CERN School of Computing

Lecture IV

Patterns for Parallel Software Development

Outline of This Lecture

The Goals:

1) *Understand a few basic patterns of sequential algorithms*
2) *Know how to map these onto parallel concepts*
3) *Understand how these scale*

What is a Pattern?

Software design pattern

General, **reusable** solution to a **commonly occurring problem** in a given context in software design

Parallel pattern

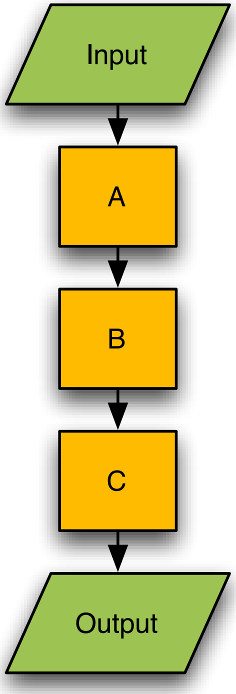
Recurring combination of **task distribution** and **data access** that solves a specific problem in parallel algorithm design

Serial Control Flow Patterns

- Before starting with parallelism let’s look at what we know about the serial case
- We will have a look at the following ones:
  - **Sequence**
  - **Selection**
  - **Iteration**
- These are all simple concepts, but the vocabulary is important!

Sequence

- A **sequence** is an ordered list of tasks/commands to be carried out in a given **order**
  - The exact dependencies of the commands do not matter
  - Side-effects do not matter
  - There is only **one task** executed at a time
  - The tasks are executed as defined

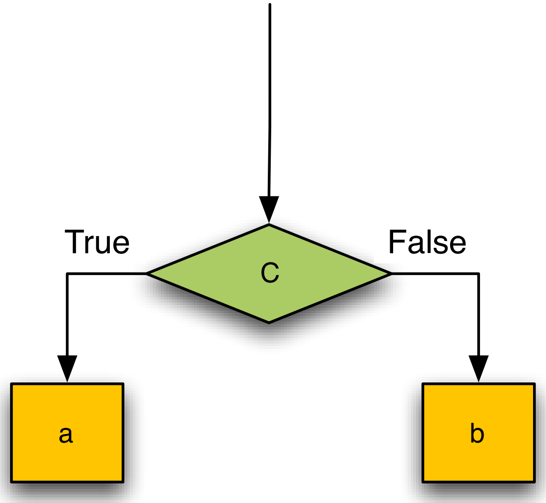


Note that

The compiler and the CPU may re-order instructions if they think it optimises runtime

Selection

- In a **selection**
    - The commands a and b **depend on decision** of c
    - **Always only one** of the two sides is being executed



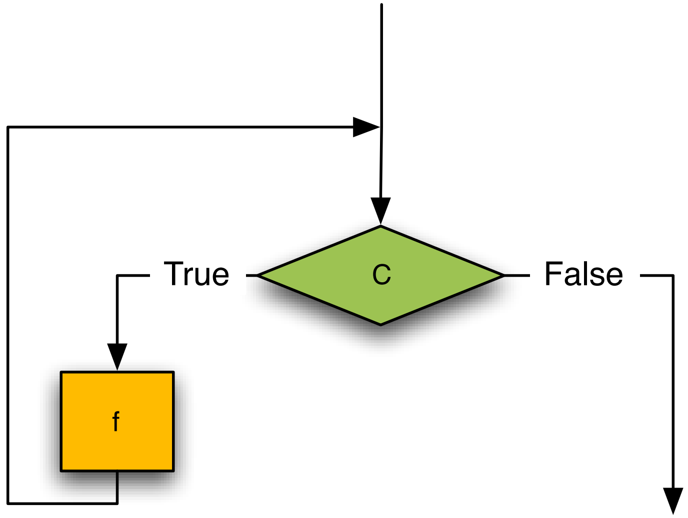
The «if» statement

The CPU may apply speculative execution, but it always takes care of sanity

Iteration

- In an **iteration** a certain function f is executed as long as a certain condition c is true.
  - This is the famous *while* loop

```
while ( c ) {
   f;
}
```



Iteration II

- **How do condition and function depend on each other?**
  - There must be some dependency, otherwise it is an infinite loop
- Sometimes the dependency is trivial and can be re-formulated as a *`for`* loop (a.k.a. counted loop)

```
i = 0;
while ( i < n ) {
   f;
   ++i;
}
```

→

```
for (i = 0; i < n; ++i ) {
   f;
}
```

- In serial code this is mainly just syntactic sugar
  - However, it gives some nice hints to the compiler

Iteration III

- The serial iteration pattern might seem trivially parallelisable but…
  - Beware of **dependencies**!
- **Do multiple iterations depend on each other?**
  - Loop-carried dependency
- Different kinds of dependencies translate to **different parallelisation possibilities**

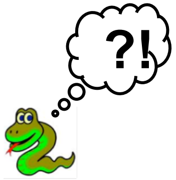
Iteration IV

```
void doIt( int n, double x[], int a[], int b[], int c[] ) {

  for (int i = 0; i < n; ++i) {
    x[ a[i] ] = x[ a[i] ] * x[ b[i] ] * x[ c[i] ];
  }

}
```

- Any chance of parallelising this?
- What are the obstacles?
    - i.e. what are the dependencies?

Modern Syntax: An Interlude

- C++ is ever improving with new standards (C++11, C++14, C++17, C++20, C++23)
- Two (not so) recent additions are:
  - `auto var = retrieveSomeObject();`
  - `for (auto & element : myCollection)`



- `auto` : do not specify the type, the compiler finds it out at compile time. Useful to avoid tedious typing also detrimental for readability of the code!
- **Range-based loops**: build a loop with a concise syntax!

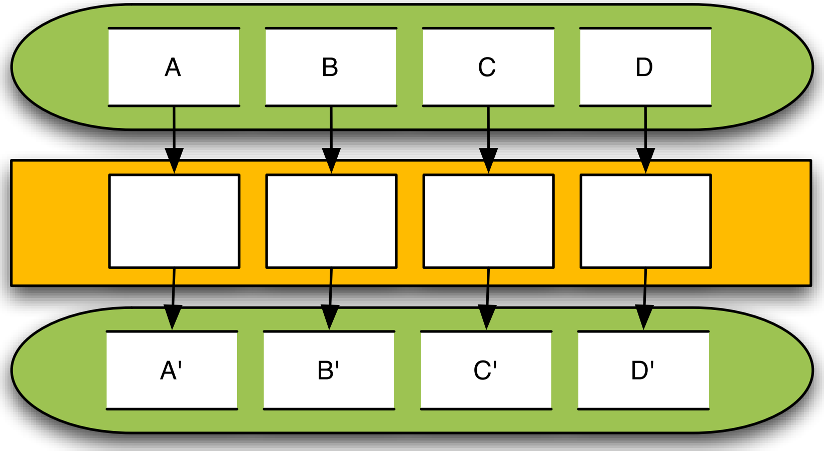
Take advantage of this! ☺

Parallel Patterns

- After reminding ourselves about serial control patterns, let’s have a look at a few parallel patterns
  - Can help you structure your parallel program
- The serial **iteration** pattern has many parallel offsprings
  - **Map**
  - **Partition**
  - **Reduce**
  - **Scan**
- Other useful patterns
  - **Pipeline**
  - **Superscalar Sequences**

Map

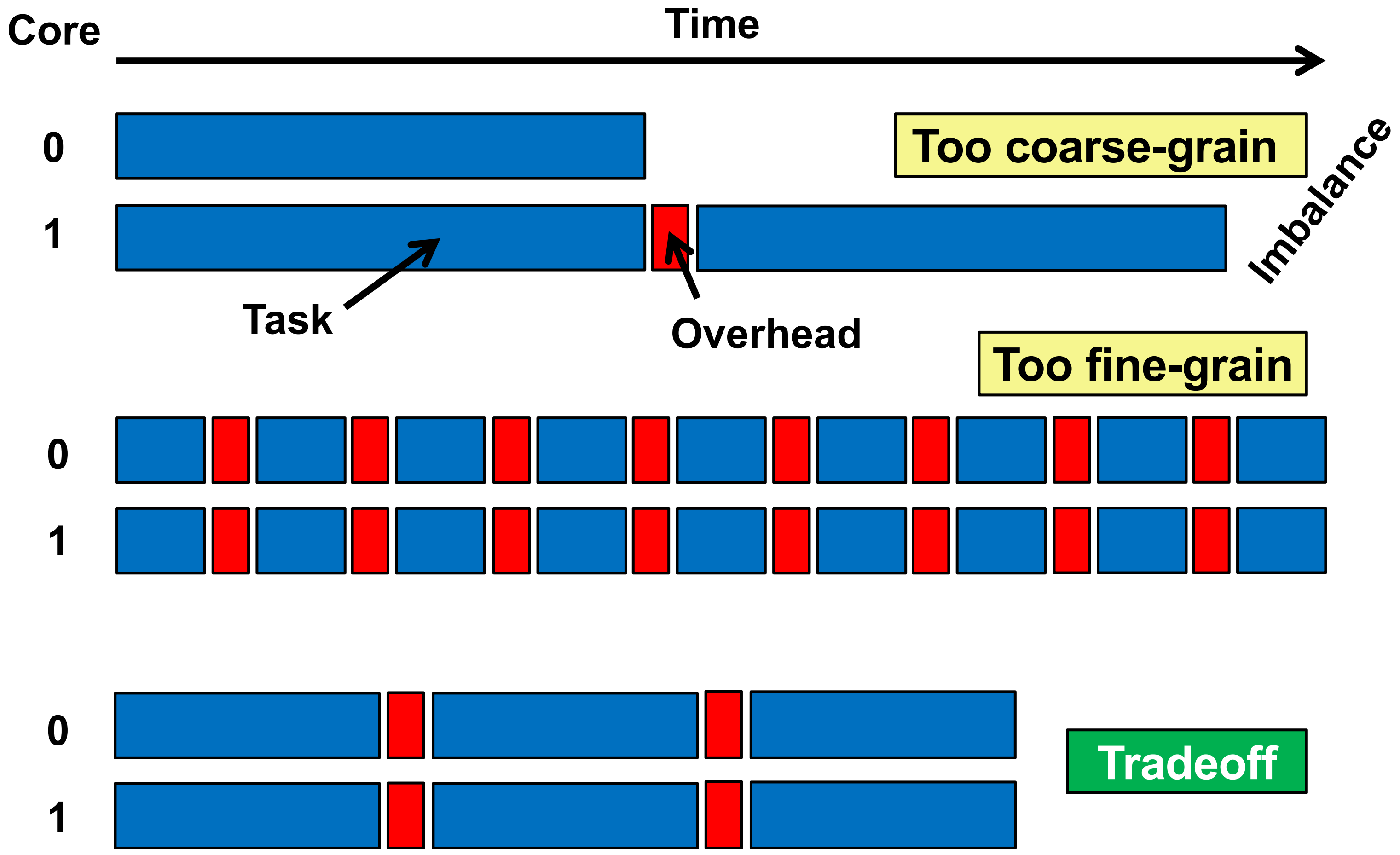
- The **map** is the most trivial parallel extension of the serial iteration
  - Apply the same function `f` on multiple elements of a collection in parallel
  - We **hide the loop**!
- **Requirements**:
  - No loop-carried dependency
  - Function `f` is pure, i.e. without side-effects
- **Scaling:** n (linear w.r.t. the number of elements in the collection)



Partition

- The map pattern helps when parallelising on collections
- However, sometimes it is useful to treat multiple items together
  - E.g. for the combination of multithreading and vectorisation
  - Multi-level parallelism!
- **Partitioning** allows for a custom split of the collection into subcollections or *chunks*
- A variant of partitioning is called **geometric decomposition**
  - Update of a partition needs data from other partitions
  - Might require synchronisation

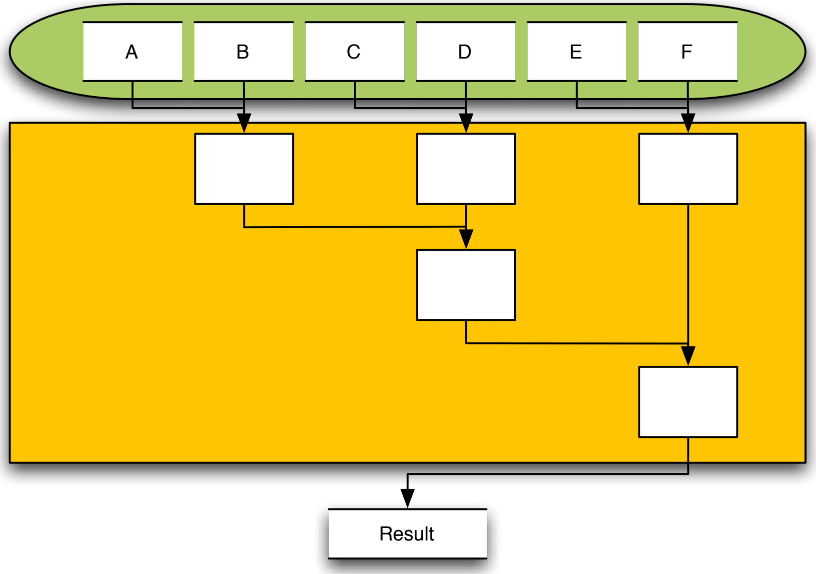
Granularity



Reduce

- A **reduction** combines the elements of a collection into a single result using a **combiner function**
- **Requirements**:
  - No loop-carried dependency apart from the combined result
  - Combiner function is **associative**
    - Be careful with floating-point operations!
  - Having a commutative function is beneficial

Reduce II



- **Speedup:** n / log n
- Counters are a typical example for reduction input
- Before coming to a real example, let's have a look at modern C++ again…

Interlude – Lambdas

- Lambda expressions are anonymous functions and can be assigned to the `std::function` type
- They can be passed as parameters as if they were regular variables
- When defined, they can capture a specific set of variables (or all)
- Once they have been defined, they can be passed to functions like `std::for_each` or TBB's `parallel_for`

```
std::function< double (double, double) >
  f = [ ] (double a, double b) { return a + b; };
std::cout << f ( 23.0, 24.0 );
```

Interlude – Lambdas II

- Using the C++ `auto` keyword simplifies the syntax, but does not change the behavior

```
auto f = [ ] ( double a, double b ) { return a + b; };
```

- Capture the variable `globalOffset` as a reference and use it in the computation

```
auto f = [ &globalOffset ] ( double a, double b )
          { return a + b + globalOffset; };
```

- Capture all variables defined in the current scope by value

```
auto f = [ = ] ( double a, double b )
          { return a + b + globalOffset; };
```

- Can you think of the difference in behavior when using capture-by-value instead of capture-by-reference?

Reduce III

- Libraries like Intel's Threading Building Blocks (TBB) provide already all ingredients for standard patterns like reduce:

```
int sum = tbb::parallel_reduce(
  // The input array, which will be partitioned automatically
  tbb::blocked_range<int*>(array, array + size),
  // Identity value for the sum reduction
  0,
  // Lambda that returns the sum of all elements in a partition
  [](const tbb::blocked_range<int*>& r, int v) {
    for (auto i = r.begin(); i != r.end(); ++i) v += *i;
    return v;
  },
  // Reduction operation that combines the per-partition sums
  [](int x, int y) { return x+y; }
);
```

Map and Reduce Combined

- Usually **map** and **reduce** go hand in hand:
  - A **function** being applied to single elements
  - The results are then passed to a **combiner function**
- A concrete example:
  - Count the number of times a certain word appears in a text
- Solution:
  - **Partition**: Split the text in equally-sized chunks
  - **Map:** Do the word count
  - **Reduce:** Add the counts
- Various map/reduce frameworks at your disposal!

The Power of Map-Reduce

- The combination of the Map and Reduce patterns has been extremely successful in **massive distributed data processing**
- A little bit of history…
  - 2004: Google publishes the MapReduce paper
  - 2006: Hadoop is released, inspired by MR
- Nowadays, MR is **behind every click** on popular web sites or services
  - Facebook, Twitter, Yahoo, …
  - Analytics to predict user interests, target ads, show recommendations, … and many more
  - Robust, fault tolerant
  - Scale to crunch large datasets

Map-Reduce and Functional Chains

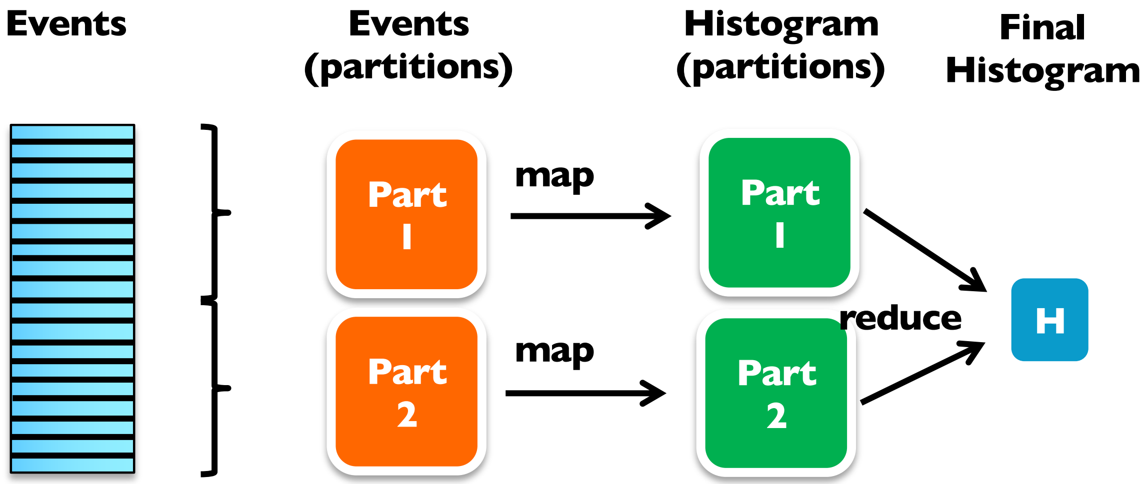
- Map and reduce were born in **functional programming**
  - Declare **what** you want to do, not how
  - No side-effects
- **High-level view**, based on two main concepts:
  - Data is organised in **collections** of elements
  - We apply functions to those elements, possibly in a chain

```
histo = events.map(fillHist).reduce(mergeHist)
```

- Implemented by frameworks like Spark and ROOT’s RDataFrame
  - No need to manage parallelisation, just **think about opportunities for parallelism**!

Map-Reduce and Functional Chains II

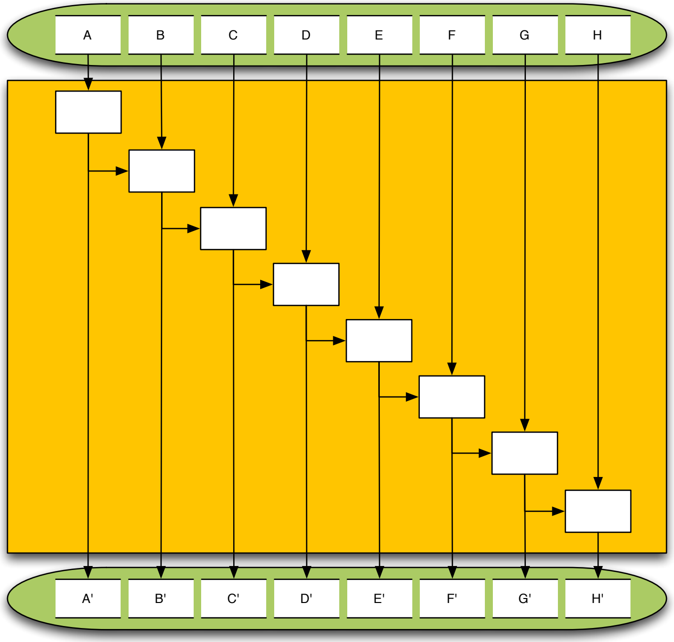
- Implementation responsible for producing a **parallel execution plan**
  - Where are the data?
  - What resources are available?
  - What optimisations can be applied?

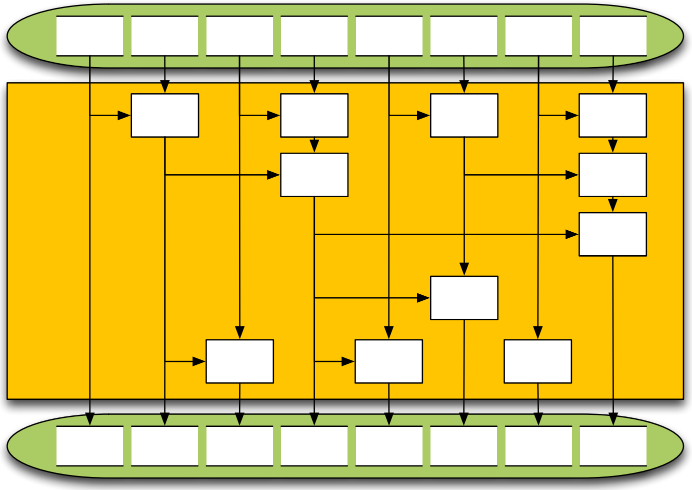


Scan

- **Scan** is another offspring of the iteration pattern with more relaxed boundary conditions
- **Requirements**:
  - Result of element `n` depends on `n-1`
  - Successor function is **associative**

Scan II





Serial version

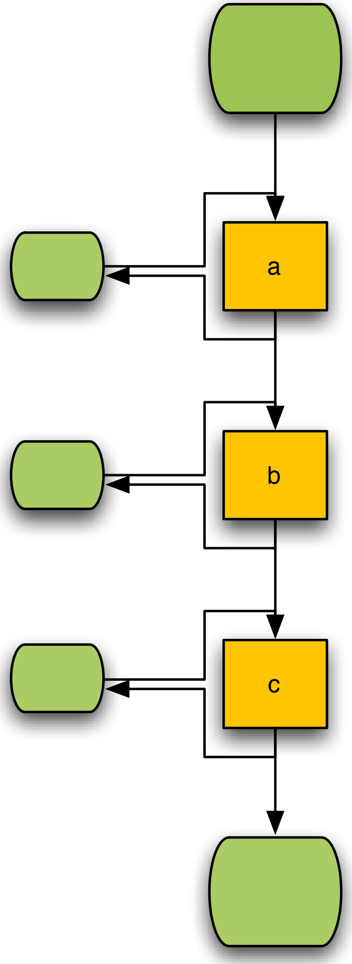
Parallel version

Scan III

- **Scan** is another offspring of the iteration pattern with more relaxed boundary conditions
- **Requirements**:
  - Result of element `n` depends on `n-1`
  - Successor function is **associative**
- Already a non-trivial implementation necessary
- **Speedup**: very limited
  - At most n / log n
  - Number of instructions required is worse (up to x2)

Pipeline

- The **pipeline** pattern is the good old assembly line
    - Work split into a sequence of operations with a **producer-consumer relationship**
    - Work items go from one stage to the next
    - The order of steps is important
    - Different operations on different items are independent
    - Stages can be serial or parallel (accept one or more items simultaneously)
- More complex cases can have a directed acyclic graph instead of a purely linear setup
- The **speedup** of a pipeline is given by **Amdahl's Law**

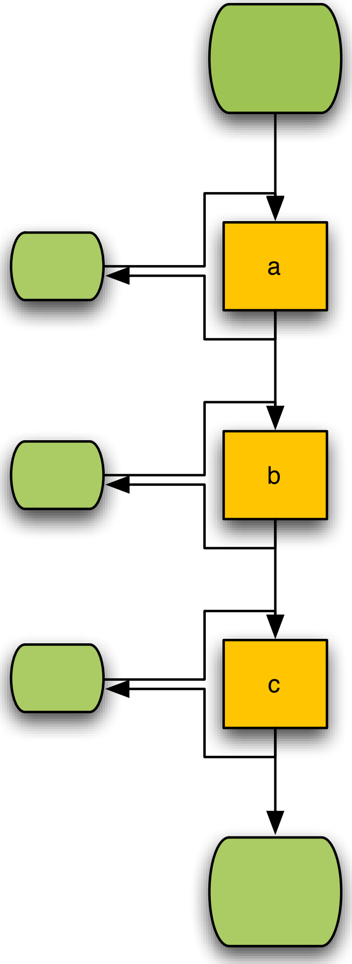


Pipeline II

- Intel’s TBB offers a feature for implementing a pipeline too:

```
parallel_pipeline( max_number_of_live_tokens,
                   make_filter<void,I1> (mode0, a) &
                   make_filter<I1,I2> (mode1, b) &
                   make_filter<I2,void> (mode2, c)
);
```

```
parallel
serial_in_order
serial_out_of_order
```



Pipeline III

```
float RootMeanSquare( float* first, float* last, int n ) {
  float sum = 0;
  parallel_pipeline(16,
    make_filter<void,float*>(
      filter::serial_in_order,
      [&](flow_control& fc) -> float* {
        if ( first < last ) {
          return first++;
        } else {
          fc.stop();
          return nullptr;
        }
      }
    ) &
    make_filter<float*,float>(
      filter::parallel,
      [](float* p) { return (*p)*(*p); }
    ) &
    make_filter<float,void>(
      filter::serial_in_order,
      [&](float x) { sum += x; }
    )
  );
  return sqrt(sum / n);
}
```

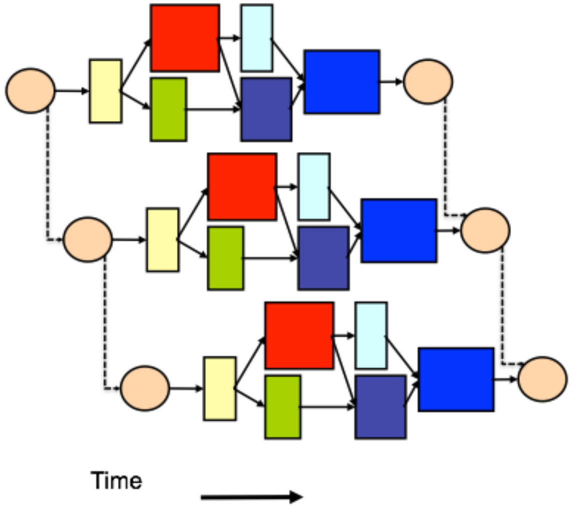
Step 1 handles the data stream

Step 2 can run in parallel with itself

Step 3 is not thread-safe

Superscalar Sequences

- Split work into several tasks and define their data dependencies
- Let a task scheduler do the rest
- **Pattern followed by concurrent HEP data processing frameworks**



- Assumption of this model is that there are **no hidden data dependencies** and **no side-effects** unknown to the scheduler
  - Let's have a look at these assumptions…

Hidden Data Dependencies

```
std::atomic_bool doit(false);

void task1() {
  …
  if (doit)) {
    eventstore.put(fancystuff);
  }
}
void task2() {
  doit = true;
}
```

Thread-safe boolean variable

- Content of the event store depends on the execution order
- Thread-safe objects don’t help at all
- It is a pure logic flaw

Side Effects

- Triggered when a computation modifies some **shared state** outside of its local environment
  - e.g. a global variable
- They are a major obstacle for parallelism
  - Watch out for them when applying your parallel patterns!
- In general, every non thread-safe resource is an issue
- Remember from previous lectures:
  - Side-effect free resources are the ideal solution
  - If not possible, tell the scheduler about what you need and **“reserve”** what is unsafe

Take-Away Messages

- There exist design patterns to help you parallelising your programs
  - Check if you can **reuse** them!
- They all have their origin in serial patterns, but add constraints to the operations allowed
- **Map-Reduce** is a very successful pattern, used every day for distributed processing of large amounts of data
- High-level features like C++ lambdas, the TBB library or the Spark framework make it easier for you to get started with these patterns