

Ruminations on Parallel STL

Zoltán Porkoláb

<https://gsd.web.elte.hu>
gsd@inf.elte.hu

Bolyai kollégium
2024.11.27.

Warm up task

- Create a function to count the words in a character vector
 - A word is a continuous sequence of non-whitespace characters

Word count

```
#include <cctype>
#include <iostream>
#include <vector>
#include <algorithm>

int wordcount(const std::vector<char>& v)
{
    int cnt = 0;
    char prev = '\n'; // the imaginary char on -1 position is a white space.

    for (char curr : v)
    {
        if (std::isspace(prev) && !std::isspace(curr)) // new word starts
        {
            ++cnt;
        }
        prev = curr;
    }
    return cnt;
}

int main()
{
    const char s[] = "this is a sentence with a few words inside";
    std::vector<char> v{std::begin(s), std::end(s)};
    std::cout << wordcount(v) << '\n';
    return 0;
}

$ ./a.out
```

Word count

```
#include <cctype>
#include <iostream>
#include <vector>
#include <algorithm>

int wordcount(const std::vector<char>& v)
{
    int cnt = 0;
    char prev = '\n'; // the imaginary char on -1 position is a white space.

    for (char curr : v)
    {
        if (std::isspace(prev) && !std::isspace(curr)) // new word starts
        {
            ++cnt;
        }
        prev = curr;
    }
    return cnt;
}

int main()
{
    const char s[] = "this is a sentence with a few words inside ";
    std::vector<char> v{std::begin(s), std::end(s)};
    std::cout << wordcount(v) << '\n';
    return 0;
}

$ ./a.out
```

Word count

```
#include <cctype>
#include <iostream>
#include <vector>
#include <algorithm>

int wordcount(const std::vector<char>& v)
{
    int cnt = 0;
    char prev = '\n'; // the imaginary char on -1 position is a white space.

    for (char curr : v)
    {
        if (std::isspace(prev) && !std::isspace(curr)) // new word starts
        {
            ++cnt;
        }
        prev = curr;
    }
    return cnt;
}

int main()
{
    const char s[] = "this is a sentence with a few words inside ";
    std::vector<char> v{std::begin(s), std::end(s)-1};
    std::cout << wordcount(v) << '\n';
    return 0;
}

$ ./a.out
```

Word count

```
#include <cctype>
#include <iostream>
#include <vector>
#include <algorithm>

int wordcount(const std::vector<char>& v)
{
    if (v.empty())
        return 0;

    char prev = *v.begin();

    return std::accumulate(v.begin() + 1, v.end(), // sum = op(sum, curr)
                          std::isspace(prev) ? 0 : 1,
                          [&prev](int sum, char curr) {
                                int incr = std::isspace(prev) && !std::isspace(curr);
                                prev = curr;
                                return sum + incr;
                            });
}

int main()
{
    const char s[] = "this is a sentence with a few words inside ";
    std::vector<char> v{std::begin(s), std::end(s) - 1};
    std::cout << wordcount(v) << '\n';
    return 0;
}

$ ./a.out
```

Word count

```
#include <cctype>
#include <iostream>
#include <vector>
#include <algorithm>

int wordcount(const std::vector<char>& v)
{
    if (v.empty())
        return 0;

    char prev = *v.begin();

    return std::accumulate(v.begin() + 1, v.end(), // sum = op(sum, curr)
                          !std::isspace(prev), // std::isspace(prev) ? 0 : 1
                          [&prev](int sum, char curr) {
                                int incr = std::isspace(prev) && !std::isspace(curr);
                                prev = curr;
                                return sum + incr;
                        });
}

int main()
{
    const char s[] = "this is a sentence with a few words inside ";
    std::vector<char> v{std::begin(s), std::end(s) - 1};
    std::cout << wordcount(v) << '\n';
    return 0;
}

$ ./a.out
```

Word count

```
#include <cctype>
#include <iostream>
#include <vector>
#include <algorithm>

int wordcount(const std::vector<char>& v) // from Bryce Adelstein Lelbach
{
    if (v.empty())
        return 0;

    return std::transform_reduce(
        v.begin(), v.end() - 1, // binop(*it1, *i2)
        v.begin() + 1,
        std::isspace(v.front()) ? 0 : 1,
        std::plus{},
        [] (char curr, char next) {
            return std::isspace(curr) && !std::isspace(next);
        });
}

int main()
{
    const char s[] = "this is a sentence with a few words inside ";
    std::vector<char> v{std::begin(s), std::end(s) - 1};
    std::cout << wordcount(v) << '\n';
    return 0;
}

$ ./a.out
```

Word count

```
#include <cctype>
#include <iostream>
#include <vector>
#include <algorithm>
#include <execution>

int wordcount(const std::vector<char>& v)
{
    if (v.empty())
        return 0;

    return std::transform_reduce(std::execution::par,
                                v.begin(), v.end() - 1, // binop(*it1, *i2)
                                v.begin() + 1,
                                std::isspace(v.front()) ? 0 : 1,
                                std::plus{},
                                [] (char curr, char next) {
                                    return std::isspace(curr) && !std::isspace(next);
                                });
}

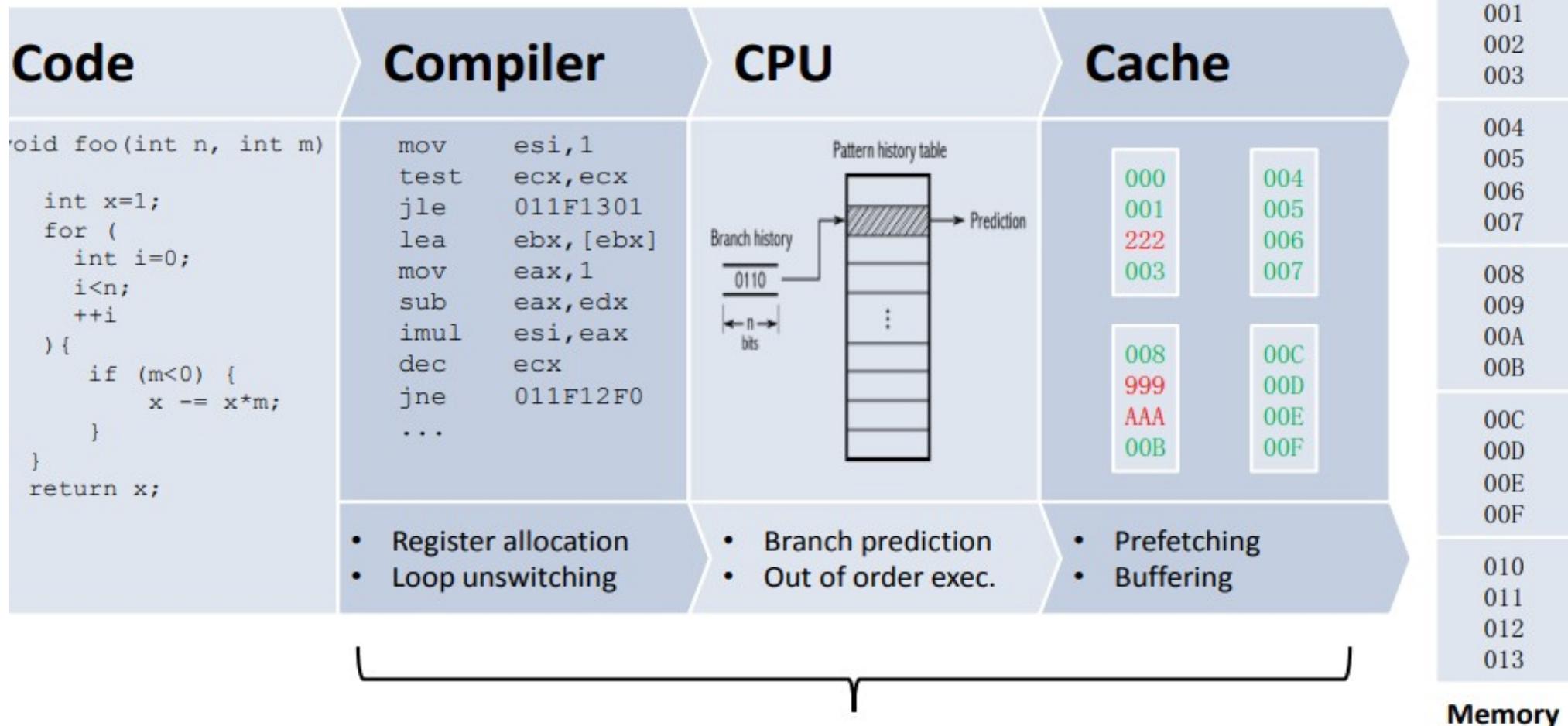
int main() // g++ -ltbb
{
    const char s[] = "this is a sentence with a few words inside ";
    std::vector<char> v{std::begin(s), std::end(s) - 1};
    std::cout << wordcount(v) << '\n';
    return 0;
}

$ ./a.out
```

Concurrent programming in C++11

- Multithreading is just one damn thing after, before, or simultaneous with another. --Andrei Alexandrescu
- Problems with C++98 memory model
- C++11 memory model
- `std::thread`, `std::jthread` (C++20), mutexes, guards
- Conditional variable
- Future/Promise/Async
- Parallel STL
- Atomics
- Lock free/wait free programming

Parallel Programming is hard

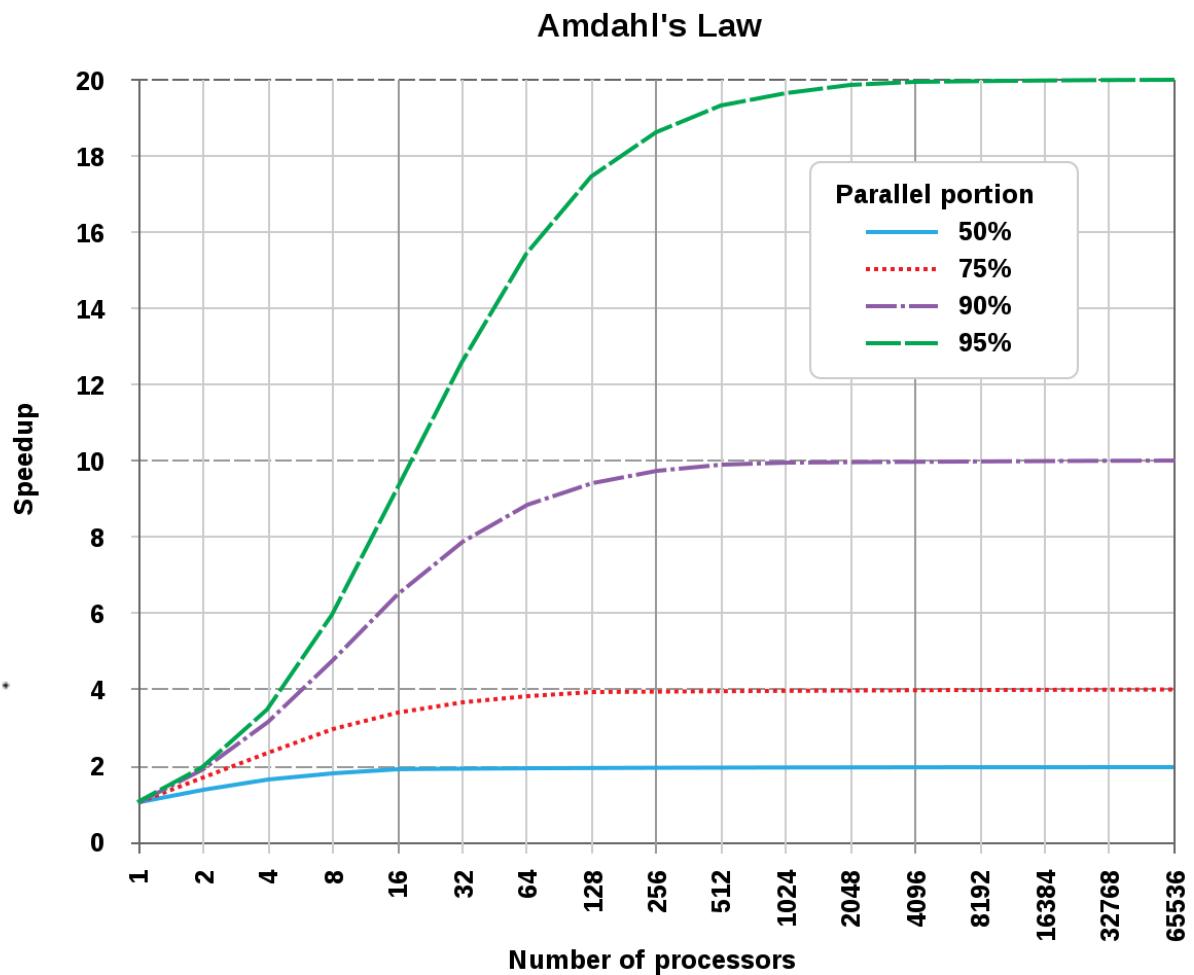


Valentin Ziegler, Fabio Fracassi C++ Memory Model (Meeting C++ Berlin, 2014)
https://www.think-cell.com/en/career/talks/pdf/think-cell_talk_memorymodel.pdf

Parallel Programming is hard

- Improper use of locks can cause deadlock
- Locking/unlocking can cause context switch
 - Clear the cache
 - Further cache-misses
- False sharing
- Amdahl law

$$S_{\text{latency}}(s) = \frac{TW}{T(s)W} = \frac{T}{T(s)} = \frac{1}{1 - p + \frac{p}{s}}.$$



Parallel algorithms (C++17)

- Extends STL algorithms with execution policy
 - `std::execution::seq` Sequential execution
 - `std::execution::par` Parallel execution
 - `std::execution::par_unseq` Parallel SIMD execution
 - `std::execution::unseq` Sequential SIMD execution (C++20)
- Execution policies are “open”
 - Can be extended with new policies

Parallel algorithms (C++17)

- Based on Intel's Threading Building Blocks (TBB)
- Implementation may choose what can be parallelized
- Minimal requirement: forward iterator
- The programmer's task to ensure that element access functions will not cause dead lock or data race
- In case of parallelization and vectorization access must not use any blocking synchronization

Parallel STL

```
// Example from Stroustrup

template<class T, class V>
struct Accum      // simple accumulator function object
{
    T* b;
    T* e;
    V val;
    Accum(T* bb, T* ee, const V& vv) : b{bb}, e{ee}, val{vv} {}
    V operator() () { return std::accumulate(b,e,val); }
};

double comp(vector<double>& v)    // spawn many tasks if v is large enough
{
    if (v.size()<10000) return std::accumulate(v.begin(),v.end(),0.0);

    auto f0 {async(Accum{&v[0],&v[v.size()/4]},0.0)};
    auto f1 {async(Accum{&v[v.size()/4],&v[v.size()/2]},0.0)};
    auto f2 {async(Accum{&v[v.size()/2],&v[v.size()*3/4]},0.0)};
    auto f3 {async(Accum{&v[v.size()*3/4],&v[v.size()]},0.0)};

    return f0.get()+f1.get()+f2.get()+f3.get();
}
```

Parallel STL

```
// Example from cppreference

template<class T, class V>
struct Accum      // simple accumulator function object
{
    T* b;
    T* e;
    V val;
    Accum(T* bb, T* ee, const V& vv) : b{bb}, e{ee}, val{vv} {}
    V operator() () { return std::accumulate(b,e,val); }
};

double comp(vector<double>& v)
{
    double res = std::reduce(std::execution::par, v.begin(), v.end(), 0.0);
    return res;
}
```

Parallel STL

```
// Example from cppreference
```

```
template<class T, class V>
struct Accum      // simple accumulator function object
{
    T* b;
    T* e;
    V val;
    Accum(T* bb, T* ee, const V& vv) : b{bb}, e{ee}, val{vv} {}
    V operator() () { return std::accumulate(b,e,val); }
};

double comp(vector<double>& v)
{
    double res = std::reduce(std::execution::par, v.begin(), v.end(), 0.0, std::plus<{}>());
    return res;
}
```

Algorithms with execution policy

- `std::adjacent_difference`
- `std::adjacent_find`
- `std::all_of`
- `std::any_of`
- `std::copy`
- `std::copy_if`
- `std::copy_n`
- `std::count`
- `std::count_if`
- `std::equal`
- `std::fill`
- `std::fill_n`
- `std::find`
- `std::find_end`
- `std::find_first_of`
- `std::find_if`
- `std::find_if_not`
- `std::generate`
- `std::generate_n`
- `std::includes`
- `std::inner_product`
- `std::inplace_merge`
- `std::is_heap`
- `std::is_heap_until`
- `std::is_partitioned`
- `std::is_sorted`
- `std::is_sorted_until`
- `std::lexicographical_compare`
- `std::max_element`
- `std::merge`
- `std::min_element`
- `std::minmax_element`
- `std::mismatch`
- `std::move`
- `std::none_of`
- `std::nth_element`
- `std::partial_sort`
- `std::partial_sort_copy`
- `std::partition`
- `std::partition_copy`
- `std::remove`
- `std::remove_copy`
- `std::remove_copy_if`
- `std::remove_if`
- `std::replace`
- `std::replace_copy`
- `std::replace_copy_if`
- `std::replace_if`
- `std::reverse`
- `std::reverse_copy`
- `std::rotate`
- `std::rotate_copy`
- `std::search`
- `std::search_n`
- `std::set_difference`
- `std::set_intersection`
- `std::set_symmetric_difference`
- `std::set_union`
- `std::sort`
- `std::stable_partition`
- `std::stable_sort`
- `std::swap_ranges`
- `std::transform`
- `std::uninitialized_copy`
- `std::uninitialized_copy_n`
- `std::uninitialized_fill`
- `std::uninitialized_fill_n`
- `std::unique`
- `std::unique_copy`

... and a few new algorithms

```
std::for_each
std::for_each_n
std::exclusive_scan
std::inclusive_scan
std::transform_exclusive_scan
std::transform_inclusive_scan
std::reduce
std::transform_reduce
```

Data race

```
//  
// measuring how many comparision happen during sort  
//  
int measure_sort(std::vector vec)  
{  
    int numComp= 0;  
  
    std::sort(std::execution::par, vec.begin(), vec.end(),  
              [&numComp](int a, int b){ numComp++; return a < b; });  
    return numComp;  
}
```

Data race

```
//  
// measuring how many comparision happen during sort  
//  
int measure_sort(std::vector vec)  
{  
    int numComp= 0;  
  
    std::sort(std::execution::par, vec.begin(), vec.end(),  
              [&numComp](int a, int b){ numComp++; return a < b; });  
    return numComp;  
}
```

- Data race == undefined behavior

accumulate() vs reduce()

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<long long> v1;
    for ( int i = 0; i < 10; ++i)
    {
        v1.insert( v1.end(), {0,1,2,3,4}); // creates 50 elements
    }

    long long sum = 0;
    for ( std::size_t i = 0; i < v1.size(); ++i) // summa x^2 x in [0..49]
    {
        sum += v1[i]*v1[i];
    }

    std::cout << sum << '\n';

    return 0;
}

$ ./a.out
300
```

accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>

std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
    for ( int i = 0; i < 10; ++i)
    {
        v1.insert( v1.end(), {0,1,2,3,4}); // creates 50 elements
    }

    auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum); // classical STL

    std::cout << sum1 << '\n';
    return 0;
}

$ ./a.out
300
```

accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>

std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
    for ( int i = 0; i < 10; ++i)
    {
        v1.insert( v1.end(), {0,1,2,3,4}); // creates 50 elements
    }
    // accumulate is guaranteed left associative
    auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum); // classical STL

    std::cout << sum1 << '\n';
    return 0;
}

$ ./a.out
300
```

accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>
#include <execution>

std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
    for ( int i = 0; i < 10; ++i)
    {
        v1.insert( v1.end(), {0,1,2,3,4});
    }
    // accumulate is guaranteed left associative
    auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
    // reduce can work parallel
    auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);

    std::cout << sum1 << ", " << sum2 << '\n';
    return 0;
}

$ ./a.out
300, 300
```

accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>
#include <execution>

std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
    for ( int i = 0; i < 1000; ++i)
    {
        v1.insert( v1.end(), {0,1,2,3,4});
    }
    // accumulate is guaranteed left associative
    auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
    // reduce can work parallel
    auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);

    std::cout << sum1 << ", " << sum2 << '\n';
    return 0;
}

$ ./a.out
30000, 30000
```

accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>
#include <execution>

std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
    for ( int i = 0; i < 1000000; ++i)
    {
        v1.insert( v1.end(), {0,1,2,3,4});
    }
    // accumulate is guaranteed left associative
    auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
    // reduce can work parallel
    auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);

    std::cout << sum1 << ", " << sum2 << '\n';
    return 0;
}

$ ./a.out
30000000, 59820950156796
```

accumulate() vs reduce()

C++17 - The Biggest Traps - Nicolai Josuttis

C++17: accumulate() versus reduce()

```
std::cout << 0 + 1*1 + 2*2 + 3*3 + 4*4 << '\n';      // 30

std::vector<long long> coll;
for (long long i=0; i < 10; ++i) {                      // coll: 1 2 3 4 1 2 3 4 1 2 ...
    coll.insert(coll.end(), {1, 2, 3, 4});
}

auto squaredSum = [] (auto sum, auto val) {
    return sum + val * val;
};

// use accumulate():
auto sum = std::accumulate(coll.begin(), coll.end(),
                           0LL, squaredSum);
std::cout << "accumulate(): " << sum << '\n'; // 300

// use reduce():
sum = std::reduce(std::execution::par,
                  coll.begin(), coll.end(),
                  0LL, squaredSum);
std::cout << "reduce(): " << sum << '\n'; // 300
```



©2019 by IT-communication.com

43

josuttis | eckstein
IT communication



accumulate() vs reduce()

C++17 - The Biggest Traps - Nicolai Josuttis

C++17: accumulate() versus reduce()

```
std::cout << 0 + 1*1 + 2*2 + 3*3 + 4*4 << '\n';      // 30

std::vector<long long> coll;
for (long long i=0; i < 1'000'000; ++i) {                // coll: 1 2 3 4 1 2 3 4 1 2 ...
    coll.insert(coll.end(), {1, 2, 3, 4});
}

auto squaredSum = [] (auto sum, auto val) {
    return sum + val * val;
};

// use accumulate():
auto sum = std::accumulate(coll.begin(), coll.end(),
                           0LL, squaredSum);
std::cout << "accumulate():      " << sum << '\n'; // 30,000,000

// use reduce():
sum = std::reduce(std::execution::par,
                  coll.begin(), coll.end(),
                  0LL, squaredSum);
std::cout << "reduce():        " << sum << '\n'; // 59,820,950,156,796
```



©2019 by IT-communication.com

45

josuttis | eckstein
IT communication



accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>
#include <execution>

std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; }; // not commutative

int main()
{
    for ( int i = 0; i < 1000000; ++i)
    {
        v1.insert( v1.end(), {0,1,2,3,4});
    }
    // accumulate is guaranteed left associative
    auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
    // reduce can work parallel
    auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);

    std::cout << sum1 << ", " << sum2 << '\n';
    return 0;
}

$ ./a.out
30000000, 59820950156796
```

accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>
#include <execution>
#include <functional>
std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
    for ( int i = 0; i < 1000000; ++i)
    {
        v1.insert( v1.end(), {0,1,2,3,4});
    }
    // accumulate is guaranteed left associative
    auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
    auto sum2 = std::transform_reduce(std::execution::par, // map-reduce
                                    v1.begin(), v1.end(), 0LL,
                                    std::plus<>(),
                                    [] (auto v) { return v*v; });
    std::cout << sum1 << ", " << sum2 << '\n';
    return 0;
}

$ ./a.out
30000000, 30000000
```

Concept based approach

```
int main()
{
    auto sum2 = safe_pstl::transform_reduce(std::execution::par, // map-reduce
                                            v1.begin(), v1.end(), 0LL,
                                            std::plus<>(),
                                            [](auto v) { return v*v; });
}

$ g++ accumulate.cpp
error: no matching function for call to 'transform_reduce(...)'
note: candidate 'T pstl::numeric::transform_reduce(ExecutionPolicy&&,
    ForwardIt1&&, ForwardIt1&&, T&&, ReduceOp&&, TransformOp&&)'
note: constraints not satisfied:
      required for the satisfaction of 'derived_from<
          ReduceOp, pstl::associative_tag>' [with ReduceOp = std::plus<int>]
      required for the satisfaction of 'derived_from<
          TransformOp, pstl::associative_tag>' [with TransformOp = <lambda:2>]
```

Concept based approach

```
int main()
{
    auto sum2 = safe_pstl::transform_reduce(std::execution::par, // map-reduce
                                           v1.begin(), v1.end(), 0LL, safe_pstl::wrapper{
                                               std::plus<>(),
                                               safe_pstl::associative{}, safe_pstl::commutative{} },
                                           safe_pstl::wrapper{
                                               [](auto v) { return v*v; },
                                               safe_pstl::associative{}, safe_pstl::commutative{} });
}

$ g++ accumulate.cpp
$
```

Find the first element

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <execution>

int main() {
    std::vector<int> v{100'000, 2};

    v[5001] = 123;      // ----- 1st
    v[11001] = 999;
    v[22222] = 1233;
    v[33333] = 546464;
    v[55555] = 555;
    v[77777] = 7657;
    v[77778] = 7658;
    v[77779] = 7659;
    v[88888] = 312231;

    const auto res = std::find_if( v.begin(), v.end(),
                                  [](>int i) { return i > 50; });

    std::cout << *res << '\n';
    return 0;
}
$ g++ -Wextra -std=c++17 findif_sec.cpp -p -ltbb && ./a.out
```

Find the first element

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <execution>

int main() {
    std::vector<int> v{100'000, 2};

    v[5001] = 123;      // ----- 1st
    v[11001] = 999;
    v[22222] = 1233;
    v[33333] = 546464;
    v[55555] = 555;
    v[77777] = 7657;
    v[77778] = 7658;
    v[77779] = 7659;
    v[88888] = 312231;

    const auto res = std::find_if( v.begin(), v.end(),
                                  [](>int i) { return i > 50; });

    std::cout << *res << '\n';
    return 0;
}
$ g++ -Wextra -std=c++17 findif_sec.cpp -p -ltbb && ./a.out
123
```

Find the first element

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <execution>

int main() {
    std::vector<int> v{100'000, 2};

    v[5001] = 123;      // ----- 1st
    v[11001] = 999;
    v[22222] = 1233;
    v[33333] = 546464;
    v[55555] = 555;
    v[77777] = 7657;
    v[77778] = 7658;
    v[77779] = 7659;
    v[88888] = 312231;

    const auto res = std::find_if( std::execution::par, v.begin(), v.end(),
                                  [](>int i) { return i > 50; });

    std::cout << *res << '\n';
    return 0;
}
$ g++ -Wextra -std=c++17 findif_par.cpp -p -ltbb && ./a.out
```

Find the first element

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <execution>

int main() {
    std::vector<int> v{100'000, 2};

    v[5001] = 123;      // ----- 1st
    v[11001] = 999;
    v[22222] = 1233;
    v[33333] = 546464;
    v[55555] = 555;
    v[77777] = 7657;
    v[77778] = 7658;
    v[77779] = 7659;
    v[88888] = 312231;

    const auto res = std::find_if( std::execution::par, v.begin(), v.end(),
                                  [](>int i) { return i > 50; });

    std::cout << *res << '\n';
    return 0;
}
$ g++ -Wextra -std=c++17 findif_par.cpp -p -ltbb && ./a.out
123
```

Find the 5th element

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <execution>

int main() {
    std::vector<int> v{100'000, 2};

    v[5001] = 123;
    v[11001] = 999;
    v[22222] = 1233;
    v[33333] = 546464;
    v[55555] = 555;      // <--- 5th
    v[77777] = 7657;
    v[77778] = 7658;
    v[77779] = 7659;
    v[88888] = 312231;

    const auto res = std::find_if( v.begin(), v.end(),
        [counter = 0](int i) mutable { if ( i>50 ) return ++counter == 5;
                           else           return false; });
    std::cout << *res << '\n';
    return 0;
}
$ g++ -Wextra -std=c++17 findif5_seq.cpp -p -ltbb && ./a.out
```

Find the 5th element

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <execution>

int main() {
    std::vector<int> v{100'000, 2};

    v[5001] = 123;
    v[11001] = 999;
    v[22222] = 1233;
    v[33333] = 546464;
    v[55555] = 555;      // <--- 5th
    v[77777] = 7657;
    v[77778] = 7658;
    v[77779] = 7659;
    v[88888] = 312231;

    const auto res = std::find_if( v.begin(), v.end(),
        [counter = 0](int i) mutable { if ( i>50 ) return ++counter == 5;
                           else           return false; });
    std::cout << *res << '\n';
    return 0;
}
$ g++ -Wextra -std=c++17 findif5_seq.cpp -p -ltbb && ./a.out
555
```

Find the 5th element

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <execution>

int main() {
    std::vector<int> v{100'000, 2};

    v[5001] = 123;
    v[11001] = 999;
    v[22222] = 1233;
    v[33333] = 546464;
    v[55555] = 555;      // <--- 5th
    v[77777] = 7657;
    v[77778] = 7658;
    v[77779] = 7659;
    v[88888] = 312231;

    const auto res = std::find_if( std::execution::par, v.begin(), v.end(),
        [counter = 0](int i) mutable { if ( i>50 ) return ++counter == 5;
                           else           return false; });
    std::cout << *res << '\n';
    return 0;
}
$ g++ -Wextra -std=c++17 findif5_par.cpp -p -ltbb && ./a.out
```

Find the 5th element

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <execution>

int main() {
    std::vector<int> v{100'000, 2};

    v[5001] = 123;
    v[11001] = 999;
    v[22222] = 1233;
    v[33333] = 546464;
    v[55555] = 555;      // <--- 5th
    v[77777] = 7657;
    v[77778] = 7658;
    v[77779] = 7659;
    v[88888] = 312231;

    const auto res = std::find_if( std::execution::par, v.begin(), v.end(),
        [counter = 0](int i) mutable { if ( i>50 ) return ++counter == 5;
                           else           return false; });
    std::cout << *res << '\n';
    return 0;
}
$ g++ -Wextra -std=c++17 findif5_par.cpp -p -ltbb && ./a.out
0
```

Find the 3rd element

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <execution>

int main() {
    std::vector<int> v{100'000, 2};

    v[5001] = 123;
    v[11001] = 999;
    v[22222] = 1233; // <--- 3rd
    v[33333] = 546464;
    v[55555] = 555;
    v[77777] = 7657;
    v[77778] = 7658;
    v[77779] = 7659;
    v[88888] = 312231;

    const auto res = std::find_if( std::execution::par, v.begin(), v.end(),
        [counter = 0](int i) mutable { if ( i>50 ) return ++counter == 3;
                           else           return false; });
    std::cout << *res << '\n';
    return 0;
}
$ g++ -Wextra -std=c++17 findif3_par.cpp -p -ltbb && ./a.out
```

Find the 3rd element

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <execution>

int main() {
    std::vector<int> v{100'000, 2};

    v[5001] = 123;
    v[11001] = 999;
    v[22222] = 1233;      // <--- 3rd
    v[33333] = 546464;
    v[55555] = 555;
    v[77777] = 7657;
    v[77778] = 7658;
    v[77779] = 7659;      // <--- found this
    v[88888] = 312231;

    const auto res = std::find_if( std::execution::par, v.begin(), v.end(),
        [counter = 0](int i) mutable { if ( i>50 ) return ++counter == 3;
                           else           return false; });
    std::cout << *res << '\n';
    return 0;
}
$ g++ -Wextra -std=c++17 findif3_par.cpp -p -ltbb && ./a.out
7659
```

```

#include <vector>
#include <iostream>
#include <execution>

int main() {
    std::vector<int> v{100'000, 2};

    v[5001] = 123;
    v[11001] = 999;
    v[22222] = 1233;
    v[33333] = 546464;
    v[55555] = 555;
    v[77777] = 7657;
    v[77778] = 7658;
    v[77779] = 7659;
    v[88888] = 312231;

    std::for_each( std::execution::par, v.begin(), v.end(),
                  [](int i) { if ( i > 50 ) std::cout << i << '\n'; });
    return 0;
}
$ g++ -Wextra -std=c++17 foreach_par.cpp -lpthread && ./a.out
546464
123
7657
7658
7659
555
999
312231
1233

```

Find the 3rd element

```

#include <vector>
#include <iostream>
#include <execution>
#include <syncstream>

std::osyncstream out{std::cout};
int main() {
    std::vector<int> v{100'000, 2};

    v[5001] = 123;
    v[11001] = 999;
    v[22222] = 1233;
    v[33333] = 546464;
    v[55555] = 555;
    v[77777] = 7657;
    v[77778] = 7658;
    v[77779] = 7659;
    v[88888] = 312231;

    std::for_each( std::execution::par, v.begin(), v.end(),
                  [](int i) { if ( i > 50 ) out << std::this_thread::get_id()
                                << ':' << i << '\n'; });
    return 0;
}
$ g++ -Wextra -std=c++20 foreach_par.cpp -lpthread && ./a.out
139997169759808:7657
139997169759808:7658
139997169759808:7659
139997157164608:1233
139997181261632:123
139997152966208:999
139997173958208:555
139997181261632:312231
139997161363008:546464

```

Find all elements

How many threads and slices?

```
#include <vector>
#include <iostream>
#include <execution>
#include <sstream>

int main() {
    std::vector<int> v(100'000);
    for ( int i = 0; i < std::ssize(v); ++i ) { v[i] = i; }
    std::for_each(
        std::execution::par, v.begin(), v.end(),
        [cnt = 0](int i) mutable { std::ostringstream os;
            if ( 0 == cnt++ )
            {
                os << std::this_thread::get_id() << ":" << i << '\n';
                std::cout << os.str();
            } });
    return 0;
}
$ g++ -Wextra -std=c++20 foreach_first.cpp -p -ltbb && ./a.out
140558520792896:0
140558520792896:195
140558513489472:50000
140558505092672:25000
140558500894272:12500
140558509291072:18750
140558520792896:390
140558492497472:31250
140558496695872:37500
140558513489472:50195
```

How many threads and slices?

```
#include <vector>
#include <iostream>
#include <execution>
#include <sstream>

int main() {
    std::vector<int> v(100'000);
    for ( int i = 0; i < std::ssize(v); ++i ) { v[i] = i; }
    std::for_each(
        std::execution::par, v.begin(), v.end(),
        [cnt = 0](int i) mutable { std::ostringstream os;
            if ( 0 == cnt++ )
            {
                os << std::this_thread::get_id() << ":" << i << '\n';
                std::cout << os.str();
            } });
    return 0;
}
$ g++ -Wextra -std=c++20 foreach_first.cpp -p -ltbb && ./a.out
140558520792896:0
140558520792896:195
140558513489472:50000
140558505092672:25000
140558500894272:12500
140558509291072:18750
140558520792896:390
140558492497472:31250
140558496695872:37500
140558513489472:50195
```

7 different threads
65-135 slices per each
~73 slices expected

How many threads and slices?

```
#include <vector>
#include <iostream>
#include <execution>
#include <sstream>

int main() {
    std::vector<int> v(100'000);
    for ( int i = 0; i < std::ssize(v); ++i ) { v[i] = i; }
    std::for_each(
        std::execution::par, v.begin(), v.end(),
        [cnt = 0](int i) mutable { std::ostringstream os;
            if ( 0 == cnt++ )
            {
                os << std::this_thread::get_id() << ":" << i << '\n';
                std::cout << os.str();
            } });
    return 0;
}
```

How many threads and slices?

```
#include <vector>
#include <iostream>
#include <execution>
#include <sstream>

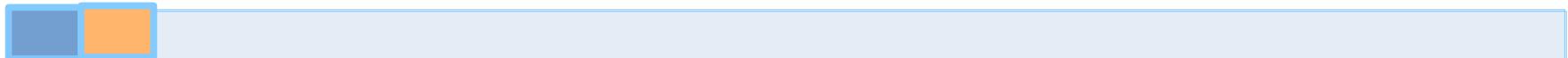
int main() {
    std::vector<int> v(100'000);
    for ( int i = 0; i < std::ssize(v); ++i ) { v[i] = i; }
    std::for_each(
        std::execution::par, v.begin(), v.end(),
        [cnt = 0](int i) mutable { std::ostringstream os;
            if ( 0 == cnt++ )
            {
                os << std::this_thread::get_id() << ":" << i << '\n';
                std::cout << os.str();
            } });
    return 0;
}
```



How many threads and slices?

```
#include <vector>
#include <iostream>
#include <execution>
#include <sstream>

int main() {
    std::vector<int> v(100'000);
    for ( int i = 0; i < std::ssize(v); ++i ) { v[i] = i; }
    std::for_each(
        std::execution::par, v.begin(), v.end(),
        [cnt = 0](int i) mutable { std::ostringstream os;
            if ( 0 == cnt++ )
            {
                os << std::this_thread::get_id() << ":" << i << '\n';
                std::cout << os.str();
            } });
    return 0;
}
```



How many threads and slices?

```
#include <vector>
#include <iostream>
#include <execution>
#include <sstream>

int main() {
    std::vector<int> v(100'000);
    for ( int i = 0; i < std::ssize(v); ++i ) { v[i] = i; }
    std::for_each(
        std::execution::par, v.begin(), v.end(),
        [cnt = 0](int i) mutable { std::ostringstream os;
            if ( 0 == cnt++ )
            {
                os << std::this_thread::get_id() << ":" << i << '\n';
                std::cout << os.str();
            } });
    return 0;
}
```



How many threads and slices?

```
#include <vector>
#include <iostream>
#include <execution>
#include <sstream>

int main() {
    std::vector<int> v(100'000);
    for ( int i = 0; i < std::ssize(v); ++i ) { v[i] = i; }
    std::for_each(
        std::execution::par, v.begin(), v.end(),
        [cnt = 0](int i) mutable { std::ostringstream os;
            if ( 0 == cnt++ )
            {
                os << std::this_thread::get_id() << ":" << i << '\n';
                std::cout << os.str();
            } });
    return 0;
}
```



How many threads and slices?

```
#include <vector>
#include <iostream>
#include <execution>
#include <sstream>

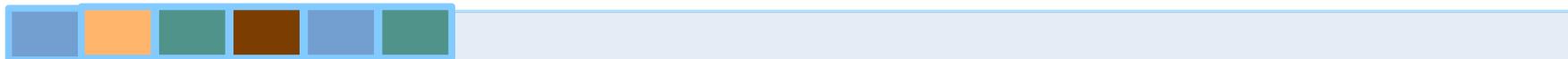
int main() {
    std::vector<int> v(100'000);
    for ( int i = 0; i < std::ssize(v); ++i ) { v[i] = i; }
    std::for_each(
        std::execution::par, v.begin(), v.end(),
        [cnt = 0](int i) mutable { std::ostringstream os;
            if ( 0 == cnt++ )
            {
                os << std::this_thread::get_id() << ":" << i << '\n';
                std::cout << os.str();
            } });
    return 0;
}
```



How many threads and slices?

```
#include <vector>
#include <iostream>
#include <execution>
#include <sstream>

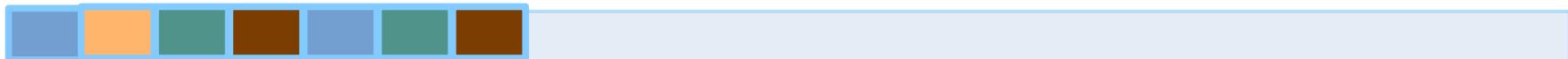
int main() {
    std::vector<int> v(100'000);
    for ( int i = 0; i < std::ssize(v); ++i ) { v[i] = i; }
    std::for_each(
        std::execution::par, v.begin(), v.end(),
        [cnt = 0](int i) mutable { std::ostringstream os;
            if ( 0 == cnt++ )
            {
                os << std::this_thread::get_id() << ":" << i << '\n';
                std::cout << os.str();
            } });
    return 0;
}
```



How many threads and slices?

```
#include <vector>
#include <iostream>
#include <execution>
#include <sstream>

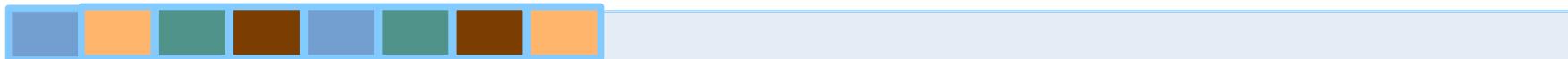
int main() {
    std::vector<int> v(100'000);
    for ( int i = 0; i < std::ssize(v); ++i ) { v[i] = i; }
    std::for_each(
        std::execution::par, v.begin(), v.end(),
        [cnt = 0](int i) mutable { std::ostringstream os;
            if ( 0 == cnt++ )
            {
                os << std::this_thread::get_id() << ":" << i << '\n';
                std::cout << os.str();
            } });
    return 0;
}
```



How many threads and slices?

```
#include <vector>
#include <iostream>
#include <execution>
#include <sstream>

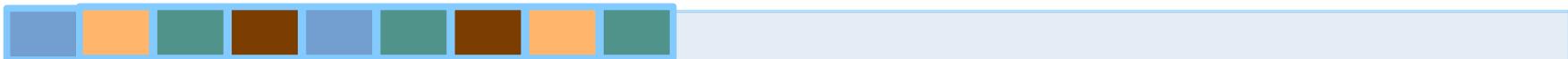
int main() {
    std::vector<int> v(100'000);
    for ( int i = 0; i < std::ssize(v); ++i ) { v[i] = i; }
    std::for_each(
        std::execution::par, v.begin(), v.end(),
        [cnt = 0](int i) mutable { std::ostringstream os;
            if ( 0 == cnt++ )
            {
                os << std::this_thread::get_id() << ":" << i << '\n';
                std::cout << os.str();
            } });
    return 0;
}
```



How many threads and slices?

```
#include <vector>
#include <iostream>
#include <execution>
#include <sstream>

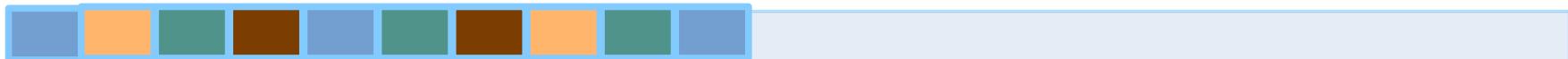
int main() {
    std::vector<int> v(100'000);
    for ( int i = 0; i < std::ssize(v); ++i ) { v[i] = i; }
    std::for_each(
        std::execution::par, v.begin(), v.end(),
        [cnt = 0](int i) mutable { std::ostringstream os;
            if ( 0 == cnt++ )
            {
                os << std::this_thread::get_id() << ":" << i << '\n';
                std::cout << os.str();
            } });
    return 0;
}
```



How many threads and slices?

```
#include <vector>
#include <iostream>
#include <execution>
#include <sstream>

int main() {
    std::vector<int> v(100'000);
    for ( int i = 0; i < std::ssize(v); ++i ) { v[i] = i; }
    std::for_each(
        std::execution::par, v.begin(), v.end(),
        [cnt = 0](int i) mutable { std::ostringstream os;
            if ( 0 == cnt++ )
            {
                os << std::this_thread::get_id() << ":" << i << '\n';
                std::cout << os.str();
            } });
    return 0;
}
```



How many threads and slices?

```
#include <vector>
#include <iostream>
#include <execution>
#include <sstream>

int main() {
    std::vector<int> v(100'000);
    for ( int i = 0; i < std::ssize(v); ++i ) { v[i] = i; }
    std::for_each(
        std::execution::par, v.begin(), v.end(),
        [cnt = 0](int i) mutable { std::ostringstream os;
            if ( 0 == cnt++ )
            {
                os << std::this_thread::get_id() << ":" << i << '\n';
                std::cout << os.str();
            } });
    return 0;
}
$ g++ -Wextra -std=c++20 foreach_first.cpp -p -ltbb && ./a.out
140558520792896:0
140558520792896:195
140558513489472:50000
140558505092672:25000
140558500894272:12500
140558509291072:18750
140558520792896:390
140558492497472:31250
140558496695872:37500
140558513489472:50195
```

7 different threads
65-135 slices per each
~73 slices expected

filter_reduce

- Split the filter and the reduce phases (similar to transform_reduce)
- Filter runs parallel, reduce sequentially
- Parallel phase implemented over Parallel STL
- The usual requirements for the parallel functor
- No requirement for the reduce functor

filter_reduce

```
#include <vector>
#include <iostream>
#include <execution>

int main() {
    std::vector<int> v{100'000, 2};

    v[5001] = 123;
    v[11001] = 999;
    v[22222] = 1233;      // <--- 3rd
    v[33333] = 546464;
    v[55555] = 555;
    v[77777] = 7657;
    v[77778] = 7658;
    v[77779] = 7659;
    v[88888] = 312231;

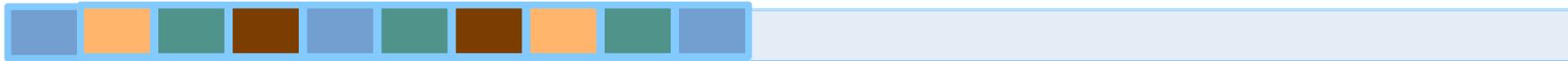
    auto res = filter_reduce( std::execution::par, v.begin(), v.end(),
        [](>int i) { return i > 50; },                                // filter, parallel
        [cnt = 0](int &mutable { return 3 == ++cnt; }); // reduce, sequential

    std::cout << *res << '\n';

    return 0;
}

$ g++ -Wextra -std=c++20 filter_reduce.cpp -lpthread && ./a.out
1233
```

filter_reduce



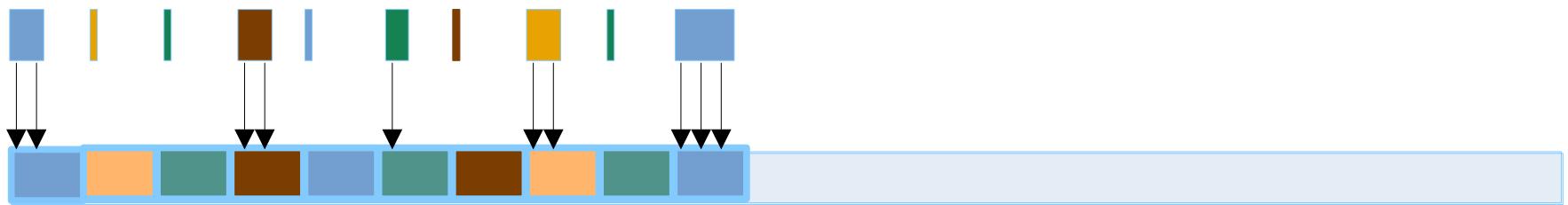
```
auto res = filter_reduce( std::execution::par, v.begin(), v.end(),
    [](int i) { return i > 50; }, // filter, parallel
    [cnt = 0](int mutable { return 3 == ++cnt; }); // reduce, sequential

std::cout << *res << '\n';

return 0;
}

$ g++ -Wextra -std=c++20 filter_reduce.cpp -lpthread && ./a.out
1233
```

filter_reduce

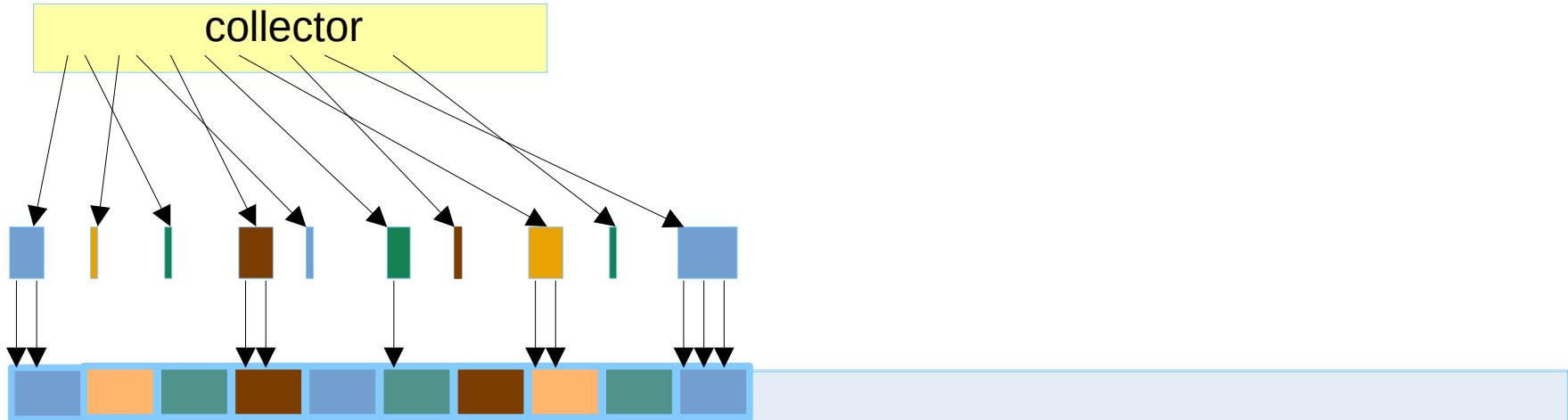


```
template <typename T>
struct Collector {

    std::map<int, std::vector<T*>> v_; // not the most optimal, but works :)

    std::mutex m_; // mutex, only to add vec, should be improved
};
```

filter_reduce

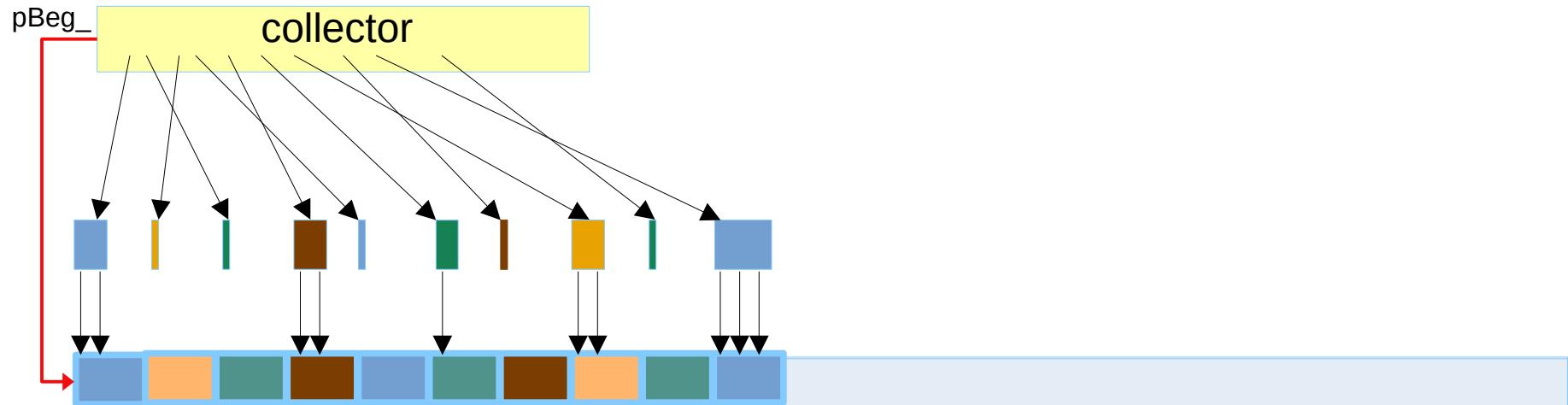


```
template <typename T>
struct Collector {

    std::map<int, std::vector<T*>> v_; // not the most optimal, but works :)

    std::mutex m_; // mutex, only to add vec, should be improved
};
```

filter_reduce



```
template <typename T>
struct Collector {
```

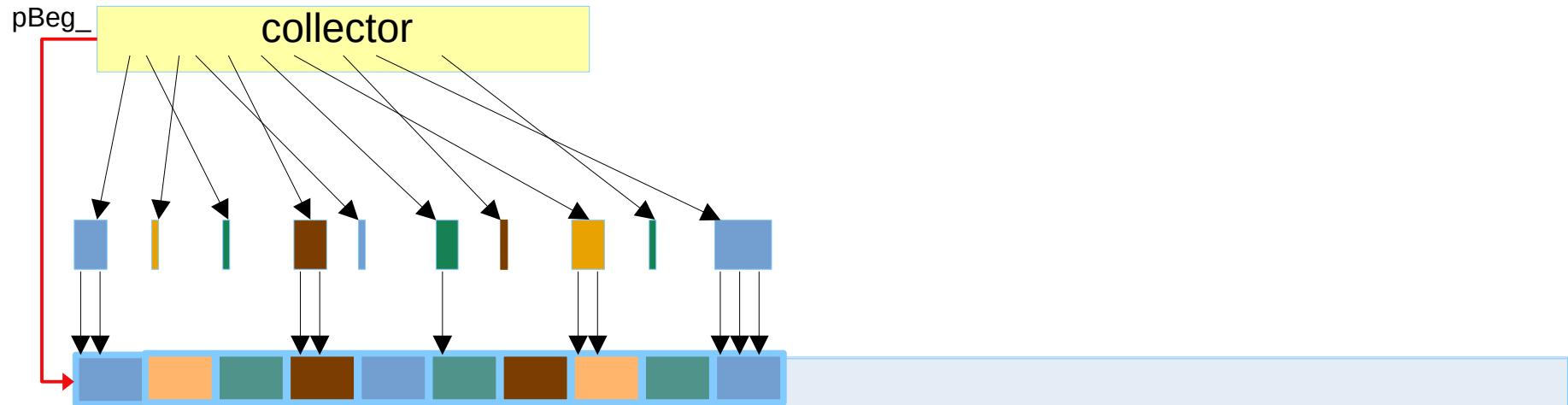
```
    std::map<int, std::vector<T*>> v_; // not the most optimal, but works :)
    T *pBeg_; // just to compute chunk index
    std::mutex m_; // mutex, only to add vec, should be improved
};
```

2024.11.27.

Z. Porkoláb: Ruminations on Parallel STL

65

filter_reduce



```
template <typename T>
struct Collector {
    template <typename Func>
    int reduce(Func fun) {           // apply fun on the filtered results
        for ( auto it = v_.begin(); it != v_.end(); ++it)           // map elements
            for ( auto jt = it->second.begin(); jt != it->second.end(); ++jt) //
                if ( fun(**jt) ) return &**jt - pBeg_; // if fun applies return the index
        return -1;                                // no element was found
    }

    std::map<int, std::vector<T*>> v_; // not the most optimal, but works :)
    T *pBeg_; // just to compute chunk index
    std::mutex m_; // mutex, only to add vec, should be improved
};
```

filter_reduce

```
template <typename T>
struct Collector {

};

template <typename Filter, typename T>
struct Func {
public:
    Func(Filter f, Collector<T>& col) : filter_(f), col_(col) { }

    void operator()(T &i) {

        if (filter_(i)) myvec_->push_back(&i); // store the hit
    }
private:
    Filter           filter_; // the filter functor given by the user
    int              cnt_ = 0; // counter to find the first element in chunk
    Collector<T>   &col_; // the collector reference
    std::vector<T*> *myvec_; // my vector to collect hits in chunk
};
```

filter_reduce

```
template <typename T>
struct Collector {

};

template <typename L, typename T>
struct Func {
public:
    Func(Filter f, Collector<T>& col) : filter_(f), col_(col) { }

    void operator()(T &i) {
        if ( 0 == cnt_++ ) { // first time in the chunk
    }
        if ( filter_(i) ) myvec_->push_back(&i); // store the hit
    }
private:
    Filter          filter_; // the filter functor given by the user
    int             cnt_ = 0; // counter to find the first element in chunk
    Collector<T>   &col_; // the collector reference
    std::vector<T*> *myvec_; // my vector to collect hits in chunk
};
```

filter_reduce

```
template <typename T>
struct Collector {

};

template <typename L, typename T>
struct Func {
public:
    Func(Filter f, Collector<T>& col) : filter_(f), col_(col)  { }

    void operator()(T &i) {
        if ( 0 == cnt_++ ) {                                // first time in the chunk
            int idx = (&i - col_.pBeg_);                  // which index is the starting?
            auto vec = std::vector<T*>{};                // local vector to store the results

            myvec_ = &col_.v_[idx];                         // keep a reference to my vector
        }
        if ( filter_(i) ) myvec_->push_back(&i); // store the hit
    }
private:
    Filter          filter_; // the filter functor given by the user
    int             cnt_ = 0; // counter to find the first element in chunk
    Collector<T>  &col_; // the collector reference
    std::vector<T*> *myvec_; // my vector to collect hits in chunk
};
```

filter_reduce

```
template <typename T>
struct Collector {

};

template <typename L, typename T>
struct Func {
public:
    Func(Filter f, Collector<T>& col) : filter_(f), col_(col)  { }

    void operator()(T &i) {
        if ( 0 == cnt_++ ) {                                // first time in the chunk
            int idx = (&i - col_.pBeg_);                  // which index is the starting?
            auto vec = std::vector<T*>{};                // local vector to store the results
            {
                std::lock_guard g{col_.m_};                 // only lock, should be improved to lock-free
                col_.v_[idx] = vec;                          // add to the collector
            }
            myvec_ = &col_.v_[idx];                         // keep a reference to my vector
        }
        if ( filter_(i) ) myvec_->push_back(&i); // store the hit
    }
private:
    Filter          filter_; // the filter functor given by the user
    int             cnt_ = 0; // counter to find the first element in chunk
    Collector<T>  &col_; // the collector reference
    std::vector<T*> *myvec_; // my vector to collect hits in chunk
};
```

filter_reduce

```
template <typename T>
struct Collector { /* ... */ };
template <typename L, typename T>
struct Func { /* ... */ }

template <class ExecutionPolicy, class ForwardIter, class UnaryPredicate, class Predicate>
ForwardIter filter_reduce(ExecutionPolicy&& exec,
                         ForwardIter first, ForwardIter last,
                         UnaryPredicate filter,
                         Predicate reduce)
{
    using value_type = typename std::iterator_traits<ForwardIter>::value_type;

    Collector<value_type> collector;      // collecting the results
    collector.pBeg_ = &*first;             // the beginning of the range

    for_each( exec, first, last, Func{filter, collector}); // collecting the hits by filter

    auto res = collector.reduce(reduce); // apply reduce on filtered hits, returning index

    return res == -1 ? last : first + res; // return iterator of found element or last
}
```

filter_reduce

```
template <typename T>
struct Collector { /* ... */ };
template <typename L, typename T>
struct Func { /* ... */ }

template <class ExecutionPolicy, class ForwardIter, class UnaryPredicate, class Predicate>
ForwardIter filter_reduce(ExecutionPolicy&& exec,
                         ForwardIter first, ForwardIter last,
                         UnaryPredicate filter,
                         Predicate reduce)
{
    using value_type = typename std::iterator_traits<ForwardIter>::value_type;

    Collector<value_type> collector;      // collecting the results
    collector.pBeg_ = &*first;             // the beginning of the range

    for_each( exec, first, last, Func{filter, collector}); // collecting the hits by filter

    auto res = collector.reduce(reduce); // apply reduce on filtered hits, returning index

    return res == -1 ? last : first + res; // return iterator of found element or last
}
```

filter_reduce

```
#include <vector>
#include <iostream>
#include <execution>

int main() {
    std::vector<int> v{100'000, 2};

    v[5001] = 123;
    v[11001] = 999;
    v[22222] = 1233;      // <--- 3rd
    v[33333] = 546464;
    v[55555] = 555;
    v[77777] = 7657;
    v[77778] = 7658;
    v[77779] = 7659;
    v[88888] = 312231;

    auto res = filter_reduce( std::execution::par, v.begin(), v.end(),
        [](>int i) { return i > 50; },                                // filter, parallel
        [cnt = 0](int &mutable { return 3 == ++cnt; }); // reduce, sequential

    std::cout << *res << '\n';

    return 0;
}

$ g++ -Wextra -std=c++20 filter_reduce.cpp -lpthread && ./a.out
1233
```

filter_reduce

- Works only for containers with continuous memory buffer
 - std::array, std::vector, arrays
- Can be extended for forward_iterators
- Reduce is not generic enough

1'00'000'000 elements

time in ms

Benchmark lock-free

Ratio elapsed seconds result

0.000010	0.040429	0.040434	0.004730	99900000	99900051	1000
0.000010	0.040642	0.040647	0.004789	99900000	99900051	1000
0.000100	0.040359	0.040365	0.006888	99990000	99990051	10000
0.001000	0.042162	0.042193	0.031105	99999000	99999051	100000
0.001000	0.043215	0.043248	0.032241	99999000	99999051	100000
0.010000	0.053399	0.053667	0.267485	99999900	99999951	1000000
0.100000	0.097772	0.100079	2.307174	99999990	100000041	10000000
0.200000	0.159012	0.163586	4.573796	99999995	100000046	20000000
0.300000	0.179375	0.186357	6.982501	89999997	90000048	30000000
0.400000	0.289613	0.298749	9.136540	79999998	80000049	40000000
0.500000	0.290685	0.302148	11.462255	99999998	100000049	50000000
0.600000	0.566464	0.580202	13.737400	59999999	60000050	60000000
0.700000	0.570236	0.586431	16.195313	69999999	70000050	70000000
0.800000	0.575968	0.594321	18.352962	79999999	80000050	80000000
0.900000	0.573081	0.593858	20.777012	89999999	90000050	90000000

Benchmark mutexed

Ratio elapsed seconds result

0.000010	0.009751	0.009810	0.058923	99900000	99900051	1000
0.000010	0.008538	0.008603	0.065091	99900000	99900051	1000
0.000100	0.008311	0.008378	0.066436	99990000	99990051	10000
0.001000	0.009259	0.009404	0.145607	99999000	99999051	100000
0.001000	0.008711	0.008837	0.126032	99999000	99999051	100000
0.010000	0.008964	0.009293	0.329476	99999900	99999951	1000000
0.100000	0.013471	0.015937	2.466194	99999990	100000041	10000000
0.200000	0.017382	0.027048	9.666264	99999995	100000046	20000000
0.300000	0.020640	0.028284	7.644565	89999997	90000048	30000000
0.400000	0.027658	0.037885	10.226768	79999998	80000049	40000000
0.500000	0.029151	0.040757	11.605569	99999998	100000049	50000000
0.600000	0.066859	0.081755	14.895939	59999999	60000050	60000000
0.700000	0.051259	0.068653	17.394319	69999999	70000050	70000000
0.800000	0.050703	0.070354	19.650847	79999999	80000050	80000000
0.900000	0.055854	0.077029	21.174872	89999999	90000050	90000000

Benchmark single threaded

Ratio elapsed seconds result

0.000010	0.032873	0.032875	0.002700	99900000	99900051	1000
0.000010	0.032785	0.032788	0.002701	99900000	99900051	1000
0.000100	0.031622	0.031627	0.005414	99990000	99990051	10000
0.001000	0.033543	0.033574	0.031235	99999000	99999051	100000
0.001000	0.033004	0.033035	0.030378	99999000	99999051	100000
0.010000	0.037681	0.037926	0.244919	99999900	99999951	1000000
0.100000	0.072888	0.075225	2.337369	99999990	100000041	10000000
0.200000	0.132098	0.136697	4.599008	99999995	100000046	20000000
0.300000	0.153541	0.160441	6.899703	89999997	90000048	30000000
0.400000	0.270973	0.280121	9.147321	79999998	80000049	40000000
0.500000	0.275510	0.286950	11.439874	99999998	100000049	50000000
0.600000	0.569561	0.583280	13.719516	59999999	60000050	60000000
0.700000	0.569290	0.585268	15.978555	69999999	70000050	70000000
0.800000	0.577139	0.595641	18.501698	79999999	80000050	80000000
0.900000	0.574319	0.595145	20.826028	89999999	90000050	90000000

1'000'000'000 elements

time in ms

Benchmark lock-free

Ratio elapsed seconds result

0.000010	0.410305	0.410313	0.007571	999900000	999900051	10000
0.000010	0.410798	0.410805	0.006954	999900000	999900051	10000
0.000100	0.406982	0.407013	0.030324	999990000	999990051	100000
0.001000	0.420554	0.420803	0.248893	999999000	999999051	1000000
0.001000	0.417179	0.417441	0.261809	999999000	999999051	1000000
0.010000	0.552152	0.554476	2.324435	999999900	999999951	10000000
0.100000	0.929400	0.952340	22.940003	999999990	1000000041	100000000
0.200000	1.409089	1.454802	45.713438	999999995	1000000046	200000000
0.300000	2.281718	2.351000	69.281886	899999997	900000048	300000000
0.400000	2.634362	2.726819	92.457800	799999998	800000049	400000000
0.500000	2.642445	2.764940	122.495053	999999998	1000000049	500000000
0.600000	5.176870	5.316112	139.242238	599999999	600000050	600000000
0.700000	9.411428	9.583475	172.046500	699999999	700000050	700000000
0.800000	10.556847	10.742946	186.098799	799999999	800000050	800000000

Killed

Benchmark mutexed

Ratio elapsed seconds result

0.000010	0.078728	0.078812	0.084204	999900000	999900051	10000
0.000010	0.075475	0.075570	0.095663	999900000	999900051	10000
0.000100	0.077426	0.077627	0.201008	999990000	999990051	100000
0.001000	0.078314	0.079052	0.738013	999999000	999999051	1000000
0.001000	0.076072	0.076819	0.747302	999999000	999999051	1000000
0.010000	0.087837	0.090486	2.649196	999999900	999999951	10000000
0.100000	0.142796	0.168120	25.323559	999999990	1000000041	100000000
0.200000	0.213548	0.260901	47.353061	999999995	1000000046	200000000
0.300000	0.336774	0.407658	70.883659	899999997	900000048	300000000
0.400000	0.375917	0.468262	92.344866	799999998	800000049	400000000
0.500000	0.374626	0.490772	116.145799	999999998	1000000049	500000000
0.600000	0.678782	0.834271	155.489315	599999999	600000050	600000000
0.700000	0.685583	0.846439	160.855516	699999999	700000050	700000000
0.800000	0.677619	0.861673	184.053998	799999999	800000050	800000000
0.900000	0.674465	0.880807	206.341885	899999999	900000050	900000000

Benchmark single threaded

Ratio elapsed seconds result

0.000010	0.322050	0.322057	0.006784	999900000	999900051	10000
0.000010	0.322568	0.322574	0.005316	999900000	999900051	10000
0.000100	0.322555	0.322614	0.058409	999990000	999990051	100000
0.001000	0.321247	0.321534	0.286769	999999000	999999051	1000000
0.001000	0.322276	0.322559	0.282876	999999000	999999051	1000000
0.010000	0.411580	0.413943	2.363187	999999900	999999951	10000000
0.100000	0.783866	0.807069	23.202867	999999990	1000000041	100000000
0.200000	1.296257	1.343488	47.230980	999999995	1000000046	200000000
0.300000	2.185831	2.254393	68.561366	899999997	900000048	300000000
0.400000	2.547480	2.639559	92.078583	799999998	800000049	400000000
0.500000	2.559596	2.678317	118.721160	999999998	1000000049	500000000
0.600000	5.219504	5.357767	138.263093	599999999	600000050	600000000
0.700000	5.277819	5.442415	164.596681	699999999	700000050	700000000
0.800000	5.252166	5.439961	187.794276	799999999	800000050	800000000
0.900000	5.285139	5.491427	206.287972	899999999	900000050	900000000

Questions?