Concepts and Models of Parallel and Data-centric Programming

Shared Memory V

Lecture, Summer 2020

Dr. Christian Terboven <terboven@itc.rwth-aachen.de>

i12 High Performance Computing

RWTH AACHEN UNIVERSITY

Outline

Futures

Futures / 1

- A future implements an (asynchronous) one-off event
  - Waiting for the occurrence of the event is possible
  - The event is typically the completion of an asynchronous task with return value

- Class `std::future` and `std::shared_future`
  - `std::future`: unique reference to the event
    - Move semantics
  - `std::shared_future`: multiple references from different threads to the same event possible (including multiple threads waiting)
    - Copy semantics
  - Calling `get()` will block until the (return) value has been computed
  - Defined in header `<future>`
  - Reference: https://en.cppreference.com/w/cpp/thread/future

Futures / 2

- Class `std::promise`
  - Defined in header `<future>`
  - Implementation detail: promise to set the value in the future
  - Reference: https://en.cppreference.com/w/cpp/thread/promise

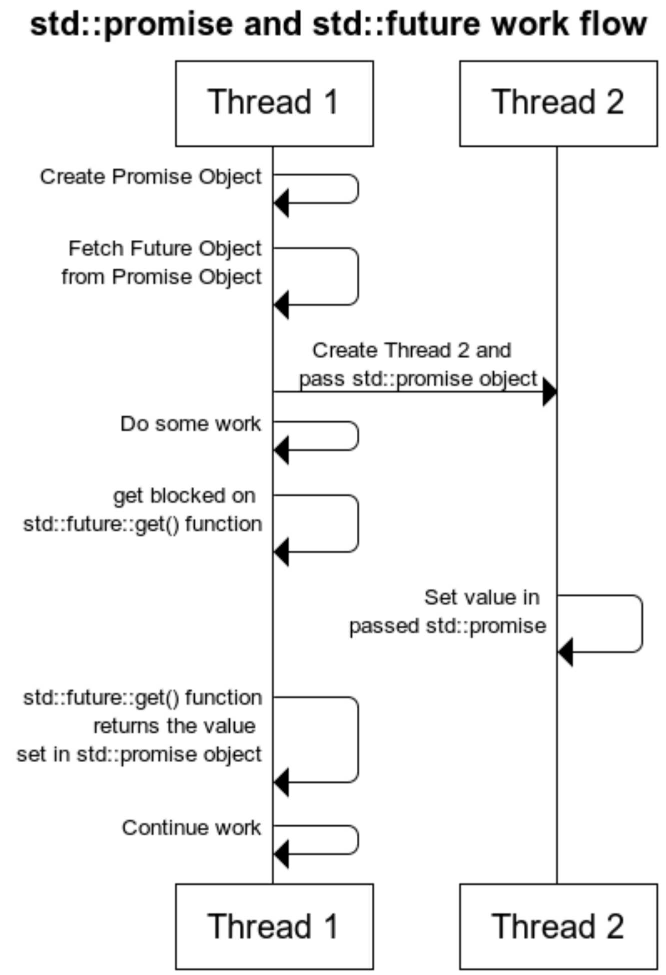
- Function `std::async`
  - Runs a function asynchronously, returns a `std::future`
  - Reference: https://en.cppreference.com/w/cpp/thread/async

Futures / 3

- Illustration of an asynchronous execution with `std::future`

```
// let T be some valid typename
std::promise<T> promiseObj;
std::future<T> futureObj =
    promiseObj.get_future();
std::thread th( f , &promiseObj);
th.join();
```

- Remark: in many cases, the `std::promise` object is not used explicitly



Example: QuickSort

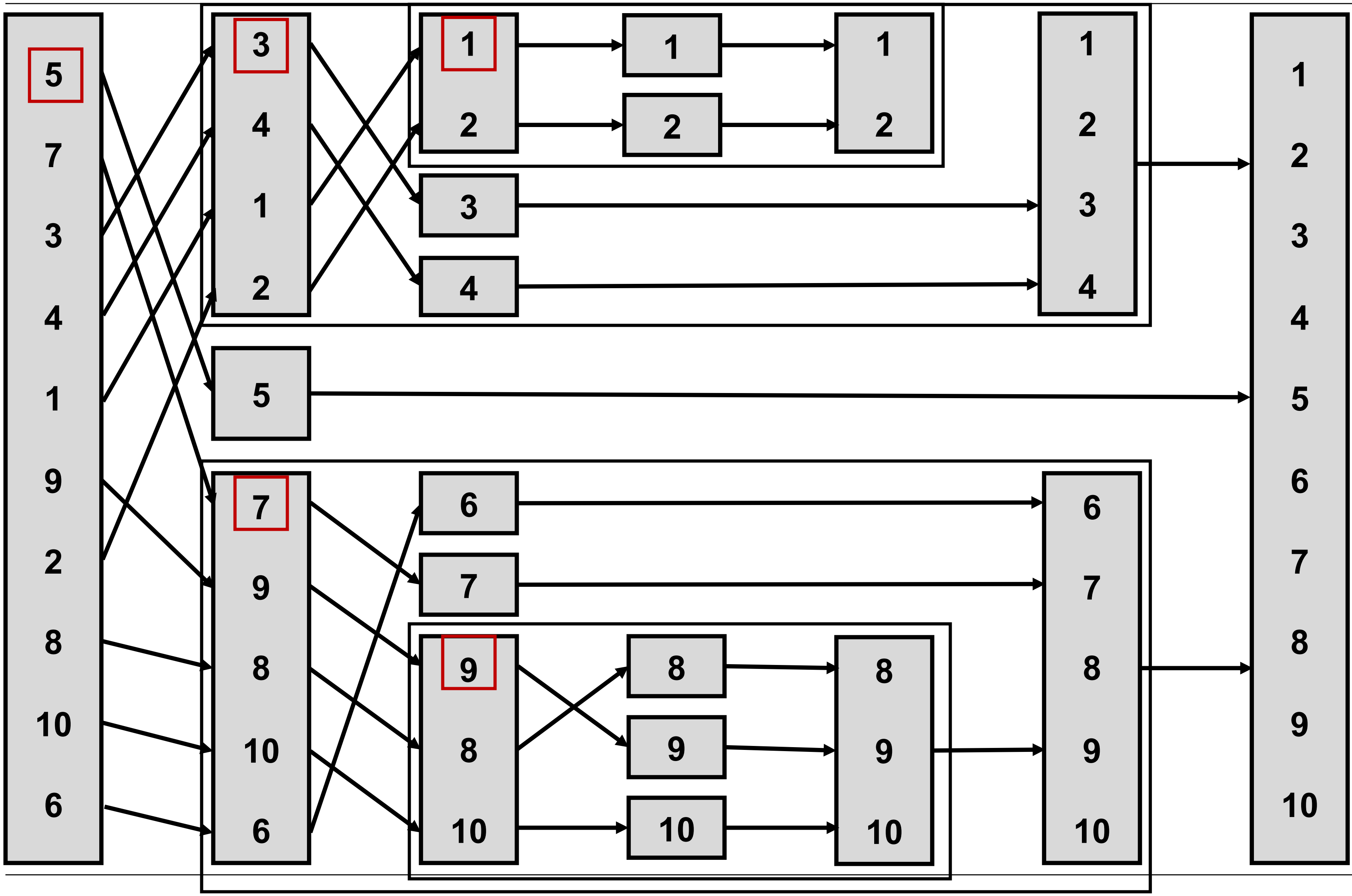
Motivation

- FP-style parallel programming
  - Functional Programming (FP) in this context: the result of a function call depends solely on the parameters to that function
    - does not depend on any external state
    - trivial example: math routines from the standard library, such as `sin()`

- A *pure* function does not modify any external state; its effects are entirely limited to the return value

- Concurrency in functional programming may eliminate data races

FP-style QuickSort / 1



FP-style QuickSort / 2

```
template<typename T>
std::list<t> seq_qsort(std::list<T> input)
{
   if(input.empty())
      return input;

   std::list<T> result;
   result.splice(result.begin(), input, input.begin());
   T const& pivot = *result.begin();

   auto divide_point = std::partition(input.begin(), input.end(),
           [&](T const& t){return t < pivot;});
   std::list<T> lower_part;
   lower_part.splice(lower_part.end(), input, input.begin(), divide_point);
   auto new_lower(seq_qsort(std::move(lower_part)));
   auto new_higher(seq_qsort(std::move(input)));
   result.splice(result.end(), new_higher);
   result.splice(result.begin(), new_lower);
   return result;
}
```

FP-style QuickSort / 3

```
template<typename T>
std::list<t> seq_qsort(std::list<T> input)
{
   if(input.empty())
      return input;

   std::list<T> result;
   result.splice(result.begin(), input, input.begin());
   T const& pivot = *result.begin();

   auto divide_point = std::partition(input.begin(), input.end(),
           [&](T const& t){return t < pivot;});
   std::list<T> lower_part;
   lower_part.splice(lower_part.end(), input, input.begin(), divide_point);
   auto new_lower(seq_qsort(std::move(lower_part)));

   auto new_higher(seq_qsort(std::move(input)));
   result.splice(result.end(), new_higher);
   result.splice(result.begin(), new_lower);
   return result;
}
```

FP-style QuickSort / 4

```
template<typename T>
std::list<t> par_qsort(std::list<T> input)
{
   if(input.empty())
      return input;

   std::list<T> result;
   result.splice(result.begin(), input, input.begin());
   T const& pivot = *result.begin();

   auto divide_point = std::partition(input.begin(), input.end(),
           [&](T const& t){return t < pivot;});
   std::list<T> lower_part;
   lower_part.splice(lower_part.end(), input, input.begin(), divide_point);
   std::future<std::list<T> > new_lower(
            std::async(&par_qsort<T>, std::move(lower_part)));
   auto new_higher(par_qsort(std::move(input)));
   result.splice(result.end(), new_higher);
   result.splice(result.begin(), new_lower);
   return result;
}
```

FP-style QuickSort / 5

- Remarks on the simple sequential implementation:
  - returns a list by-value (`std::sort` performs an in place sort)
  - interface is FP-style, implementation is partly imperative for efficiency reasons
  - `std::list`: first element is taken as pivot element
    - may result in sub-optimal sort
    - reason: avoid list traversal

- Bad parallelization: `std::partition` does a lot of work and is still sequential

- Unclear scalability: execution depends on quality of the implementation
  - how many threads will be created?