C++ prgramming in a parallel world

CPP Europe
Bucharest 2020

J. Daniel Garcia

ARCOS Group
University Carlos III of Madrid
Spain

February 25th, 2020

Warning

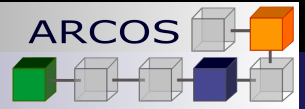
This work is under Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) license.

You are **free** to **Share** — copy and redistribute the material in any medium or format.

You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

You may not use the material for commercial purposes.

If you remix, transform, or build upon the material, you may not distribute the modified material.



Who am I?

- A C++ programmer.
  - Started writing C++ code in 1989.

Who am I?

- A C++ programmer.
  - Started writing C++ code in 1989.
- A university professor in Computer Architecture.
  - University Carlos III of Madrid (since 2001).

Who am I?

- A C++ programmer.
  - Started writing C++ code in 1989.
- A university professor in Computer Architecture.
  - University Carlos III of Madrid (since 2001).
- An ISO C++ language standards committee member.
  - AENOR: Spanish Standards National Body.

Who am I?

- A C++ programmer.
  - Started writing C++ code in 1989.
- A university professor in Computer Architecture.
  - University Carlos III of Madrid (since 2001).
- An ISO C++ language standards committee member.
  - AENOR: Spanish Standards National Body.
- **My goal**: Improve applications programming.
  - **Performance** → **faster** applications.
  - **Energy efficiency** → **better** performance per Watt.
  - **Maintainability** → **easier** to modify.
  - **Reliability** → **safer** components.

ARCOS@uc3m

- **UC3M**: A young international research oriented university.
- **ARCOS**: An applied research group.
  - **Lines**: High Performance Computing, Big data, Cyberphysical systems, **Programming Models for Applications Improvement**.
- **Improving applications**:
  - **REPARA**: Reengineering and Enabling Performance and poweR of Applications. FP7-ICT (2013–2016).
  - **RePhrase**: REfactoring Parallel Heterogeneous Resource Aware Applications. H2020-ICT (2015–2018).
  - **ASPIDE**: exAScale ProgrammIng models for extreme Data procEssing. H2020-FET-HPC (2018–2020).
- **Standardization**:
  - ISO/IEC JTC/SC22/WG21. ISO C++ Committee.

1 Times have changed

2 What do you do with multicore?

3 Parallelism in C++17

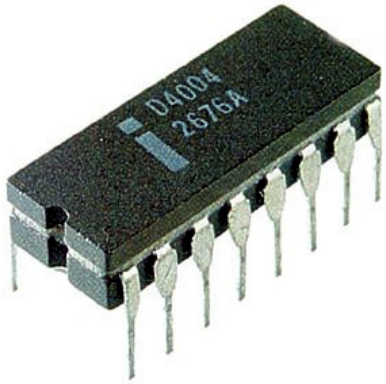
4 After C++20: Executors

5 What can else can I do?

6 Summary

First microprocessor

- Intel 4004 (1971).
  - **Application domain**: Calculators.
  - **Technology**: 10,000 nm.
  - **Data**:
    - 2300 transistors.
    - 13 mm2
    - 108 KHz
    - 12 Volts
  - **Features**:
    - 4-bits data.
    - Data-path in one cycle.



Intel 4004 photo by Rostislav Lisovy

Unicom 141P Calculator 3 photo by Michael Holley.

My first computer

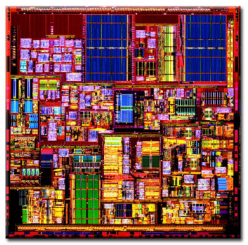
Sinclair ZX-Spectrum

- Zilog Z80 (1976).
  - **Application domain**: Home computers, videoconsoles.
  - **Technology**: 4,000 nm.
  - **Data**:
    - 8500 transistors.
    - 2.5 MHz
    - 5 Volts
  - **Features**:
    - 8-bits data.

Zilog Z80 photo by Konstantin Lanzet

ZX Spectrum photo by Bill Bertram.

Last single core

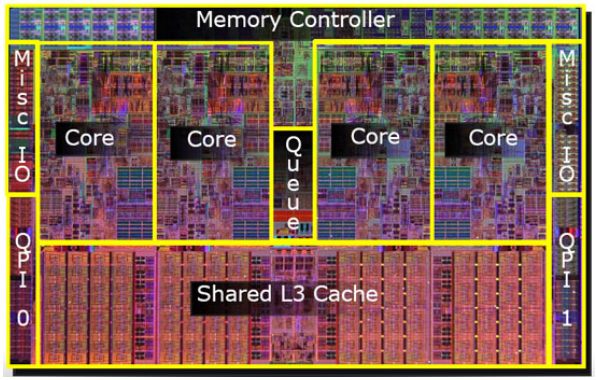


Die of Intel Pentium 4 (Northwood) Source: `http://gecko54000.free.fr`

- Intel Pentium 4 (2003).
  - **Application domain**: Desktop / Servers.
  - **Technology**: 90 nm (1/100x).
- **Data**:
  - 55M transistors (20,000x).
  - 101 mm^2 (10x).
  - 3.4 GHz (10,000x).
  - 1.2 Volts (1/10x).
- **Features**:
  - 32/64-bit data (16x).
  - Data path with 22 pipeline stages (later 31).
  - 3-4 instructions per cycle (superscalar).
  - Two level cache on chip.
  - Data parallel instructions (SIMD).
  - Hyper-threading.

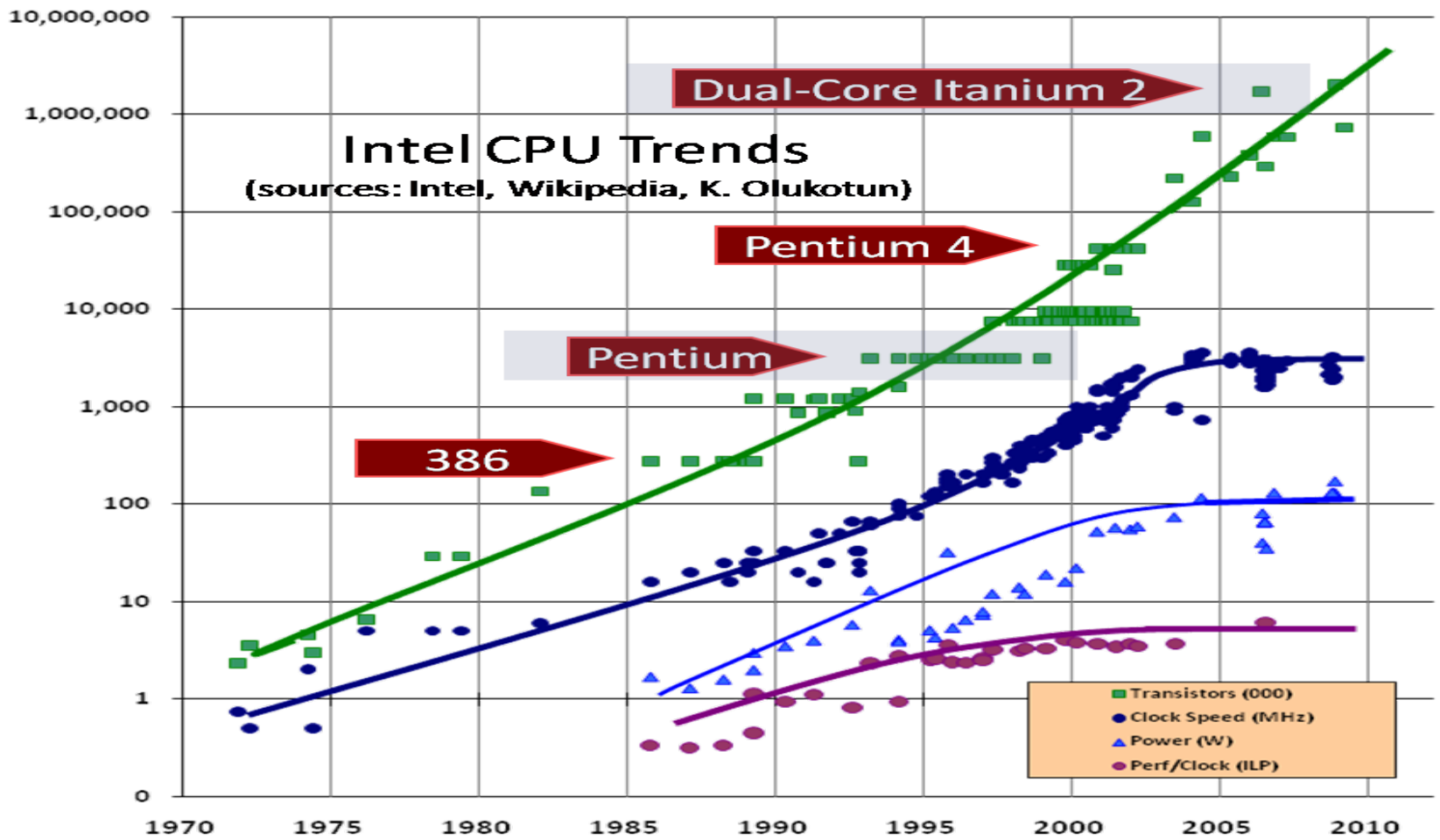
A typical multicore

- Intel Core i7 (2009).
  - **Application**: Desktop / Server.
  - **Technology**: 45 nm (1/2x).
- **Data**:
  - 774M transistors (12x).
  - 296 mm^2 (3x).
  - 3.2 GHz – 3.6 GHz (≈1x).
  - 0.7 – 1.4 Volts (≈1x).
- **Features**:
  - 128-bit data (2x).
  - Datapath with 14-stage pipeline (0.5x).
  - 4 instructions per cycle (≈1x).
  - Three level cache on chip.
  - Data parallel instructions (SIMD).
  - 4 cores (4x) + Hyper-threading.



Die of Intel Core i7 (Nehalem)
Source: `www.legitreviews.com`

What happened?



Source: The free lunch is over. Herb Sutter. `http://www.gotw.ca/publications/concurrency-ddj.htm`

1 Times have changed

2 What do you do with multicore?

3 Parallelism in C++17

4 After C++20: Executors

5 What can else can I do?

6 Summary

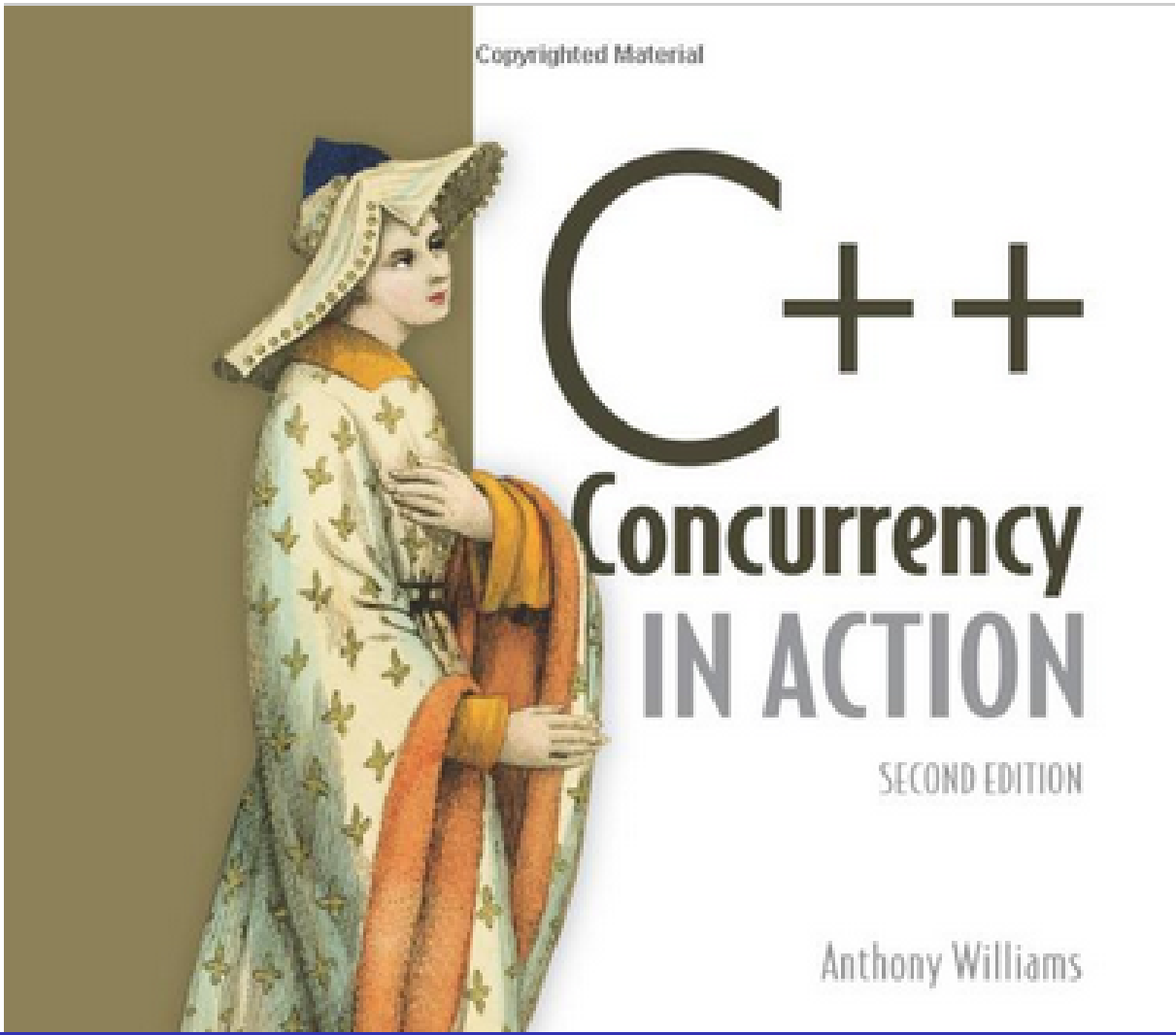
Impact of multicores

- Increase **throughput**:
  - More transactions per second.
  - Mostly **concurrent programming**.
- Increase performance
  - Faster execution of a task.
  - Mostly **parallel programming**.

C++11/14

- Focused in providing the concurrency building blocks.
- Main features:
  - Clear definition of the memory model.
  - Support for TLS (**thread_local**).
  - Concurrency portable abstractions:
    - **std::thread**.
    - **std::mutex**, **std::timed_mutex**, ...
    - **std::condition_variable**, **condition_variable_any**.
    - **std::unique_lock**, ...
    - **std::promise**, **std::future**, **std::packaged_task**.
  - Low level portable *lock-free* abstractions:
    - **std::atomic**.
    - **std::memory_order**.

Where do I get this?



1 Times have changed

2 What do you do with multicore?

3 Parallelism in C++17

4 After C++20: Executors

5 What can else can I do?

6 Summary

3 Parallelism in C++17
- Introduction
- Execution policies
- Updating global state
- Transformations
- Reductions
- Map/Reduce
- Scans
- More algorithms

Parallel algoritms

- Many algorithms in the STL have now a parallel version.
  - They take a new first argument to specify the execution policy.

```
// Traditional way -> sequential
std::for_each(v.begin(), v.end(), [](auto & x) { f(x); });
```

Parallel algoritms

- Many algorithms in the STL have now a parallel version.
  - They take a new first argument to specify the execution policy.

```
// Traditional way -> sequential
std::for_each(v.begin(), v.end(), [](auto & x) { f(x); });

// New parallel
std::for_each(std::execution::par,
  v.begin(), v.end(), [](auto & x) { f(x); });
```

Parallel algoritms

- Many algorithms in the STL have now a parallel version.
  - They take a new first argument to specify the execution policy.

```
// Traditional way -> sequential
std::for_each(v.begin(), v.end(), [](auto & x) { f(x); });

// New parallel
std::for_each(std::execution::par,
  v.begin(), v.end(), [](auto & x) { f(x); });

// New sequential
std::for_each(std::execution::seq,
  v.begin(), v.end(), [](auto & x) { f(x); });
```

Processing images

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

#include "image.h"

int main() {
  std::vector<image> v = load_images("file.dat");

  std::for_each(std::execution::par,
    v.begin(), v.end(), [](auto & img) { img.to_gray(); });

  store_images("newfile.dat", v);
}
```

Sorting

- Sorting requires many applications of the comparator.
  - Specially interesting when comparator is not trivial.

```
std::vector<customer> v = get_customers();

std::sort(std::execution::par, v.begin(), v.end(),
    [](const auto & e1, const auto & e2) {
      if (e1.name == e2.name) return e1.last < e2.last;
      else return e1.name < e2.name;
    }
);
```

3 Parallelism in C++17
- Introduction
- Execution policies
- Updating global state
- Transformations
- Reductions
- Map/Reduce
- Scans
- More algorithms

Overview of execution policies

- **std::execution::seq**.
  - Class **std::execution::sequenced_policy**.
  - Algorithm executes sequentially (single thread).
  - Might have changes over traditional algorithm.

Overview of execution policies

- **std::execution::seq**.
  - Class **std::execution::sequenced_policy**.
  - Algorithm executes sequentially (single thread).
  - Might have changes over traditional algorithm.
- **std::execution::par**.
  - Class **std::execution::parallel_policy**.
  - Algorithm executes in multiple threads.
  - No vectorization!

Overview of execution policies

- **std::execution::seq**.
  - Class **std::execution::sequenced_policy**.
  - Algorithm executes sequentially (single thread).
  - Might have changes over traditional algorithm.
- **std::execution::par**.
  - Class **std::execution::parallel_policy**.
  - Algorithm executes in multiple threads.
  - No vectorization!
- **std::execution::par_unseq**.
  - Class **std::execution::parallel_unsequenced_policy**.
  - Algorithm executes in multiple threads.
  - Vectorization allowed!

Overview of execution policies

- **std::execution::seq**.
  - Class **std::execution::sequenced_policy**.
  - Algorithm executes sequentially (single thread).
  - Might have changes over traditional algorithm.
- **std::execution::par**.
  - Class **std::execution::parallel_policy**.
  - Algorithm executes in multiple threads.
  - No vectorization!
- **std::execution::par_unseq**.
  - Class **std::execution::parallel_unsequenced_policy**.
  - Algorithm executes in multiple threads.
  - Vectorization allowed!
- **std::execution::unseq** (**C++20**).
  - Class **std::execution::unsequenced_policy**.
  - Algorithm executes in single thread.
  - Vectorization allowed!

Constraints on iterators

- Some algorithms on the STL require ranges expressed as **input iterators**.

```
template< class InputIt, class T >
typename iterator_traits<InputIt>::difference_type
  count( InputIt  first , InputIt  last , const T &value );
```

Constraints on iterators

- Some algorithms on the STL require ranges expressed as **input iterators**.

```
template< class InputIt, class T >
typename iterator_traits<InputIt >::difference_type
  count( InputIt  first , InputIt  last , const T &value );
```

- Execution policy based require iterators to be **forward iterators**

```
template< class ExecutionPolicy, class ForwardIt, class T >
typename iterator_traits<ForwardIt>::difference_type
  count(ExecutionPolicy&& policy,
        ForwardIt  first , ForwardIt  last , const T &value );
```

Changes in algorithms interface

- Some algorithms have changed their return types.

Changes in algorithms interface

- Some algorithms have changed their return types.
- Without execution policy.
  - Returns the comparator object.

```
template <class InputIt, class UnaryFunction>
constexpr UnaryFunction
  for_each(InputIt first, InputIt last, UnaryFunction f);
```

Changes in algorithms interface

- Some algorithms have changed their return types.
- Without execution policy.
  - Returns the comparator object.

```
template <class InputIt, class UnaryFunction>
constexpr UnaryFunction
  for_each(InputIt first, InputIt last, UnaryFunction f);
```

- With execution policy.
  - Does not return any value.

```
template <class ExecutionPolicy, class ForwardIt,
          class UnaryFunction2>
void
  for_each(ExecutionPolicy&& policy,
           ForwardIt first, ForwardIt last, UnaryFunction2 f);
```

What about exceptions?

- In non execution policy based exceptions can be thrown.

```
std::for_each(v.begin(), v.end(),
   [](auto & x) {
     if (valid(x)} f(x);
    else throw invalid_value{x}; // Throws exception
   });
```

What about exceptions?

- In non execution policy based exceptions can be thrown.

```
std::for_each(v.begin(), v.end(),
  [](auto & x) {
    if (valid(x)) f(x);
    else throw invalid_value{x}; // Throws exception
  });
```

- In excecution policy based exceptions translate into **std::terminate**.

```
std::for_each(std::execution::seq, v.begin(), v.end(),
  [](auto & x) {
    if (valid(x)) f(x);
    else throw invalid_value{x}; // Invoke std::terminate
  });
```

When to avoid execution policies

- Using input or output iterators.

When to avoid execution policies

- Using input or output iterators.
- Avoid calling **std::terminate** on exceptions.

When to avoid execution policies

- Using input or output iterators.
- Avoid calling **std::terminate** on exceptions.
- Avoid side effects on use of elements.

When to avoid execution policies

- Using input or output iterators.
- Avoid calling **std::terminate** on exceptions.
- Avoid side effects on use of elements.
- Make use of return values (e.g. **std::for_each()**.

3 Parallelism in C++17
- Introduction
- Execution policies
- Updating global state
- Transformations
- Reductions
- Map/Reduce
- Scans
- More algorithms

Counting valid elements

```
long count = 0;
std::vector<double> v = get_values();

std::for_each(std::execution::par,
   v.begin(), v.end(),
   [&](double x) {
      if (x>0) count++;
   }
);

std::cout << "Count= " << count << "\n";
```

Solving the data race: mutexes

```
long count = 0;
std::mutex m;
std::vector<double> v = get_values();

std::for_each(std::execution::par,
   v.begin(), v.end(),
   [&](double x) {
      if (x>0) {
         std::lock_guard<std::mutex> l{m};
         count++;
      }
   }
);

std::cout << "Count= " << count << "\n";
```

Solving the data race: atomics

```
std::atomic<long> count = 0;
std::vector<double> v = get_values();

std::for_each(std::execution::par,
   v.begin(), v.end(),
   [&](double x) {
      if (x>0) count++;
   }
);

std::cout << "Count= " << count << "\n";
```

Or even better

```
std::vector<double> v = get_values();

long count = std::count_if(std::execution::par,
   v.begin(),  v.end(),
    []( double x) {
      return x>0
   }
);

std::cout << "Count= " << count << "\n";
```

Remember

- Accessing global state from algorithms may result in data races.
  - Using mutexes may be a heavyweight solution.
  - Atomics hav limited applicability.

Remember

- Accessing global state from algorithms may result in data races.
  - Using mutexes may be a heavyweight solution.
  - Atomics hav limited applicability.
- There might be a better algorithm.
  - A **std::for_each()** call may be a **code smell**.

3 Parallelism in C++17
- Introduction
- Execution policies
- Updating global state
- Transformations
- Reductions
- Map/Reduce
- Scans
- More algorithms

The map-pattern

- A well known pattern in functional programming.
  - Apply an operation to every element in a data set to generate a new data set.

Squaring values

```
std::vector<double> square(const std::vector<double> & v)
{
  std::vector<double> r(v.size());

  std::transform(std::sequential::par,
    v.begin(), v.end(), r.begin(),
     [](double x) { return x*x; }
  );

  return r;
}
```

Adding vectors

```
std::vector<double> add(const std::vector<double> & v,
                        const std::vector<double> & w)
{
  std::vector<double> r(v.size());

  std::transform(std::sequential::par,
    v.begin(), v.end(), w.begin(), r.begin(),
     []( double x, double y) { return x+y; }
  );

  return r;
}
```

Heterogeneous transformations

```
std::vector<std::complex<double>> create_cplx(
    const std::vector<double> & re,
    const std::vector<double> & im)
{
  auto sz = std::min(re.size(), im.size());
  std::vector<std::complex<double>> res(sz);

  std::transform(std::execution::par,
    re.begin(), re.end(), im.begin(),
    res.begin(),
    []( double r, double i) -> complex<double> {
      return {r,i};
    });
  return res;
}
```

3 Parallelism in C++17
- Introduction
- Execution policies
- Updating global state
- Transformations
- Reductions
- Map/Reduce
- Scans
- More algorithms

Reduction pattern

- A **reduction** computes the sum of all elements in a data set.

Reduction pattern

- A **reduction** computes the sum of all elements in a data set.
- **Note**: **std::reduce** looks quite similar to **std::accumulate** on the surface.
  - Result is not deterministic unless the sum opration is both associative and commutative.

Add all elements in a vector

```
void print_add(const std::vector<double> & v)
{
  double r = std::reduce(std::execution::par,
    v.begin(), v.end());

  std::cout << "sum= " << r << "\n";
}
```

- Initial value is **value_type{}**.
- Binary operation is **std::plus<>()**.

Providing initial value

```
void print_add(const std::vector<double> & v)
{
  double r = std::reduce(std::execution::par,
    v.begin(), v.end(), 100.0);

  std::cout << "sum= " << r << "\n";
}
```

- Still reduction operation is **std::plus<>()**.

Providing reduction operator

```
void print_add(const std::vector<double> & v)
{
  double r = std::reduce(std::execution::par,
    v.begin(), v.end(), 0.0,
     []( double x, double y) { return x+y; }
  );

  std::cout << "sum= " << r << "\n";
}
```

3 Parallelism in C++17
- Introduction
- Execution policies
- Updating global state
- Transformations
- Reductions
- Map/Reduce
- Scans
- More algorithms

Map/reduce pattern

- A **map-reduce** pattern combines a **map** pattern with a **reduce** pattern over the results of that map.
  - In C++ it is spelled out **std::transform_reduce**.

Computing the norm of a vector

```
void print_norm(const std::vector<double> & v)
{
  double s = std::transform_reduce(std::execution::par,
    v.begin(), v.end(),
    0.0,
    [](double x, double y) { return x + y },
    [](double x) { return x * x; }
  );

  std::cout << "Norm: " << std::sqrt(s) << "\n";
}
```

Computing aggregate area

```
double area(const std::vector<shape> & shapes)
{
  return std::map_reduce(std::execution::par,
    shapes.begin(), shapes.end(),
    0.0,
    [](double x, double y) { return x+y; },
    [](const shape & s) { return s.area(); }
  );
}
```

Cannonical example

- Word frequencies from sequence of words.
  - Associative container with **<word,freq>**.

```
auto word_freq(const std::vector<std::string> & words)
{
```

Cannonical example

- Word frequencies from sequence of words.
  - Associative container with **<word,freq>**.

```
auto word_freq(const std::vector<std::string> & words)
{
  using dictionary = std::map<std::string,long>;
```

Cannonical example

- Word frequencies from sequence of words.
  - Associative container with **<word,freq>**.

```
auto word_freq(const std::vector<std::string> & words)
{
  using dictionary = std::map<std::string,long>;
  return std::transform_reduce(std::execution::par,
```

Cannonical example

- Word frequencies from sequence of words.
  - Associative container with **<word,freq>**.

```
auto word_freq(const std::vector<std::string> & words)
{
  using dictionary = std::map<std::string,long>;
  return std::transform_reduce(std::execution::par,
    words.begin(), words.end(), dictionary {},
```

Cannonical example

- Word frequencies from sequence of words.
  - Associative container with **<word,freq>**.

```
auto word_freq(const std::vector<std::string> & words)
{
  using dictionary = std::map<std::string,long>;
  return std::transform_reduce(std::execution::par,
    words.begin(), words.end(), dictionary {},
    [](dictionary & lhs, const dictionary & rhs) -> dictionary {
      for (auto & [key,value] : rhs) { lhs[key] += value; }
      return lhs;
    },
```

Cannonical example

- Word frequencies from sequence of words.
  - Associative container with **<word,freq>**.

```
auto word_freq(const std::vector<std::string> & words)
{
  using dictionary = std::map<std::string,long>;
  return std::transform_reduce(std::execution::par,
    words.begin(), words.end(), dictionary {},
    [](dictionary & lhs, const dictionary & rhs) -> dictionary {
      for (auto & [key,value] : rhs) { lhs[key] += value; }
      return lhs;
    },
    [](const std::string & s) -> dictionary { return {w,1}; }
  );
}
```

3 Parallelism in C++17
- Introduction
- Execution policies
- Updating global state
- Transformations
- Reductions
- Map/Reduce
- Scans
- More algorithms

Scan pattern

- A **scan** pattern computes a sequence of partial reductions on a dataset.
  - A scan on $x_0, x_1, x_2, \ldots$
  - Results in the sequence:
    - x_0
    - $x_0 + x_1$
    - $x_0 + x_1 + x_2$
    - $\ldots$
- Two alternatives:
  - **std::exclusive_scan()**
  - **std::inclusive_scan()**

Computing CDF

```
auto compute_cdf(const std::vector<int> & histogram)
{
  std::vector<int> cdf(histogram.size());

  std::inclusive_scan(std::execution::par,
    histogram.begin(), histogram.end(),
    cdf.begin(),
    0
  );

  return cdf;
}
```

Combining transform and scan

```
auto compute_cdf(const std::vector<int> & histogram)
{
  std::vector<int> cdf(histogram.size());

  std::transform_inclusive_scan(std::execution::par,
    histogram.begin(), histogram.end(),
    cdf.begin(),
    0,
    [](auto x, auto y) { return x+y; }
    [](auto x) {
      if (x<0) return 0;
      if (x>255) return 255;
      return x;
    }
  );
```

3 Parallelism in C++17
- Introduction
- Execution policies
- Updating global state
- Transformations
- Reductions
- Map/Reduce
- Scans
- More algorithms

What algorithms are parallel

- Most algorithms have an execution policy based version.
- Few exceptions:
  - Numerics replaced by new versions: **accumulate**, **inner_product**, **partial_sum**.
  - Backwards algorithms: **copy_backward**, **move_backward**.
  - Searching: some versions of **search**.
  - Sampling and permuting: **sample**, **shuffle**, ***_permutation**.
  - Partitioning: **partition_point**.
  - Bounds search: ***_bound**, **equal_range**.
  - Heap based: ***_heap**.

1 Times have changed

2 What do you do with multicore?

3 Parallelism in C++17

4 After C++20: Executors

5 What can else can I do?

6 Summary

DISCLAIMER

This section contains tentative design that has currently under discussion.

Context

- A possible **future**:
  - Composition of networked, asynchronous parallel computations.
  - Accelerated by diverse hardware

Context

- A possible **future**:
  - Composition of networked, asynchronous parallel computations.
  - Accelerated by diverse hardware
- But the **present**:
  - Low-level concurrency primitives (**std::thread**, **std::atomic**, . . .).
  - Components with known problems (**std::async**, **std::future**, . . .).
  - Parallel algorithms neither flexible nor composable.

Context

- A possible **future**:
  - Composition of networked, asynchronous parallel computations.
  - Accelerated by diverse hardware
- But the **present**:
  - Low-level concurrency primitives (**std::thread**, **std::atomic**, . . .).
  - Components with known problems (**std::async**, **std::future**, . . .).
  - Parallel algorithms neither flexible nor composable.
- **Solution** with two components:
  - **executor**s
  - **sender**s and **receiver**s.

Executors

- Executors:
  - A work execution interface.
    - Any **executor** type.

```
using namespace std::execution;
std::static_thread_pool p(16);
executor auto ex = p.executor();
execute(ex, []{ do_the_work(); });
```

Senders and receivers

- Senders and receivers:
  - A representation of work and interrelationships.
    - **sender** types.
    - **receiver** types.

```
sender auto begin = schedule(ex);
sender auto next = then(begin, [] { f(); return 42; });
sender auto job = then(next, [](int x) { g(x); return 99;
    });

receiver auto doit = as_receier([](int x) { store(x); });
submit(job, doit);
```

What is an executor

- A lightweight handle to an execution context.
  - A thread pool.
  - SIMD units.
  - GPUs.
  - Current thread.
  - ...

The simplest executor

- An inline executor executes the work immediately.

```
struct inline_executor {
  template<class F>
  void execute(F&& f) const noexcept {
    std::invoke(std::forward<F>(f));
  }

  auto operator<=>(const inline_executor&) const = default;
};
```

Bulk execution

- Another example of control structure provided by an executor.
  - Creates a group of functions calls in a single operation.

```
struct simd_executor : inline_executor {
  template<class F>
  simd_sender bulk_execute(F f, size_t n) const {
    #pragma simd
    for(size_t i = 0; i != n; ++i) {
      std::invoke(f, i);
    }

    return {};
  }
};
```

An executor based for-each

```
template<class Executor, class F, class Range>
void my_for_each(const Executor& ex, F f, Range rng) {
  // request bulk execution, receive a sender
  sender auto s = execution::bulk_execute(ex,
    [=]( size_t i ) {
      f(rng[i]);
    }
  );

  // initiate execution and wait for it to complete
  execution::sync_wait(s);
}
```

A future asynchronous STL?

```
sender auto s =
   just(3)  |                          // produce '3' immediately
   via(scheduler1) |                   //  transition  context
   then([]( int  a){return a+1;}) |    // chain continuation
   then([]( int  a){return a*2;}) |    // chain another continuation
   via(scheduler2) |                   //  transition  context
   handle_error([]( auto e){
      return just(3) ;}) ;             // with default value on
            errors

int  r  = sync_wait(s);                // wait for the result
```

1 Times have changed

2 What do you do with multicore?

3 Parallelism in C++17

4 After C++20: Executors

5 What can else can I do?

6 Summary

GrPPI

https://github.com/arcosuc3m/grppi

GrPPI

https://github.com/arcosuc3m/grppi

- **G**eneric **r**eusable **P**arallel **P**attern **Interface**.
  - A header only library.
  - A set of execution policies.
  - A set of type safe generic algorithms.
  - Requires **C++14**.
  - Apache 2.0 License.

5 What can else can I do?

- Controlling execution
- Pipelines
- Farm of tasks
- Controlling the buffering
- Filter

Execution types

- Execution model is encapsulated in execution types.
  - Always provided as first argument to patterns.
- Current concrete execution types:
  - **Sequential**: **sequential_execution**.
  - **ISO C++ Threads**: **parallel_execution_native**.
  - **OpenMP**: **parallel_execution_omp**.
  - **Intel TBB**: **parallel_execution_tbb**.
  - **FastFlow**: **parallel_execution_ff**.
- Run-time polymorphic wrapper through type erasure:
  - **dynamic_execution**.

Execution model properties

- Some execution types allow finer configurtion.
  - Example: Concurrency degree.

Execution model properties

- Some execution types allow finer configurtion.
  - Example: Concurrency degree.

- Interface:

```
ex.set_concurrency_degree(4);
int  n = ex.concurrency_degree();
```

Execution model properties

- Some execution types allow finer configurtion.
  - Example: Concurrency degree.
- Interface:

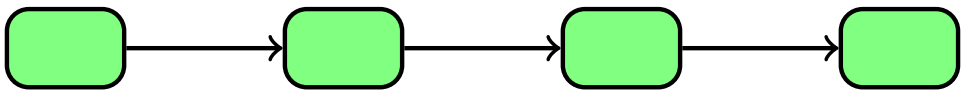
```
ex.set_concurrency_degree(4);
int  n = ex.concurrency_degree();
```

- **Default values**:
  - **Sequential** ⇒ **1**.
  - **Native** ⇒ **std::thread::hardware_concurrency()**.
  - **OpenMP** ⇒ **omp_get_num_threads()**.

5 What can else can I do?
- Controlling execution
- Pipelines
- Farm of tasks
- Controlling the buffering
- Filter

Pipeline pattern

- A **pipeline** pattern allows processing a data stream where the computation may be divided in multiple stages.
  - Each stage processes the data item generated in the previous stage and passes the produced result to the next stage.



Standalone pipeline

- A **standalone pipeline** is a top-level pipeline.
  - Invoking the pipeline translates into its execution.

Standalone pipeline

- A **standalone pipeline** is a top-level pipeline.
  - Invoking the pipeline translates into its execution.

- Given:
  - A **generator** $g : \varnothing \mapsto T_1 \cup \varnothing$
  - A sequence of **transformers** $t_i : T_i \mapsto T_{i+1}$

- For every **non-empty** value generated by g, it evaluates:
  - $t_n(t_{n-1}(\ldots t_1(g())))$

Generators

- A generator *g* is any callable C++ entity that:
  - Takes no argument.
  - Returns a value of type *T* that may hold (or not) a value.
  - Null value signals end of stream.

Generators

- A generator *g* is any callable C++ entity that:
  - Takes no argument.
  - Returns a value of type *T* that may hold (or not) a value.
  - Null value signals end of stream.
- The return value must be any type that:
  - Is copy-constructible or move-constructible.

```
T x = g();
```

Generators

- A generator *g* is any callable C++ entity that:
  - Takes no argument.
  - Returns a value of type *T* that may hold (or not) a value.
  - Null value signals end of stream.
- The return value must be any type that:
  - Is copy-constructible or move-constructible.

```
T x = g();
```

  - Is contextually convertible to **bool**

```
if (x) { /* ... */ }
if (!x) { /* ... */ }
```

Generators

- A generator *g* is any callable C++ entity that:
  - Takes no argument.
  - Returns a value of type *T* that may hold (or not) a value.
  - Null value signals end of stream.
- The return value must be any type that:
  - Is copy-constructible or move-constructible.

    ```
    T x = g();
    ```

  - Is contextually convertible to **bool**

    ```
    if (x) { /* ... */ }
    if (!x) { /* ... */ }
    ```

  - Can be derreferenced

    ```
    auto val = *x;
    ```

Generators

- A generator *g* is any callable C++ entity that:
  - Takes no argument.
  - Returns a value of type *T* that may hold (or not) a value.
  - Null value signals end of stream.
- The return value must be any type that:
  - Is copy-constructible or move-constructible.

    ```
    T x = g();
    ```

  - Is contextually convertible to **bool**

    ```
    if (x) { /* ... */ }
    if (!x) { /* ... */ }
    ```

  - Can be derreferenced

    ```
    auto val = *x;
    ```

- The standard library offers an excellent candidate **std::optional<T>**.

Simple pipeline: x -> x*x -> 1/x -> print

```
template <typename Execution>
void run_pipe(const Execution & ex, int n)
{
  grppi :: pipeline (ex,
    [ i=0,max=n] () mutable -> optional<int> {
      if  (i<max) return i++;
      else return {};
    },
    []( int  x)  -> double { return x*x; },
    []( double x) { return 1/x;  },
    []( double x) { cout << x << "\n";  }
  );
}
```

Nested pipelines

- **Pipelines** may be **nested**.
- An **inner pipeline**:
  - Does not take an execution policy.
  - All stages are transformers (no generator).
  - The last stage must also produce values.
- The **inner pipeline** uses the same execution policy than the outer pipeline.

Nested pipelines: Image processing

```
void process(std::istream & in_file, std::ostream & out_file) {
  grppi::parallel_execution_native ex;
```

Nested pipelines: Image processing

```
void process(std::istream & in_file, std::ostream & out_file) {
  grppi::parallel_execution_native ex;
  grppi::pipeline(ex,
```

Nested pipelines: Image processing

```
void process(std::istream & in_file, std::ostream & out_file) {
  grppi::parallel_execution_native ex;
  grppi::pipeline(ex,
    [& in_file ]() -> optional<frame> {
      frame f = read_frame(file);
      if (! file ) return {};
      return f;
    },
```

Nested pipelines: Image processing

```
void process(std::istream & in_file, std::ostream & out_file) {
  grppi::parallel_execution_native ex;
  grppi::pipeline(ex,
    [& in_file]() -> optional<frame> {
      frame f = read_frame(file);
      if (! file ) return {};
      return f;
    },
    pipeline(
      [](const frame & f) { return filter (f); },
      [](const frame & f) { return gray_scale(f); },
    ),
```

Nested pipelines: Image processing

```
void process(std::istream & in_file, std::ostream & out_file) {
  grppi::parallel_execution_native ex;
  grppi::pipeline(ex,
    [& in_file ]() -> optional<frame> {
      frame f = read_frame(file);
      if (! file ) return {};
      return f;
    },
    pipeline(
      []( const frame & f) { return filter (f); },
      []( const frame & f) { return gray_scale(f); },
    },
    [& out_file ]( const frame & f) { write_frame(out_file, f); }
  );
}
```

Piecewise pipelines: Image processing

```
void process(std::istream & in_file, std::ostream & out_file) {
```

Piecewise pipelines: Image processing

```
void process(std::istream & in_file , std::ostream & out_file) {
  auto reader = [& in_file ]() -> optional<frame> {
    frame f = read_frame(file);
    if (! file ) return {};
    return f;
  };
```

Piecewise pipelines: Image processing

```
void process(std::istream & in_file , std::ostream & out_file) {
  auto reader = [& in_file ]() -> optional<frame> {
    frame f = read_frame(file);
    if (! file ) return {};
    return f;
  };
  auto transformer = pipeline(
    []( const frame & f) { return filter (f); },
    []( const frame & f) { return gray_scale(f); },
  };
```

Piecewise pipelines: Image processing

```
void process(std::istream & in_file, std::ostream & out_file) {
  auto reader = [&in_file]() -> optional<frame> {
    frame f = read_frame(file);
    if (! file) return {};
    return f;
  };
  auto transformer = pipeline(
    [](const frame & f) { return filter(f); },
    [](const frame & f) { return gray_scale(f); },
  );
  auto writer = [&out_file](const frame & f) { write_frame(out_file,
      f); }
```

Piecewise pipelines: Image processing

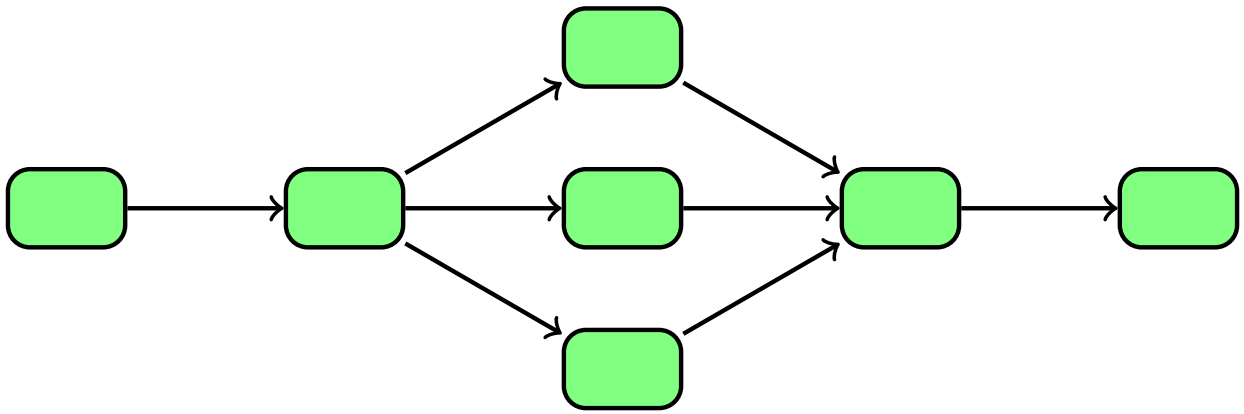
```
void process(std::istream & in_file, std::ostream & out_file) {
  auto reader = [&in_file]() -> optional<frame> {
    frame f = read_frame(file);
    if (! file ) return {};
    return f;
  };
  auto transformer = pipeline(
    [](const frame & f) { return filter (f); },
    [](const frame & f) { return gray_scale(f); },
  );
  auto writer = [&out_file](const frame & f) { write_frame(out_file,
      f); }

  grppi:parallel_execution_native ex;
  grppi::pipeline(ex, reader, transformer, writer);
}
```

5 What can else can I do?

- Controlling execution
- Pipelines
- Farm of tasks
- Controlling the buffering
- Filter

Farm pattern

- A **farm** is a streaming pattern applicable to a stage in a **pipeline**, providing multiple tasks to process data items from a data stream.
  - A **farm** has an associated **cardinality** which is the number of parallel tasks used to serve the stage.
  - Each task in a **farm** runs a **transformer** for each data item it receives.

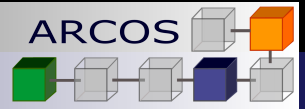


Farms in pipelines: Improving a video

```
template <typename Execution>
void run_pipe(const Execution & ex,
     std::ifstream & filein , std::ofstream & fileout )
{
  grppi::pipeline(ex,
    [& filein ] () -> optional<frame> {
      frame f = read_frame(filein);
      if (! filein ) retrun {};
      return f;
    },
    farm(4, []( const frame & f) { return improve(f); },
    [& fileout ] (const frame & f) { write_frame(f); }
  );
}
```

Piecewise farms: Improving a video

```
template <typename Execution>
void run_pipe(const Execution & ex,
     std::ifstream & filein, std::ofstream & fileout)
{
  auto improver = farm(4,
    [](const frame & f) { return improve(f); });

  grppi::pipeline(ex,
    [& filein] () -> optional<frame> {
      frame f = read_frame(filein);
      if (! filein) retrun {};
      return f;
    },
    improver,
    [& fileout] (const frame & f) { write_frame(f); }
  );
}
```



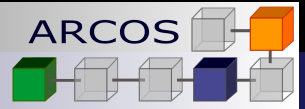
5 What can else can I do?
- Controlling execution
- Pipelines
- Farm of tasks
- Controlling the buffering
- Filter

Ordering

- Signals if pipeline items must be consumed in the same order they were produced.
  - Do they need to be *time-stamped*?
- Default is **ordered**.
- **API**
  - **ex.enable_ordering()**
  - **ex.disable_ordering()**
  - **bool** o = **ex.is_ordered()**

Queueing properties

- Some policies (**native** and **omp**) use queues to communicate pipeline stages.
- **Properties**:
  - **Queue size**: Buffer size of the queue.
  - **Mode**: *blocking* versus *lock-free*.
- **API**
  - **ex.set_queue_attributes(100, mode::blocking)**



5 What can else can I do?

- Controlling execution
- Pipelines
- Farm of tasks
- Controlling the buffering
- Filter

Filter pattern

- A **filter** pattern discards (or keeps) the data items from a data stream based on the outcome of a predicate.
- This pattern can be used only as a stage of a **pipeline**.

Filter pattern

- A **filter** pattern discards (or keeps) the data items from a data stream based on the outcome of a predicate.
- This pattern can be used only as a stage of a **pipeline**.

- **Alternatives**:
  - **Keep**: Only data items satisfying the predicate are sent to the next stage.
  - **Discard**: Only data items **not satisfying** the predicate are sent to the next stage.

Filtering in: Print primes

```
bool is_prime(int n);

template <typename Execution>
void print_primes(const Execution & ex, int n)
{
  grppi::pipeline(exec,
    [i=0,max=n]() mutable -> optional<int> {
      if (i<=n) return i++;
      else return {};
    },
    grppi::keep(is_prime),
    [](int x) { cout << x << "\n"; }
  );
}
```

Filtering out: Discard words

```
template <typename Execution>
void print_primes(const Execution & ex, std::istream & is)
{
  grppi :: pipeline (exec,
    [& file ]()  -> optional<string> {
      string  word;
      file  >> word;
      if  (! file ) { return {}; }
      else { return word; }
    },
    grppi :: discard ([]( std :: string  w) { return w.length() < 4; },
     []( std :: string  w) { cout << x << "\n"; }
  );
}
```

1 Times have changed

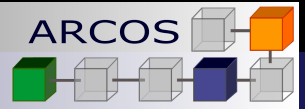
2 What do you do with multicore?

3 Parallelism in C++17

4 After C++20: Executors

5 What can else can I do?

6 Summary



Summary

- We live in a parallel world!

Summary

- We live in a parallel world!
- Many portable concurrency primitives since C++11.
  - Low level and good for solving the througput challenge.

Summary

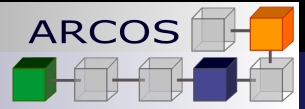
- We live in a parallel world!
- Many portable concurrency primitives since C++11.
  - Low level and good for solving the througput challenge.
- C++17 brings easy parallelism to the STL.
  - Mostly data parallelism.

Summary

- We live in a parallel world!
- Many portable concurrency primitives since C++11.
  - Low level and good for solving the througput challenge.
- C++17 brings easy parallelism to the STL.
  - Mostly data parallelism.
- C++23 (hopefully) might bring executors.

Summary

- We live in a parallel world!
- Many portable concurrency primitives since C++11.
  - Low level and good for solving the througput challenge.
- C++17 brings easy parallelism to the STL.
  - Mostly data parallelism.
- C++23 (hopefully) might bring executors.
- Stream parallelism still to be solved.



C++ prgramming in a parallel world

CPP Europe
Bucharest 2020

J. Daniel Garcia

ARCOS Group
University Carlos III of Madrid
Spain

February 25th, 2020