Parallelization of C++ code
easy or not?

SIOUX TECHNOLOGIES

Print version
(animations removed)

high tech institute

Who is Klaas van Gend?

Age: 51

Lives near Eindhoven, NL

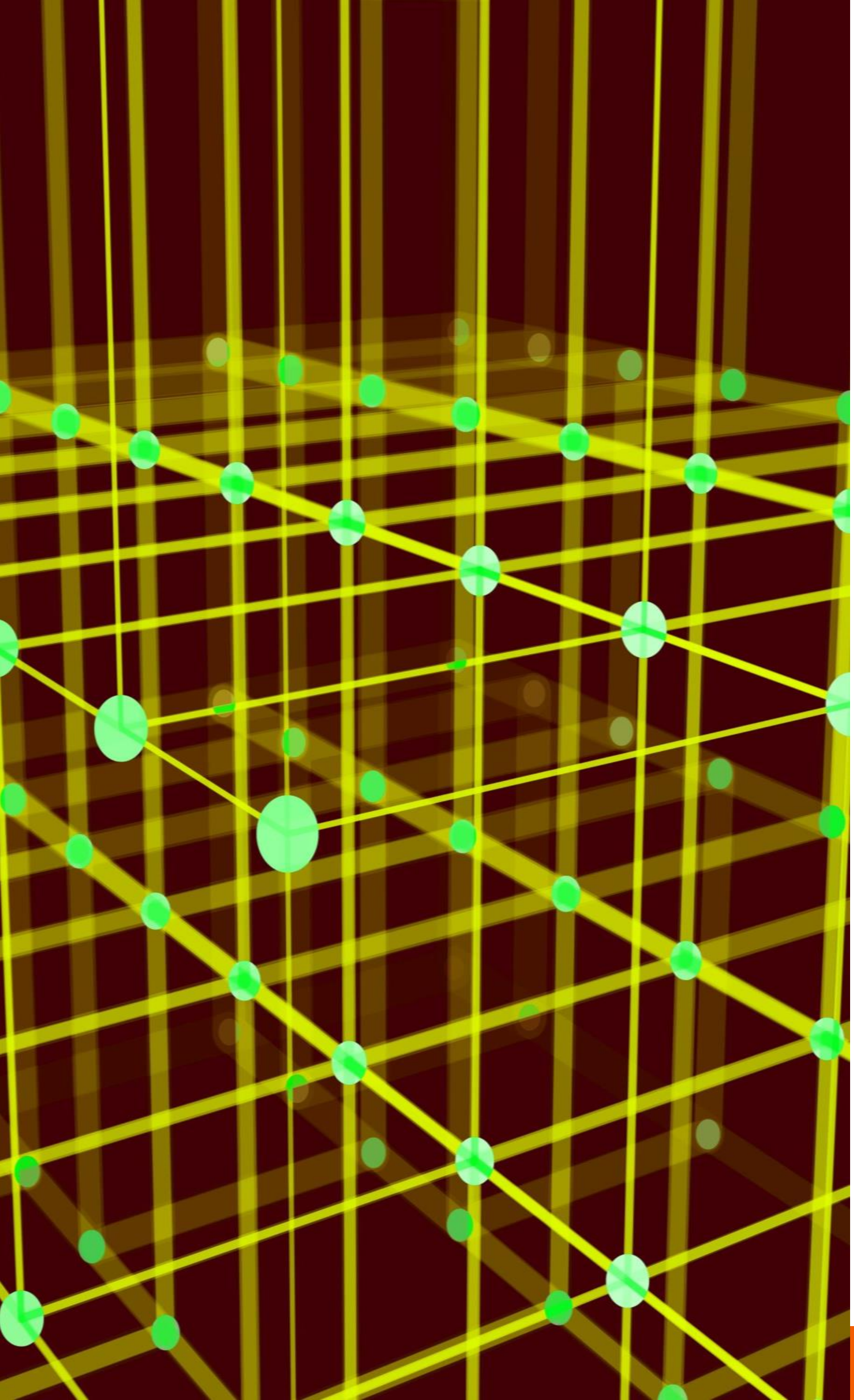
- C experience since 1990
- C++ experience since 1997
- NLUUG honorary member
- Currently working with a Swiss customer

SIOUX TECHNOLOGIES

Senior Software Architect

high tech institute

Trainer Multicore Programming in C++



- C++20 example
- C++ algorithms incl parallel options
- Theodorescu’s critique on
  C++ parallel algorithms
- Plan your concurrency: Parallel Patterns
- Enforcing patterns through libraries

Refresher:
C++11 up to C++20

Parallelism and Concurrency only

C++11 `std::thread`

`std::thread`

- Must be *`join()`*ed
- **`join()`** will wait for thread completion
- Thread destructor **`abort()`**s if not joined

```cpp
void worker(int number) {
    std::this_thread::sleep_for(std::chrono::seconds(number));
    std::cout << number << " ";
}

int main(int argc, char* argv[]) {
    std::vector<std::thread> threads;

    for (int i = 1; i < argc; ++i)
        threads.emplace_back(worker, std::stoi(argv[i]));

    for (auto& t : threads)
        t.join();

    std::cout << std::endl;
}
```

After 20 sec,
Press ctrl-C

MINGW64:/c/dev/CPP20/out/build/x64-debug

```
gendk@HONFWT3 MINGW64 ~
$ cd /c/dev/CPP20/out/build/x64-debug/

gendk@HONFWT3 MINGW64 /c/dev/CPP20/out/build/x64-debug
$ ./CPP11 1 5 12 2 1 6 3 684
1 1 2 3 5 6 12

gendk@HONFWT3 MINGW64 /c/dev/CPP20/out/build/x64-debug
$ 
```

C++20 Features

std::jthread with **`std::stop_token`**
Upon destruction, will raise the stop_token
`std::condition_variable_any` honors stop token
Thus thread ends nicely

```cpp
std::mutex m;
std::condition_variable_any cv;

void worker(std::stop_token tok, int number) {
    std::unique_lock<std::mutex> lk(m);
    cv.wait_for(lk, tok, std::chrono::seconds(number), []() {return false; });
    std::cout << number << " ";
}

int main(int argc, char* argv[]) {
    std::vector<std::jthread> threads;
    for (int i = 1; i < argc; ++i)
        threads.emplace_back(worker, std::stoi(argv[i]));

    std::this_thread::sleep_for(std::chrono::seconds(4));
    std::cout << std::endl;
}
```

MINGW64:/c/dev/CPP20/out/build/x64-debug

```
gendk@HONFWT3 MINGW64 ~
$ cd /c/dev/CPP20/out/build/x64-debug/

gendk@HONFWT3 MINGW64 /c/dev/CPP20/out/build/x64-debug
$ ./CPP11 1 5 12 2 1 6 3 684
1 1 2 3 5 6 12

gendk@HONFWT3 MINGW64 /c/dev/CPP20/out/build/x64-debug
$ ./CPP20 1 5 12 2 1 6 3 684
1 1 2 3
5 12 6 684
gendk@HONFWT3 MINGW64 /c/dev/CPP20/out/build/x64-debug
$
```

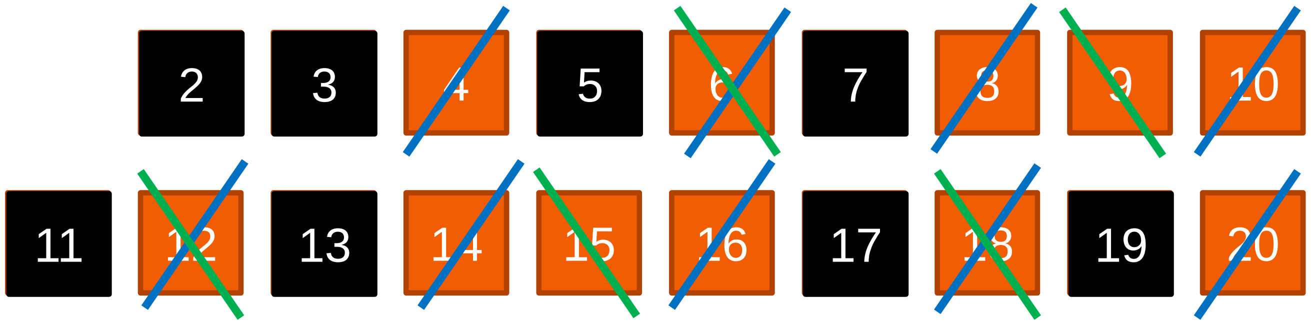
After 4 sec, application ends, all remaining values printed.

C++17’s parallel algorithms

Free speedups?

C++ Algorithms – sieve of Eratosthenes

Source: Wikipedia, public domain



- Dividable by 2
- Dividable by 3
- ~~Dividable by 4~~
- ~~Dividable by 5~~

Only need to do established primes

Only need to go to floor(sqrt(max))

C++ Algorithms – naïve sieve code

```
std::list<int> primes(int max)
{
    std::list<int> candidates;
    for (int i=1; i < max; ++i)
        candidates.push_back(i);

    for (int div=2; div < sqrt(max); ++div)
    {
        auto iter = candidates.begin();
        while (iter!=candidates.end())
        {
            auto citer = iter;
            ++iter;
            if (*citer%div==0 && *citer!=div)
                candidates.erase(citer);
        }
    }
    return candidates;
}
```

For 1000k sieve: 17.8 s (!)

```
std::vector<int> primes(int max)
{
    std::vector<int> candidates (max);
    for (int i = 0; i < max; ++i)
        candidates[i]=i;

    int highest = candidates.size();
    for (int div = 2; div < sqrt(candidates.size()); ++div) {
        int t = 2;
        for (int i = 2; i < highest; i++) {
            int c = candidates[i];
            if (c == div || c % div != 0) {
                candidates[t] = c;
                t++;
            }
        }
        highest = t;
    }
    candidates.resize(highest);
    return candidates;
}
```

50x faster:

- Using vector
  - no allocations, no jump-over-links
- Much harder to read & understand

For 1000k sieve: 356 ms

C++ Algorithms – sieve code + std algorithms

```
std::list<int> primes(int max)
{
    std::list<int> candidates;
    for (int i=1; i < max; ++i)
        candidates.push_back(i);

    for (int div=2; div < sqrt(max); ++div)
    {
        auto iter = candidates.begin();
        while (iter!=candidates.end())
        {
            auto citer = iter;
            ++iter;
            if (*citer%div==0 && *citer!=div)
                candidates.erase(citer);
        }
    }
    return candidates;
}
```

For 1000k sieve: 17.8 s

```
std::vector<int> primes(int max)
{
    std::vector<int> candidates(max);
    auto last = candidates.end();
    std::iota(candidates.begin(), last, 1);

    for (int div = 2; div < sqrt(max); ++div)
    {
        last =
            std::remove_if(candidates.begin(), last,
                [&div](int i) { return i % div == 0 && i != div; });
    }
    return std::vector<int>(candidates.begin(), last);
}
```

For 1000k sieve: 165 ms

108x

Sieve – further optimizations?

- Code also tests 4, 6, 8, 9, etc…
  - We can win quite some iterations!
  - Benefit:
    1000k elements: 41 ms

434x

- C++17's std::execution::par ???

```
std::vector<int> primesAlgo2(auto exec, int max)
{
    std::vector<int> candidates(max);
    auto last = candidates.end();
    std::iota(candidates.begin(), last, 1);

    for (int i=1; candidates[i] < sqrt(max); i++)
    {
        auto div = candidates[i];
        last =
            std::remove_if(exec,
                candidates.begin(), last,
                [&div](int i) { return i % div == 0 && i != div; });
    }
    return std::vector<int>(candidates.begin(), last);
}
```

std::execution::par_unseq and friends

```
std::vector<int> data {9, 3, 6, 4, 2, 6, 8, 1, 1};

std::sort( data.begin(), data.end());

std::sort(std::execution::seq,       data.begin(), data.end());
std::sort(std::execution::par,       data.begin(), data.end());
std::sort(std::execution::unseq,     data.begin(), data.end());
std::sort(std::execution::par_unseq, data.begin(), data.end());
```

Never run concurrently

Can be parallelized

Can be run out of order

Also known as “embarrassingly parallel”

693x

Improvements so far

approach	time	speedup
Std::list with erase()	17800 ms	1x
Std::vector with copies	356 ms	50x
Use std algorithms	165 ms	108x
Only test for primes	41 ms	434x
Use concurrent algorithms	25.7 ms	693x

Wait… Using 12 cores instead of 1.
But only 1.6x faster?

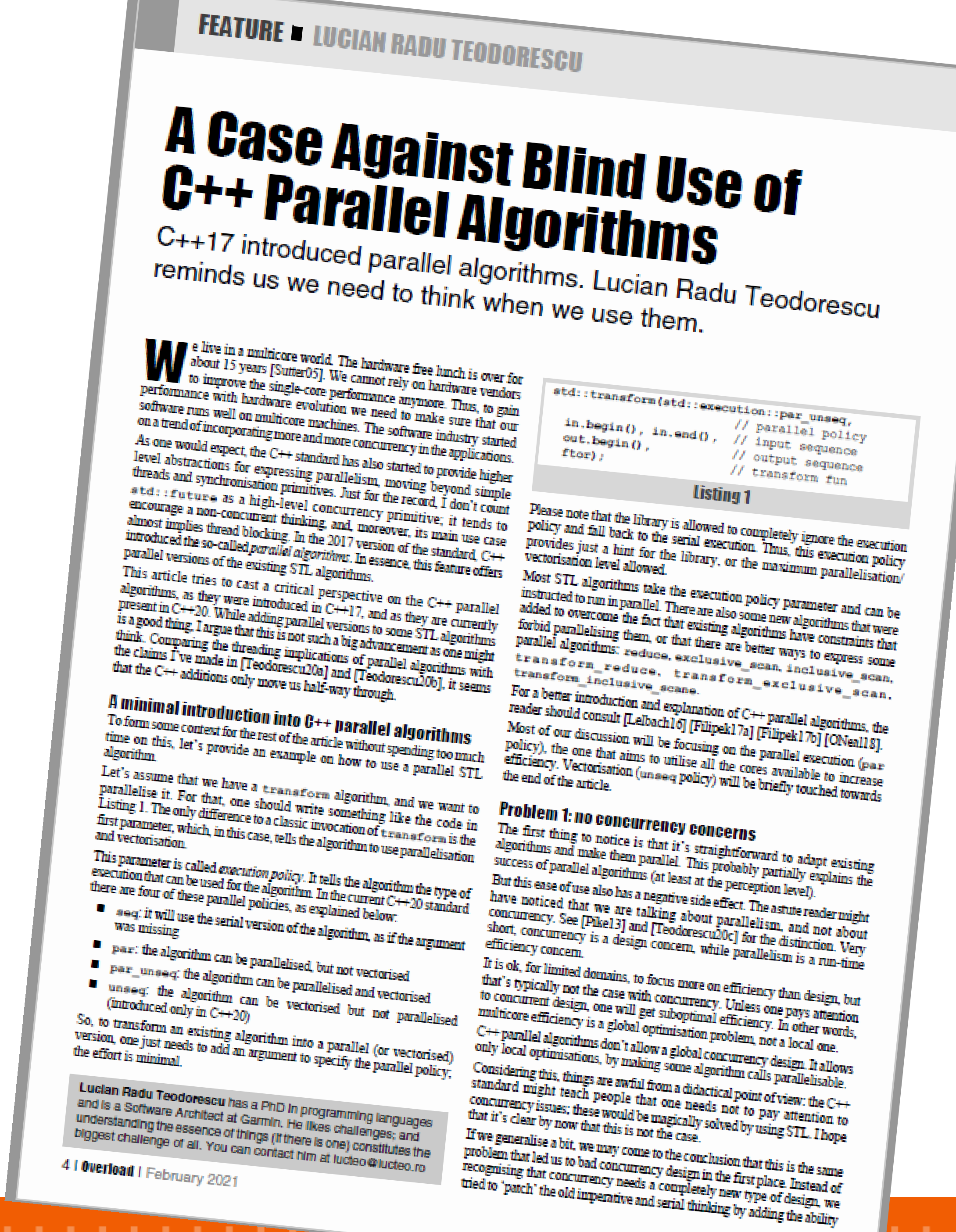
Is C++17 concurrency worth it?

Critique on C++17's `std::execution::par`

Paraphrasing Lucian Teodorescu

Critique

- By Lucian Radu Teodorescu
- Overload #161, February 2021
  - 1 of 2 magazines of ACCU
    - Association for Programmers
    - Famous for the ACCU conference
    - Europe’s premier C++ conference



FEATURE ■ LUCIAN RADU TEODORESCU

A Case Against Blind Use of C++ Parallel Algorithms

C++17 introduced parallel algorithms. Lucian Radu Teodorescu reminds us we need to think when we use them.

We live in a multicore world. The hardware free lunch is over for about 15 years [Sutter05]. We cannot rely on hardware vendors to improve the single-core performance anymore. Thus, to gain performance with hardware evolution we need to make sure that our software runs well on multicore machines. The software industry started on a trend of incorporating more and more concurrency in the applications.

As one would expect, the C++ standard has also started to provide higher level abstractions for expressing parallelism, moving beyond simple threads and synchronisation primitives. Just for the record, I don't count `std::future` as a high-level concurrency primitive; it tends to encourage a non-concurrent thinking, and, moreover, its main use case almost implies thread blocking. In the 2017 version of the standard, C++ introduced the so-called *parallel algorithms*. In essence, this feature offers parallel versions of the existing STL algorithms.

This article tries to cast a critical perspective on the C++ parallel algorithms, as they were introduced in C++17, and as they are currently present in C++20. While adding parallel versions to some STL algorithms is a good thing, I argue that this is not such a big advancement as one might think. Comparing the threading implications of parallel algorithms with the claims I've made in [Teodorescu20a] and [Teodorescu20b], it seems that the C++ additions only move us half-way through.

A minimal introduction into C++ parallel algorithms

To form some context for the rest of the article without spending too much time on this, let's provide an example on how to use a parallel STL algorithm.

Let's assume that we have a `transform` algorithm, and we want to parallelise it. For that, one should write something like the code in Listing 1. The only difference to a classic invocation of `transform` is the first parameter, which, in this case, tells the algorithm to use parallelisation and vectorisation.

This parameter is called *execution policy*. It tells the algorithm the type of execution that can be used for the algorithm. In the current C++20 standard there are four of these parallel policies, as explained below:

- `seq`: it will use the serial version of the algorithm, as if the argument was missing
- `par`: the algorithm can be parallelised, but not vectorised
- `par_unseq`: the algorithm can be parallelised and vectorised
- `unseq`: the algorithm can be vectorised but not parallelised (introduced only in C++20)

So, to transform an existing algorithm into a parallel (or vectorised) version, one just needs to add an argument to specify the parallel policy; the effort is minimal.

Lucian Radu Teodorescu has a PhD in programming languages and is a Software Architect at Garmin. He likes challenges; and understanding the essence of things (if there is one) constitutes the biggest challenge of all. You can contact him at lucteo@lucteo.ro


```
std::transform(std::execution::par_unseq,
  in.begin(), in.end(),   // parallel policy
  out.begin(),            // input sequence
  ftor);                  // output sequence
                          // transform fun
```

Listing 1

Please note that the library is allowed to completely ignore the execution policy and fall back to the serial execution. Thus, this execution policy provides just a hint for the library, or the maximum parallelisation/vectorisation level allowed.

Most STL algorithms take the execution policy parameter and can be instructed to run in parallel. There are also some new algorithms that were added to overcome the fact that existing algorithms have constraints that forbid parallelising them, or that there are better ways to express some parallel algorithms: `reduce`, `exclusive_scan`, `inclusive_scan`, `transform_reduce`, `transform_exclusive_scan`, `transform_inclusive_scane`.

For a better introduction and explanation of C++ parallel algorithms, the reader should consult [Lelbach16] [Filipek17a] [Filipek17b] [ONeal18].

Most of our discussion will be focusing on the parallel execution (`par` policy), the one that aims to utilise all the cores available to increase efficiency. Vectorisation (`unseq` policy) will be briefly touched towards the end of the article.

Problem 1: no concurrency concerns

The first thing to notice is that it's straightforward to adapt existing algorithms and make them parallel. This probably partially explains the success of parallel algorithms (at least at the perception level).

But this ease of use also has a negative side effect. The astute reader might have noticed that we are talking about parallelism, and not about concurrency. See [Pike13] and [Teodorescu20c] for the distinction. Very short, concurrency is a design concern, while parallelism is a run-time efficiency concern.

It is ok, for limited domains, to focus more on efficiency than design, but that's typically not the case with concurrency. Unless one pays attention to concurrent design, one will get suboptimal efficiency. In other words, multicore efficiency is a global optimisation problem, not a local one.

C++ parallel algorithms don't allow a global concurrency design. It allows only local optimisations, by making some algorithm calls parallelisable.

Considering this, things are awful from a didactical point of view: the C++ standard might teach people that one needs not to pay attention to concurrency issues; these would be magically solved by using STL. I hope that it's clear by now that this is not the case.

If we generalise a bit, we may come to the conclusion that this is the same problem that led us to bad concurrency design in the first place. Instead of recognising that concurrency needs a completely new type of design, we tried to 'patch' the old imperative and serial thinking by adding the ability

Lucian’s major critique

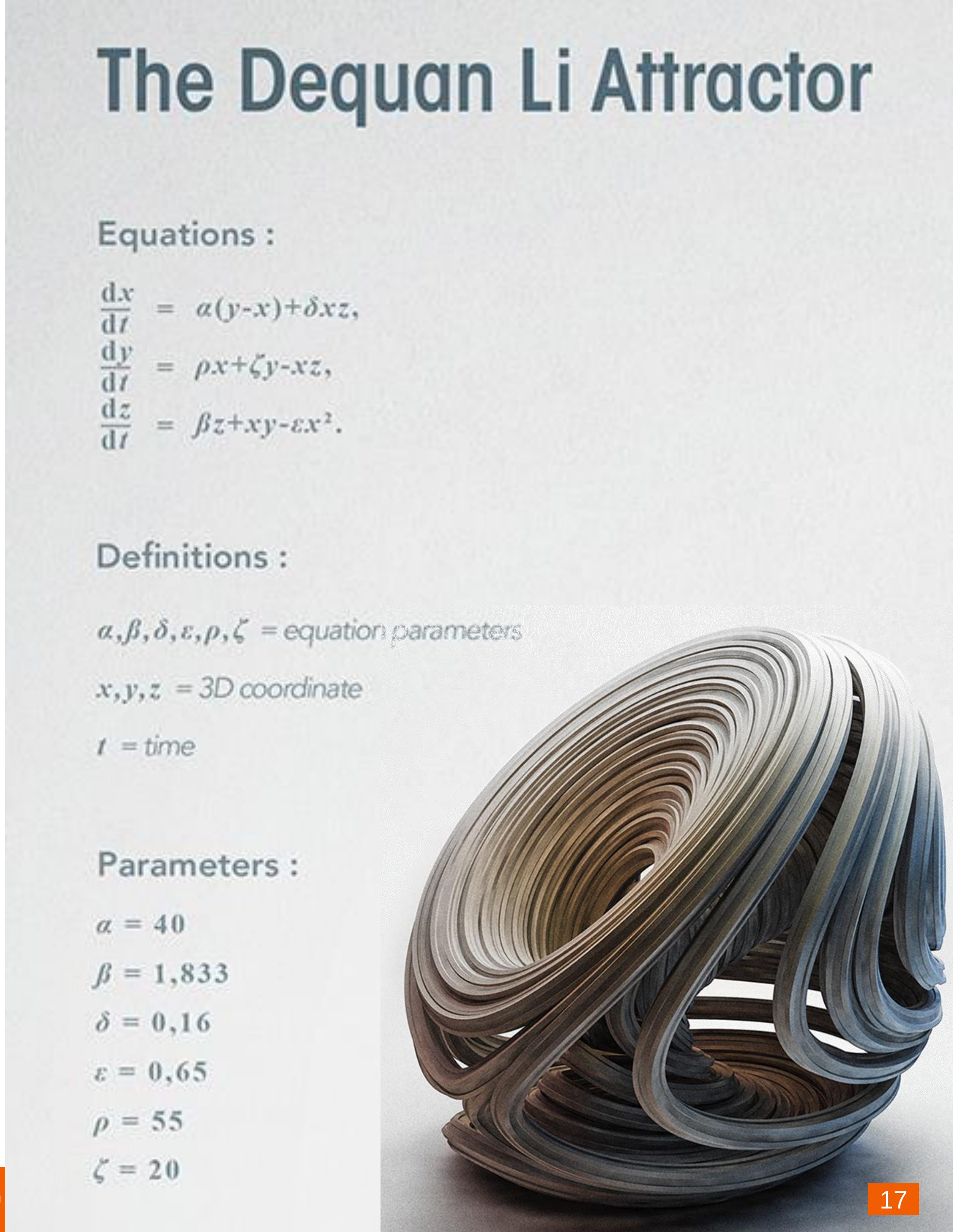
- *Applications have more than just algorithms*
- *Multiple algorithms introduce serial behavior when combined*
- *Small datasets are not good for parallelization*
- *Cannot tune the algorithms*
- *More work to finish faster*

And:

- Algorithms usually cannot monopolize the CPU

Example code to underpin Lucian's critique

- Example from HTI course "multicore programming in C++"
- Render differential equation using 'vector balls'
- Example downloadable from: https://gitlab.com/kaa-chingSX/showcase-parallel-transform
  (QR code at end of presentation)



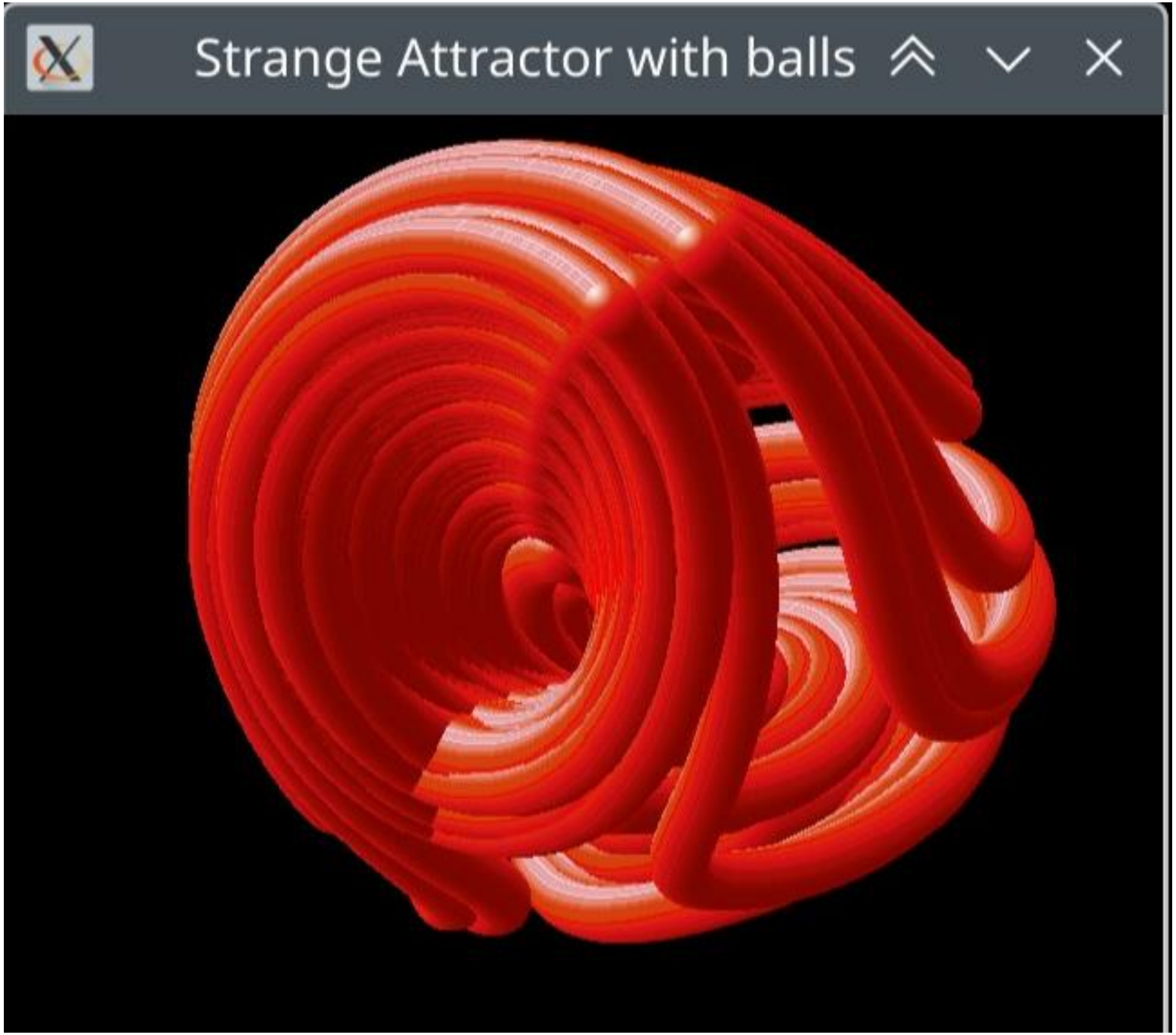
Main “3D” algorithm in “Vector Balls”

Rotate all points around vertical axis

“Painter’s algorithm”: sort based on depth

3D -> 2D conversion

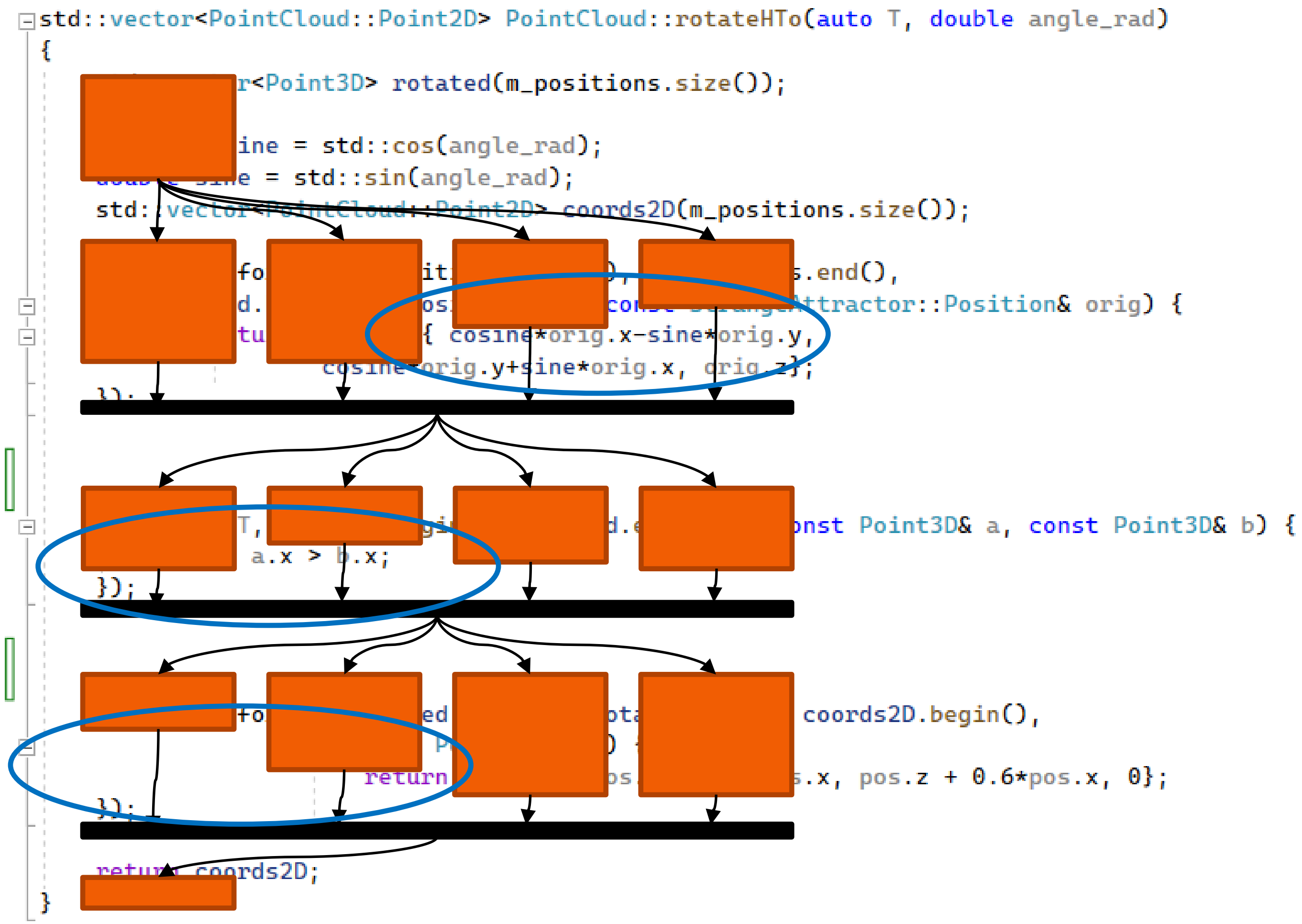
```
std::vector<PointCloud::Point2D> PointCloud::rotateHTo(auto T, double angle_rad)
{
    std::vector<Point3D> rotated(m_positions.size());

    double cosine = std::cos(angle_rad);
    double sine = std::sin(angle_rad);
    std::vector<PointCloud::Point2D> coords2D(m_positions.size());

    std::transform(T, m_positions.begin(), m_positions.end(),
        rotated.begin(), [cosine, sine](const StrangeAttractor::Position& orig) {
            return Point3D { cosine*orig.x-sine*orig.y,
                    cosine*orig.y+sine*orig.x, orig.z};
    });


    std::sort(T, rotated.begin(), rotated.end(), [](const Point3D& a, const Point3D& b) {
        return a.x > b.x;
    });


    std::transform(T, rotated.begin(), rotated.end(), coords2D.begin(),
                    [](const Point3D& pos) {
                        return Point2D{ pos.y - 0.2*pos.x, pos.z + 0.6*pos.x, 0};
    });

    return coords2D;
}
```



- Uses libSDL 1.2 to draw the ball bitmaps
- 1M+ balls to draw
- Various optimizations explored in exercise

Theodorescu's critique:

- Applications have more than just algorithms
- Multiple algorithms introduce serial behavior when combined

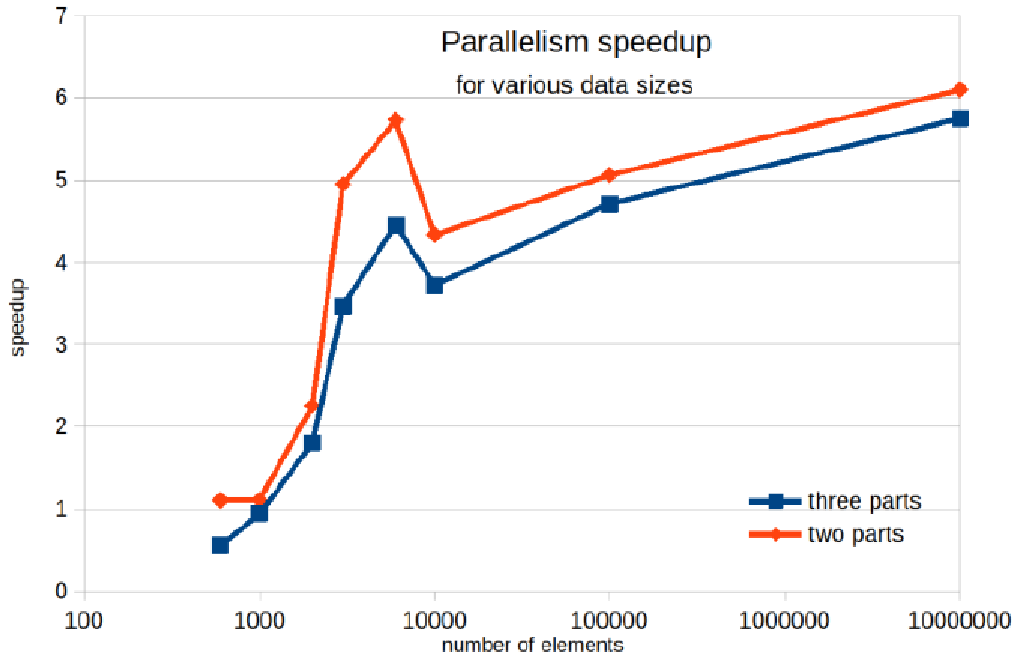


- 3 algorithms
  - Barriers in between the algorithms
    - All threads wait for slowest <= waste
- Drawing of all points is separate
  - Roughly as expensive as the calculations
  - Speeding up algorithms infinitely only gives 2x total speedup
    - **Amdahl's Law!**

Theodorescu's critique:

- Small datasets are not good for parallelization
- Cannot tune the algorithms

- Optimal:
  - Big Chunks of Work
  - Many Chunks of Work
- Either is OK, Both is better



- Theodorescu:

  - "Cannot tune the algorithms"

- This is partially correct:
  - Std suggest parallelize > 500 counts
  - But if element size is big enough, count is less relevant
  - **Programmers have no influence at all**

Theodorescu's critique:
- More work to finish faster

From earlier slide →

Wait… Using 12 cores instead of 1.
But only 1.6x faster?

- Overhead starting/stopping threads*
- Overhead barriers
- Higher speed, much more power consumption
- **Is it worth it?**

- Algorithms usually cannot monopolize the CPU
(Additional to Lucian’s critique)

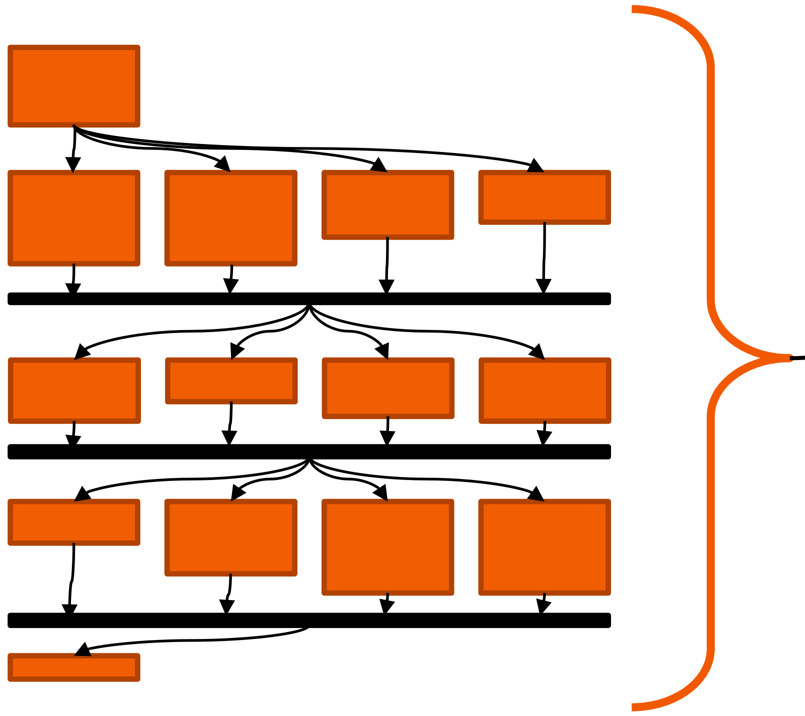
- Multicore Embedded Systems
  - Usually do not have “spare/idle cores”
  - They are planned for other teams / processes

- **You may not have the option to use concurrency**

Plan your concurrency: use parallel patterns

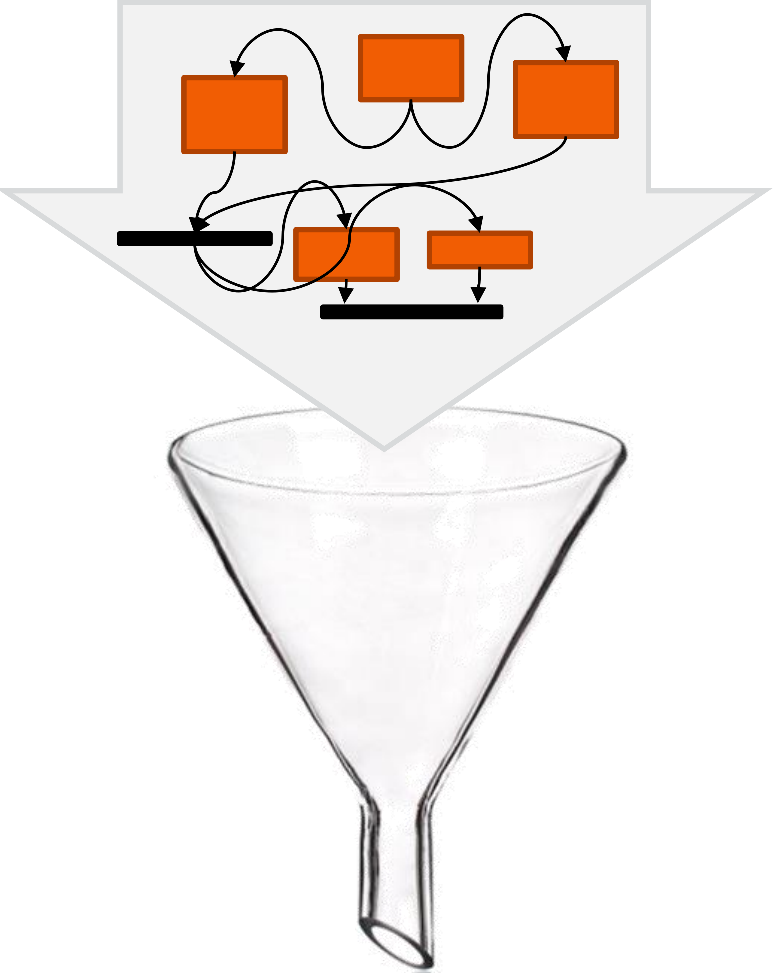
(and don’t invent libraries to implement those yourself)

Pattern example 1: Task Queue



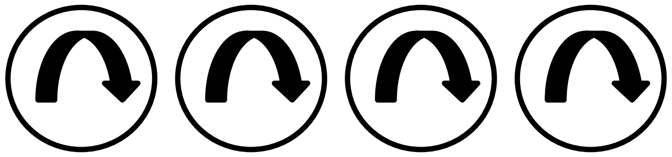
“partitioning”

Tasks



Threadsafe Queue

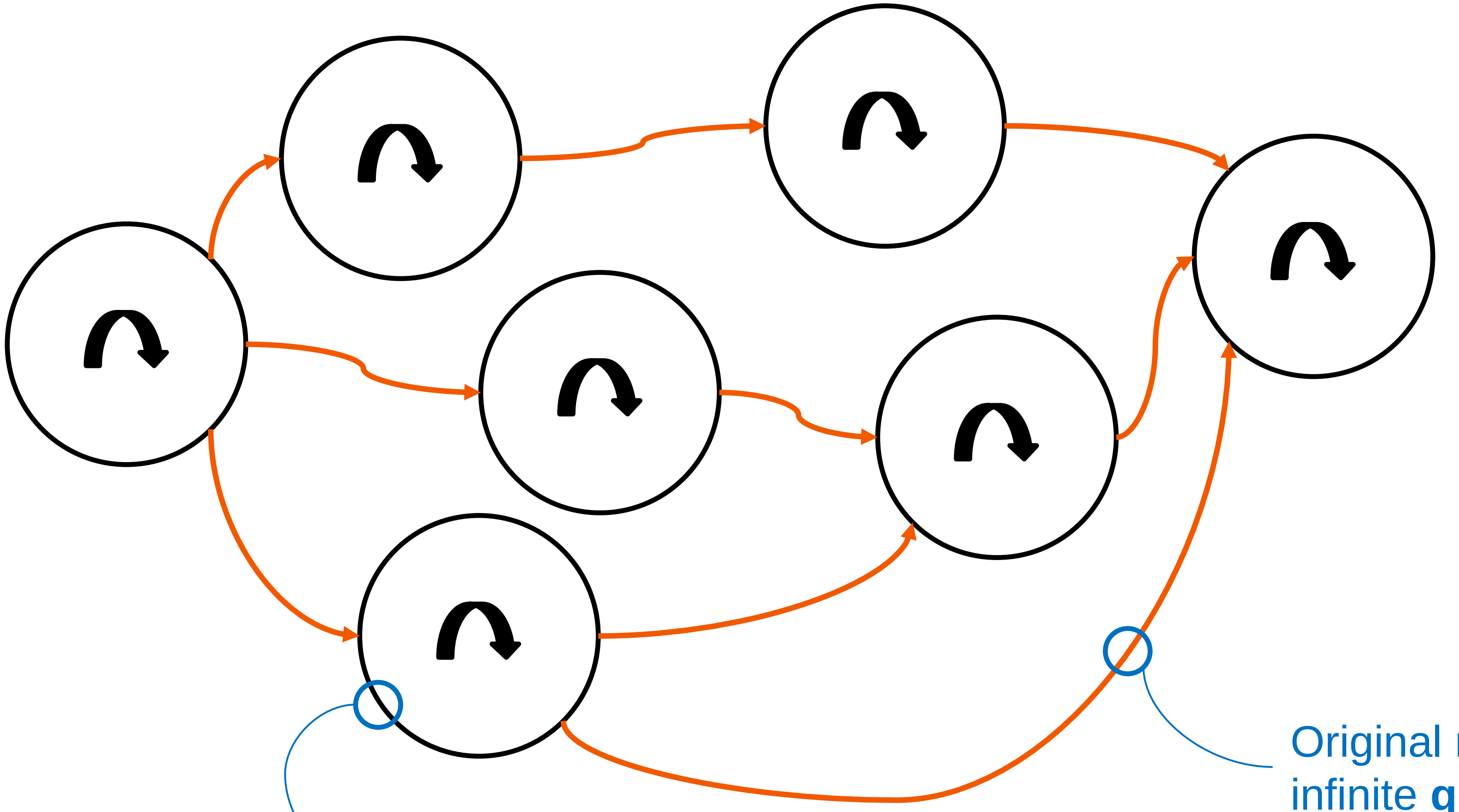
Thread Pool



Task Results may become new tasks

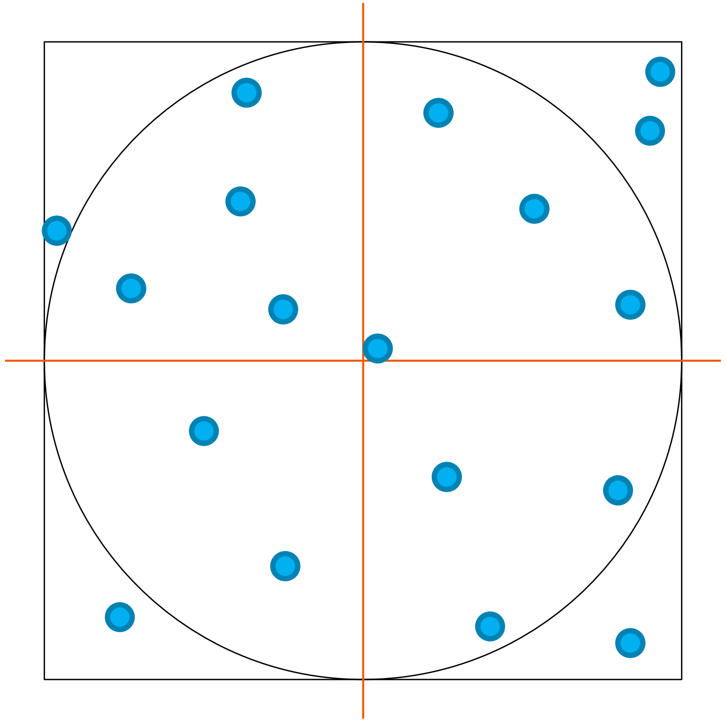
This is how OpenMP and OneTBB work internally.
Conversion from flow to tasks is done by annotations.

Pattern Example 2:
Kahn Process Network



- Code in a node follows defined pattern
- Guaranteed always progress
- Clear design to newcomers
- Slowest node is always performance bottleneck

Pattern example 3:
Algorithm: Monte Carlo-approach



An approach to calculate Pi:

- Throw darts at a unity square
  - For each point:
    - calculate distance to center
    - distance < 1 ? inside : outside
- $\text{Pi} \approx 4 \frac{\#\,inside}{\#outside}$

#total	#inside	Pi
0	0	0.0
10	10	4.0
40	32	3.2
200	158	3.17

- All experiments are independent
- All experiments use random input

Thus “embarrassingly parallel”

Don’t implement your own library!

OpenMP

- Annotate code with pragmas:

```
#pragma omp parallel for
for (auto c : candidates)
{ do work in parallel here }
```

- Needs compiler support
  Notably Visual Studio is lagging!
- Compile without OpenMP results in sequential code
- Also allows offloading of blocks to GPUs or FPGAs

- Industry consortium
- Around since late 90s

OneAPI OneTBB

- C++ library
  Supports all major compilers
- Needs code transformations similar to C++ algorithms
- Is used by gcc 12.1 to implement parallel algorithms

- Fully open sourced (github)
- Intel-supported
- Around since early 2000s

Plain C++ (DIY)

- Implement abstractions or all of your code will contain locks and asyncs
- Hard to maintain
- Do you have good unit tests?
- Do you have a sequential “golden standard” to compare results against?

- Supported by you only?

Summary

Summary

- Is multithreading in C++ hard?
  - Not anymore, is standardized now
  - Getting it right is still hard

Not an excuse to roll your own, use a library to abstract threading away

- C++17 brings parallelized algorithms
  - Be careful with performance expectations!
- Parallel patterns have seen a lot of research
  - See the paper for links to books
  - There’s also a training course!

Gitlab repo with paper and slides

Multicore Programming Training course

Questions?

klaas.van.gend@sioux.eu

- Is multithreading in C++ hard?
  - Not anymore, is standardized no
  - Getting it right is still hard
- C++17 brings parallelized algorithms
  - Be careful with performance expectation
- Parallel patterns have seen a lot of research
  - See the paper for links to books
  - There's also a training course!

Not an excuse to roll your own, use a library to abstract threading away

Gitlab repo with paper and slides

Multicore Programming Training course