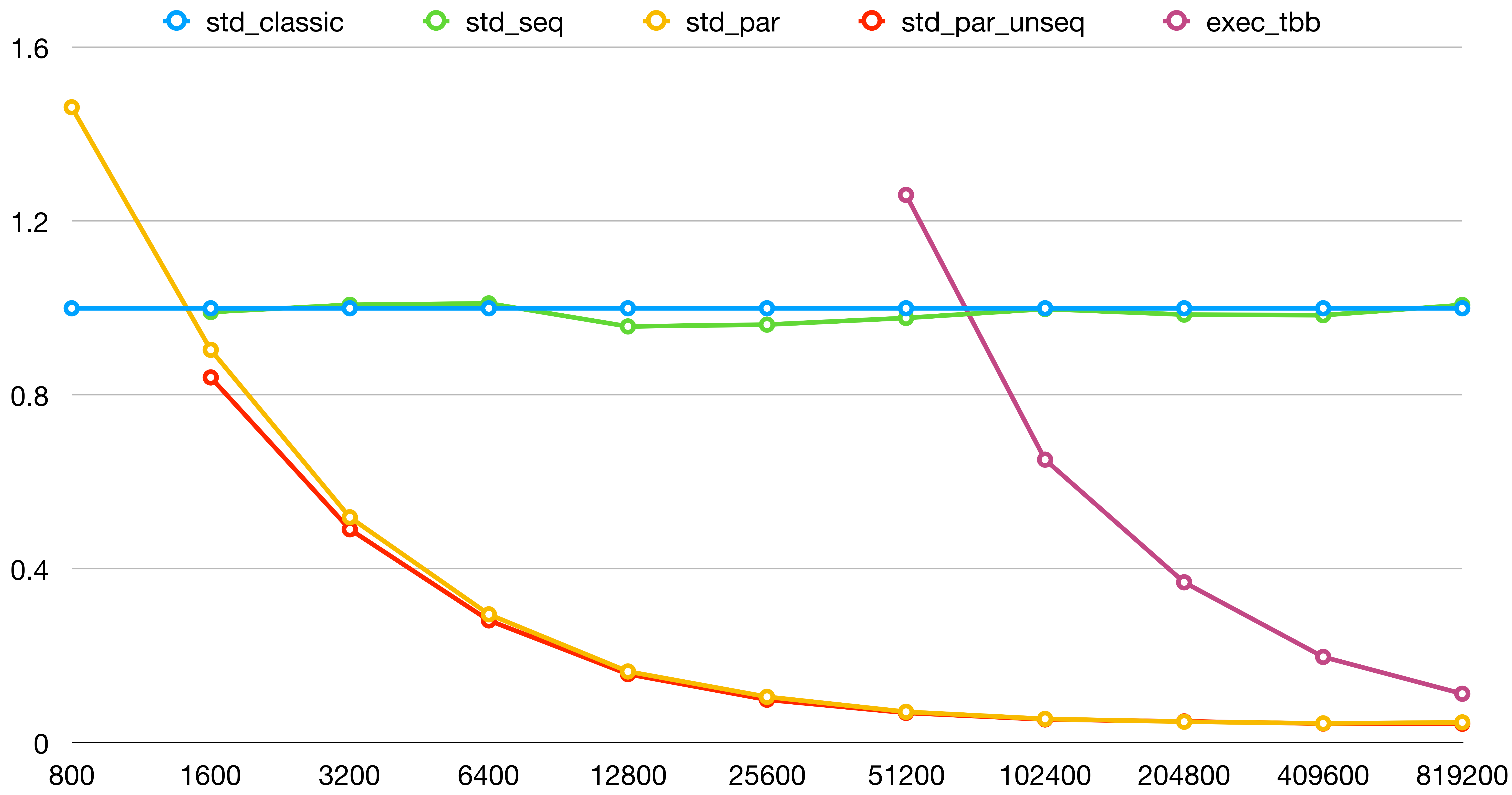
  to_end = std::inclusive_scan(std::execution::par_unseq, begin, to, op);

  to_end = std::inclusive_scan(std::execution::seq, begin, end, to, op);

- Most algorithms in the standard C++ library have a parallel version

- These provide a benchmark to compare against

Performance of Standard Library Algorithms



Conclusion

- Parallel algorithms can speed up processing of larger data sets
- … even if the processing isn’t massively parallel
- Implementing parallel algorithms isn’t magic
    - The algorithm from the book needs quite a bit of work to make it practical
    - Doing so would be easier with standard thread pools (executors)
- The standard C++ library has parallel versions and does this better

Thank you!

Questions ?