Vlaanderen
is supercomputing

Shared memory programming in C++ with Threading Building Blocks

Geert Jan Bex
email: geertjan.bex@uhasselt.be

Introduction & motivation

Why shared memory programming?

- All CPUs have multiple cores
- Cores have vector registers
- Accelerators
  - GPGPUs
  - FPGAs
  - …

Various levels of parallelism

Options for C++?

- pthreads
  - disadvantages: programming model not really suited for scientic programming
- C++ threads
  - advantages: part of C++ standard
  - disadvantages: programming model not really suited for scientic programming
- OpenMP
  - advantages: de facto standard in scientific computing; standard supported by "all" compilers; standard for C and Fortran
  - disadvantages: multilevel parallelization is tricky
- Threading Building Blocks (TBB)
  - advantages: multilevel parallelism is easier; integrates well with modern C++
  - disadvantages: no standard, C++ only

Whence TBB?

- History
  - 1995-2006: MIT Cilk
  - 2006-2009: Cilk++
  - 2009-...: Intel Cilk Plus
  - 2006-...: Intel Threading Building Blocks
    - optimized for Intel hardware
  - 2016-...: Threading building blocks open source
    - https://www.threadingbuildingblocks.org/

What is TBB?

- Algorithms
  - `parallel_for`, `parallel_reduce`, `parallel_scan`, `parallel_do`, `pipeline`
- Tasks & flow graphs
- Concurrent containers
  - `concurrent_queue`, `concurrent_vector`, `concurrent_hash_map`
- Mutual exclusion
- Atomic operations
- Memory allocation

Where to find info?

- TBB website:
  https://www.threadingbuildingblocks.org/
- Documentation: see website
- Book:
  Intel Threading Building Blocks: outfitting C++ for multi-core parallelism
  James Reinders, 2010, O'Reilly
- Slides
  https://github.com/gjbex/training-material/blob/master/CPlusPlus/Tbb/tbb.pptx
- Sample code
  https://github.com/gjbex/training-material/tree/master/CPlusPlus/Tbb

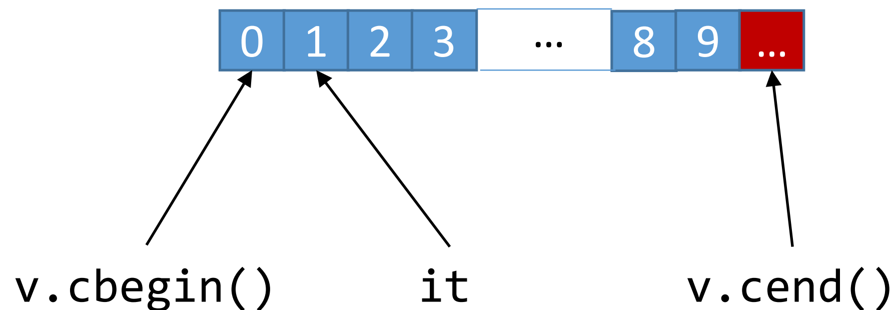
C++ refresher

C++ prerequisites

- Iterators
- Classes/structs defining `operator()`
- Lambda functions
- Placement new

Iterators

- Iterators over C++ STL containers, e.g.,
  - cbegin: iterator to first element
  - cend: iterator to last element
  - it: iterator pointing to current value (const)
  - non-const iterators: begin/end




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

int main() {
    std::vector<int> v;
    for (int i = 0; i < 10; ++i)
        v.push_back(i);

    for (auto it = v.cbegin(); it != v.cend(); ++it)
        std::cout << *it << " ";
    std::cout << std::endl;

    for (auto it = v.begin(); it != v.end(); ++it)
        *it = (*it)*(*it);
    …
    return 0;
}
```

Classes/structs defining operator()

$f_1 : \mathrm{x} \rightarrow\ x^2 - 1$

$f_2 : \mathrm{x} \rightarrow\ -x^2 + 3x$

```
#include <iostream>

class QuadraticFunction {
    private:
        double a_, b_, c_;
    public:
        QuadraticFunction(double a, double b, double c) :
            a_ {a}, b_ {b}, c_ {c} {}
        double operator()(const double x) {return (a_*x + b_)*x + c_; }
};

int main() {
    QuadraticFunction f1(1.0, 0.0, -1.0);
    QuadraticFunction f2(-1.0, 3.0, 0.0);
    for (double x = -3.0; x <= 3.0; x += 0.2)
        std::cout << x << " " << f1(x) << " " << f2(x) << "\n";
    return 0;
}
```

Lambda functions

- `std::for_each` modifies container element in-place
- Here, $x \to x^2$
- Lambda function = unnamed function, used once
- Return type: often deduced
- General form:

```
[ … ] ( … ) -> return_type { … }
```

context — arguments — function body

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

int main() {
    std::vector<double> v {0.5, 0.75, 1.0, 1.5, 3.5};
    std::for_each(v.begin(), v.end(),
                  [] (double& x) { x *= x; });
    for (auto x: v)
        std::cout << x << " ";
    std::cout << std::endl;
    return 0;
}
```

Lambda functions & context

- [ ]: nothing
- [=]: everything in scope by value (copy)
- [&]: everything in scope by reference
- [=var]: variable var by value (copy)
- [&var]: variable var by reference

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

int main() {
    std::vector<double> x_vals;
    for (double x = -3.0; x <= 3.0; x += 0.2)
        x_vals.push_back(x);
    double a {-11.0};
    double b {3.0};
    double c {1.5};
    std::vector<double> y_vals;
    std::transform(x_vals.begin(), x_vals.end(),
                   std::back_inserter(y_vals),
                   [&a, &b, &c] (double& x) {
                       return (a*x + b)*x + c;});
    for (std::size_t i = 0; i < x_vals.size(); ++i)
        std::cout << x_vals[i] << " " << y_vals[i] << "\n";
    return 0;
}
```

Placement new

- Create new object in pre-allocated memory location
- E.g., custom memory allocator

```
#include <cmath>
#include <iostream>

struct Point {
    double x, y;
    Point() {…}
};

int main() {
    Point* p_ptr = (Point*) malloc(sizeof(Point));
    double dist {0.0};
    for (long i = 0; i < i_max; ++i) {
        Point* p = new(p_ptr) Point();
        dist += std::sqrt(p->x*p->x + p->y*p->y);
    }
    free(p_ptr);
    return 0;
}
```

TBB algorithms

parallel_for: simplest form

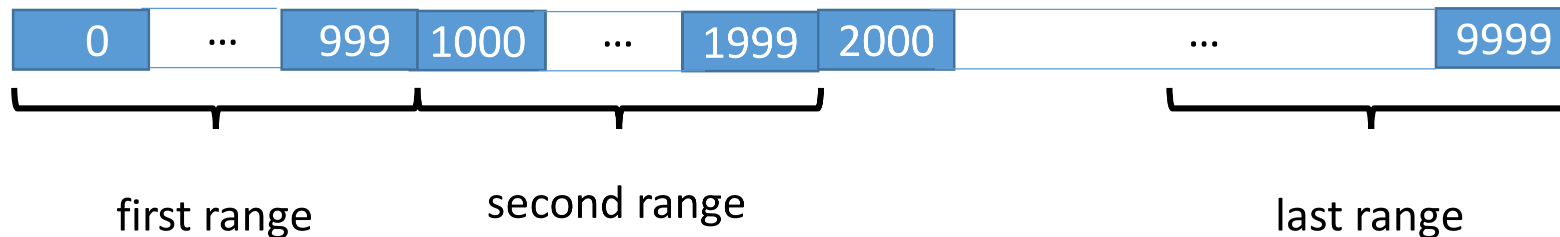
```
#include <algorithm>
#include <iostream>
#include <numeric>
#include <tbb/tbb.h>
#include <valarray>

int main() {
    std::valarray<int> data(10000);
    std::iota(std::begin(data), std::end(data), 0);

    std::for_each(std::begin(data), std::end(data),
                  [] (int& i) { i = i*i; });

    tbb::parallel_for(0ul, data.size(),
                      [&data] (std::size_t i) { data[i] = data[i]*data[i]; });

    std::cout << data.sum() << std::endl;
    return 0;
}
```

work automatically divided over thread pool

parallel_for: more control

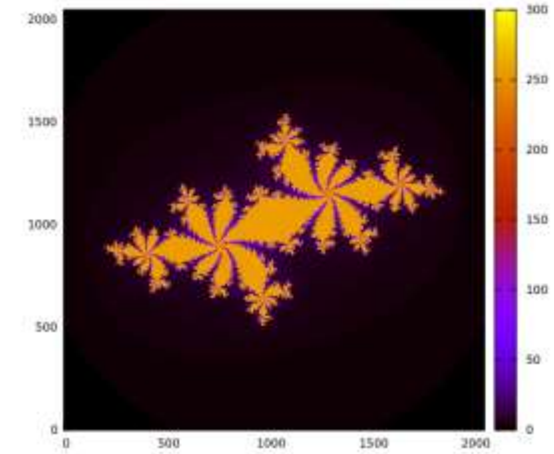
```
…

int main() {
    …
    std::size_t grain_size {1000};

    tbb::parallel_for(tbb::blocked_range<std::size_t>(0ul, data.size(), grain_size),
                      [&data] (const tbb:blocked_range<std::size_t>& range) {
                          for (std::size_t i = range.begin(); i < range.end(); ++i)
                              data[i] = data[i]*data[i]; });


    …
    return 0;
}
```

blocked ranges divided over thread pool, rule of thumb: 10,000 cycles per blocked range

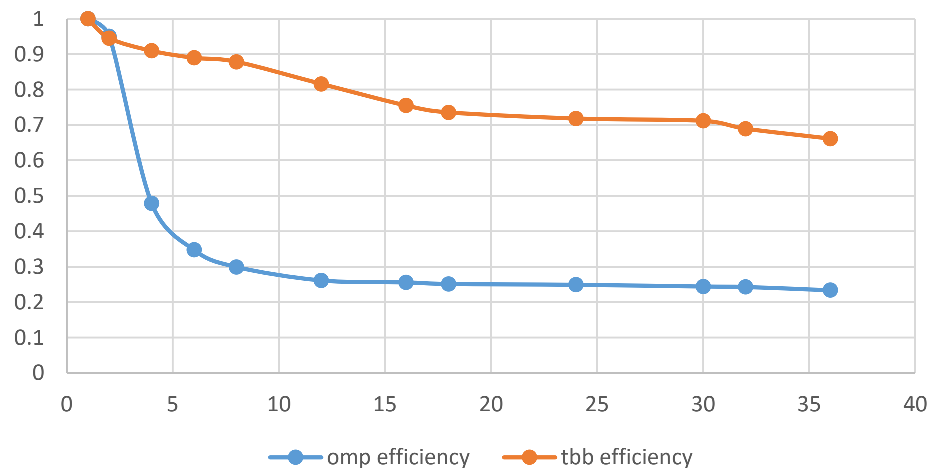


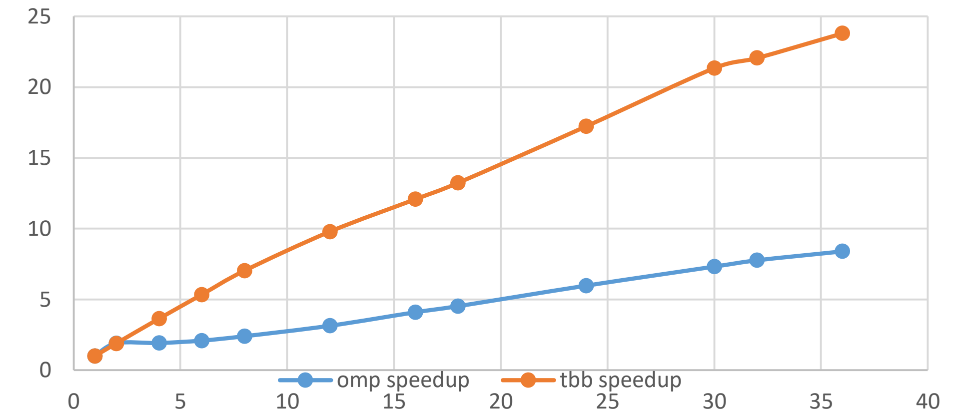
parallel_for: Julia sets

```
#include <complex>
…
const std::complex<double> c(-0.62772, - 0.42193);
for (double x = -1.8; x < 1.8; x += 0.001)
    for (double y = -1.8; y < 1.8; y += 0.001) {
        std::complex<double> z(x, y);
        while (std::abs(z) < 2.0 && n++ < max_n)
            z = z*z + c;
        std::cout << x << " " << y << " " << n << std::endl;
    }
```



16384×16384 array





parallel_reduce: simplest form

```
#include <iostream>
#include <numeric>
#include <tbb/tbb.h>
#include <valarray>

int main() {
    std::size_t grain_size {1000};
    std::valarray<int> data(10000);
    std::iota(std::begin(data), std::end(data), 0);
    int sum = tbb::parallel_reduce(
            tbb::blocked_range<std::size_t>(0ul, data.size(), grain_size), 0,
            [&data] (const tbb::blocked_range<std::size_t>& range, int init) {
                for (std::size_t i = range.begin(); i < range.end(); ++i)
                    init += data[i];
                return init;
            },
            [] (int x, int y) { return x + y; }
    );
    …
    return 0;
}
```

parallel_reduce: class

```
class Stats {
    private:
        Vector* const data_;
        const std::size_t n_;
        double sum_, sum2_;
public:
        Stats(Vector* data) : data_ {data}, n_ {data->size()}, sum_ {0.0}, sum2_ {0.0} {}
        Stats(Stats& stats, tbb::split) : Stats(stats.data_) {}
        void operator()(const tbb::blocked_range<std::size_t>& r) {
            for (auto i = r.begin(); i != r.end(); ++i) {
                double val = (*data_)[i];
                sum_ += val;
                sum2_ += val*val; }
        }
        void join(const Stats& stats) {
             sum_ += stats.sum_;
             sum2_ += stats.sum2_;
        }
        double mean() const { return sum_/n_; }
        double stddev() const {…};
};
```

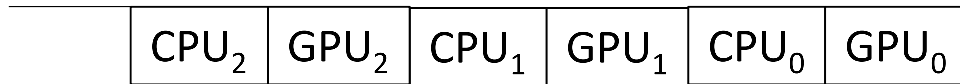
```
…
Vector data = …;
Stats stats(&data);
tbb::blocked_range<std::size_t> ranges(
    0ul, data.size(), grain_size
);
tbb::parallel_reduce(ranges, stats);
std::cout << stats.mean();
…
```

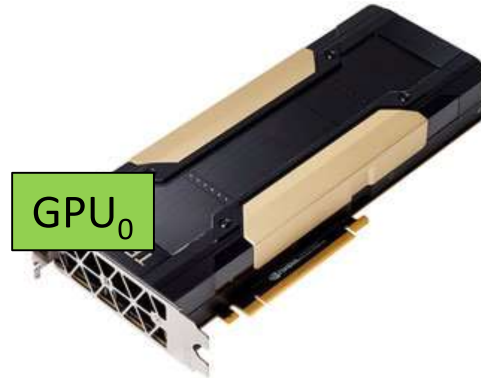
parallel_do

- Create processing class
  - overloads `operator()` to process single item
  - optionally add new items to process (using `tbb::parallel_do_feeder`)
- Create STL std::vector with initial items
- Call `tbb::parallel_do` with
  - begin of iterator for initial items to process
  - end of iterator for items to process
  - processing class instance

Note: items in the vector *can* be computed, *no* dependencies on other items

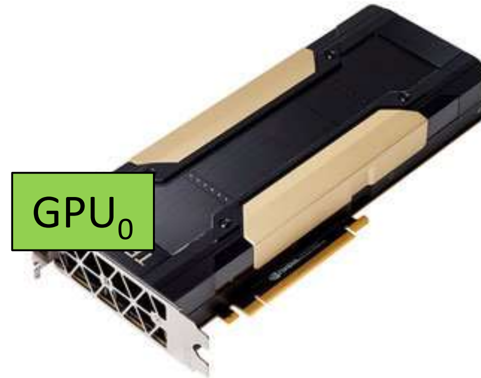
no pipeline

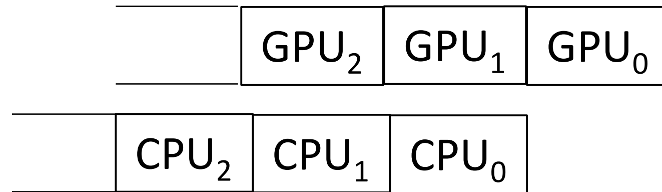




Both GPU and CPU are idle half of the time

pipeline





Pipeline start-up
→ pipeline full
→ pipeline wind-down

TBB tasks

Task spawning

$$\mathrm{fib}(n) = \mathrm{fib}(n-1) + \mathrm{fib}(n-2)$$

$$\mathrm{fib}(0) = \mathrm{fib}(1) = 1$$

```
class FibTask : public tbb::task {
    private:
        long n_;
        long* result_;
public:
        FibTask(long n, long* result) : n_ {n}, result_ {result} {}
        tbb::task* execute() {
            if (n_ < 2) {
                *result_ = 1;
            } else {
                long result_n_1;
                long result_n_2;
                tbb::task_list list;
                list.push_back(*new(allocate_child()) FibTask(n_ - 1, &result_n_1));
                list.push_back(*new(allocate_child()) FibTask(n_ - 2, &result_n_2));
                set_ref_count(3);
                spawn_and_wait_for_all(list);
                *result_ = result_n_1 + result_n_2;
            }
            return nullptr;
        }
};
```

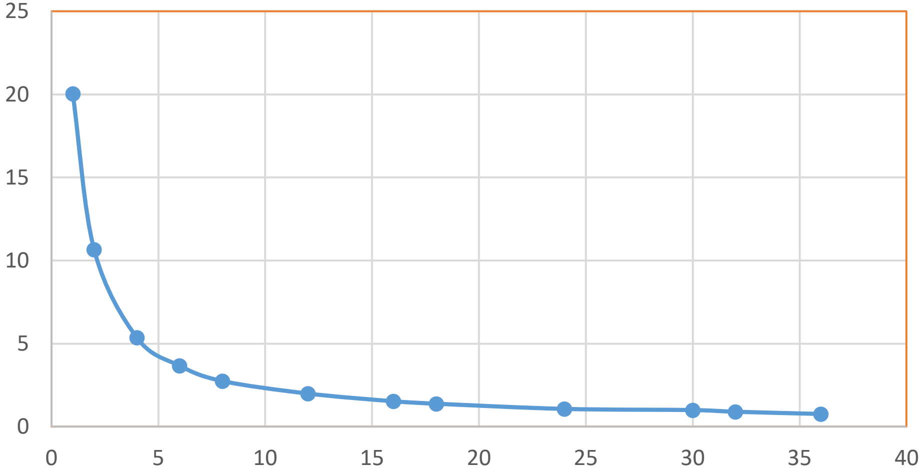
```
…
auto task = new(tbb::task::allocate_root()) FibTask(n, &result);
tbb::task::spawn_root_and_wait(*task);
…
```

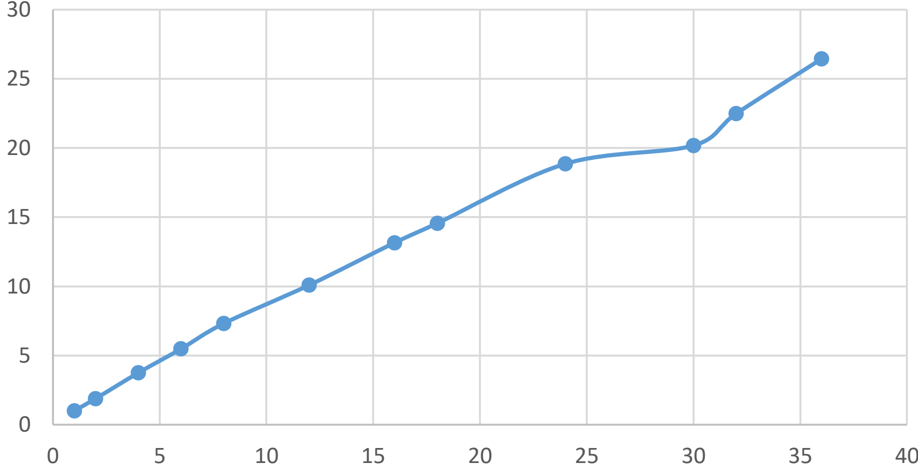
However, task granularity!

Task spawning: granularity

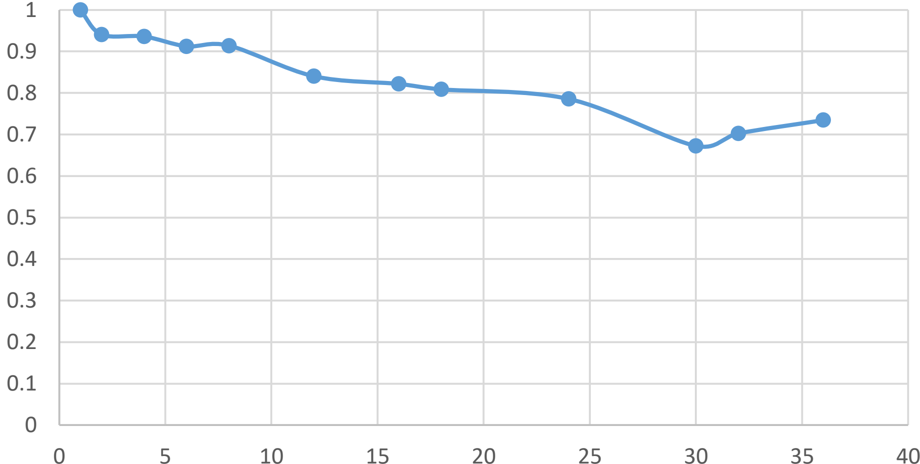
```
class FibTask : public tbb::task {
    private:
        long n_;
        long* result_;
public:
        FibTask(long n, long* result) : n_ {n}, result_ {result} {}
        tbb::task* execute() {
            if (n_ < 10) {
                *result_ = sequential_fib(n_);
            } else {
                …
            }
        }
};
```

Task spawning: timings

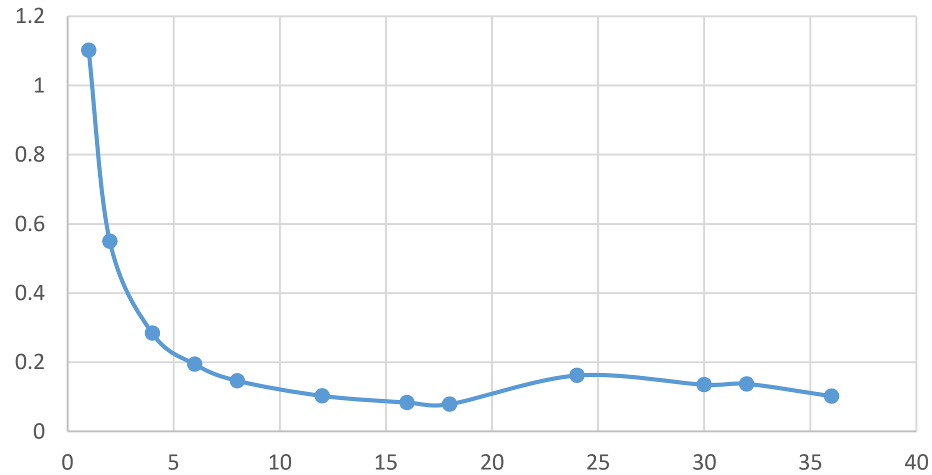


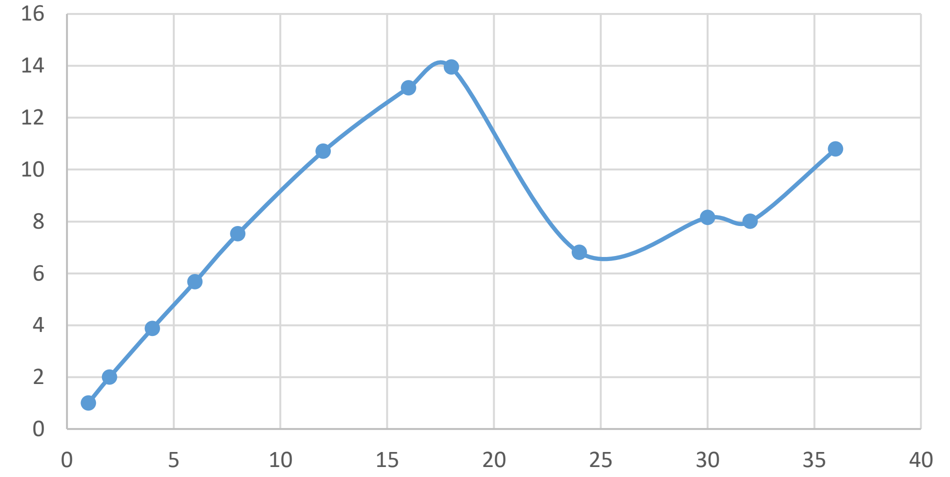


$n = 50$

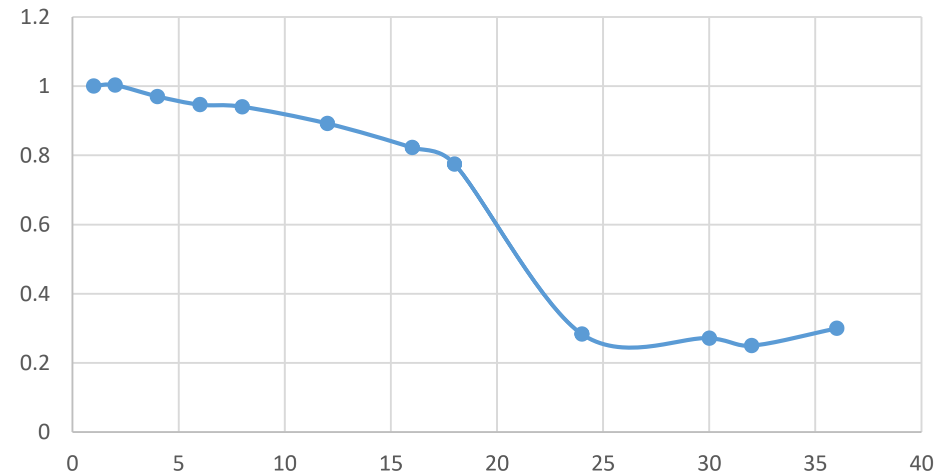


Tree traversal





54,899,183 nodes



dual Intel Xeon skylake
2×18 cores

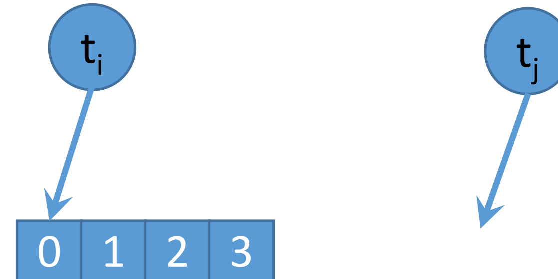
Flow graphs

TBB scheduler

TBB task scheduler

- thread pool
- task queue
- when thread is done, it steals work

- Load balancing: thread on busy core will poll less often for work



Conclusions

Conclusions

- Advantages
  - Interesting programming model
  - Integrates nicely with modern C++
  - Works with any compiler
  - Nice for heterogeneous hardware
- Disadvantages
  - Thread affinity?
  - Code base critically depends on TBB