DLAF - Distributed Linear Algebra with Futures

real-world use-cases

Advanced C++

Alberto Invernizzi (alberto.invernizzi@cscs.ch), CSCS

October 12, 2021

HPC and Linear Algebra

The de-facto standard library for distributed linear algebra is ScaLAPACK, a library that has been developed in 1995, when supercomputers were based on nodes which had a single CPU core.

Since then, node architectures have evolved (e.g. multicores and multi-GPU), and new programming paradigms are necessary.

Distributed Linear Algebra with Futures - DLAF

DLAF uses a **task-based approach** aiming to reduce the amount of synchronization necessary between distinct routines.

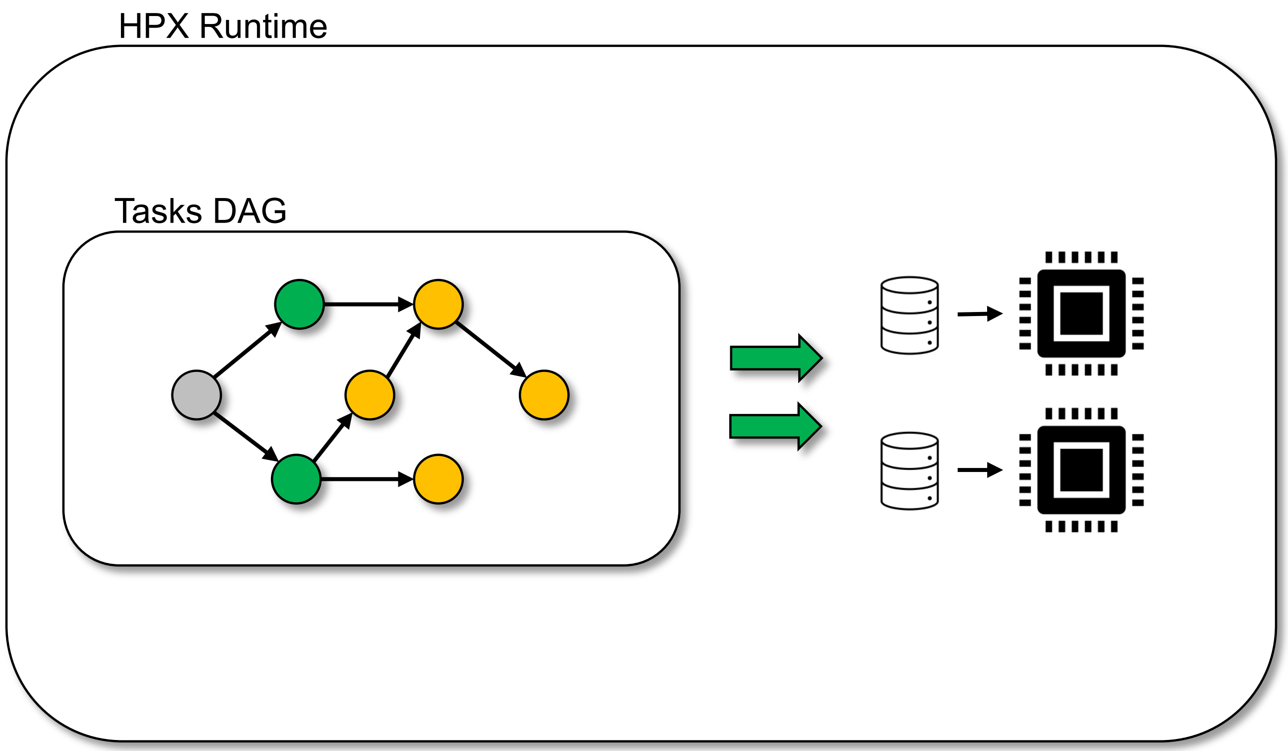
The task-based approach is possible thanks to HPX, a dependency of DLAF for which represents the backbone.

What is *HPX*?

HPX is a C++ Standard Library for Concurrency and Parallelism. It implements all of the corresponding facilities as defined by the C++ Standard. Additionally, in *HPX* we implement functionalities proposed as part of the ongoing C++ standardization process. We also extend the C++ Standard APIs to the distributed case. *HPX* is developed by the STE||AR group (see People).

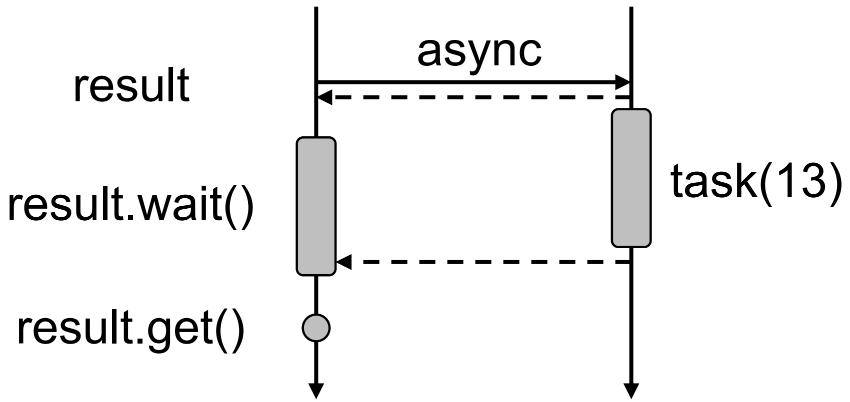
https://github.com/STEllAR-GROUP/hpx

HPX overview

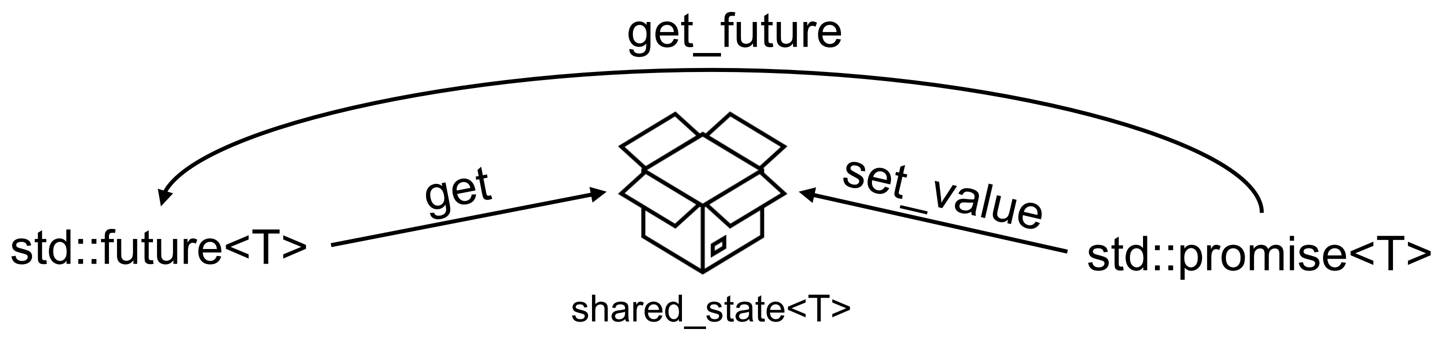


Task: a trivial example

```cpp
float task(int input);

// ...

std::future<float> result = std::async(task, 26);

result.wait();
float output = result.get();
```

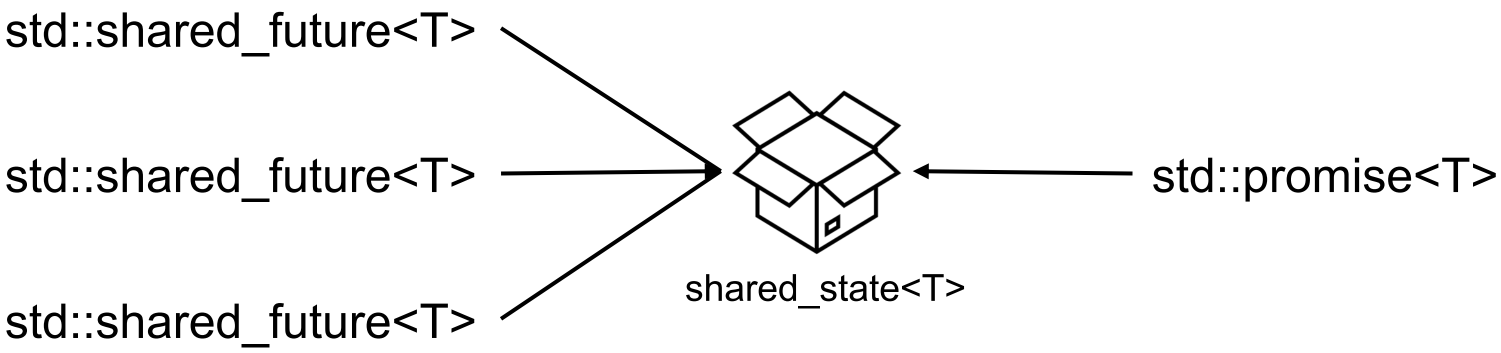


std::promise + std::future




```cpp
#include <future>

int main() {
  std::promise<int> promise;
  std::future<int>  future = promise.get_future();

  promise.set_value(26);

  int value = future.get();
}
```

… and std::shared_future



```cpp
#include <future>

float task(int value) {
  return value * 1.3;
}

int main() {
  std::future<float> result = std::async(task, 26);

  result.wait();

  assert(result.valid());
  float output = result.get();
  assert(not result.valid());
}
```

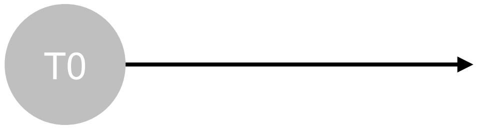
```cpp
#include <future>

float task(int value) {
  return value * 2.33;
}

int main() {
  std::shared_future<float> result = std::async(task, 26);

  result.wait();

  assert(result.valid());
  const float& output = result.get();
  assert(result.valid());
}
```

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

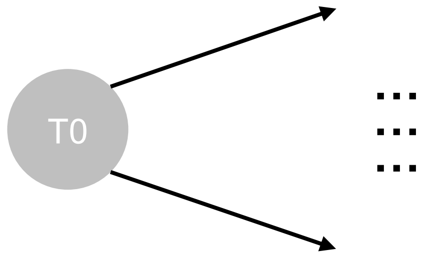
ETH zürich

HPX 101

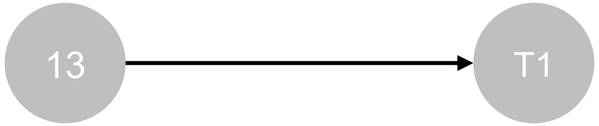
DAG: future



DAG: shared_future



.then

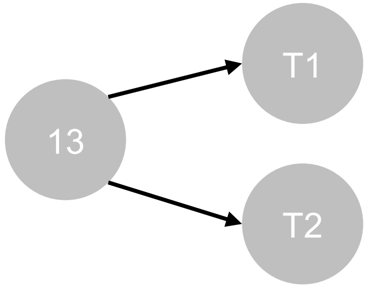



```cpp
#include <hpx/future.hpp>
#include <hpx/hpx_main.hpp>

int main() {
  using namespace hpx;

  future<int> future_value = make_ready_future<int>(13);

  future_value.then([](future<int> value) {});
}
```

It returns another future

Inside here the value is ready to be retrieved

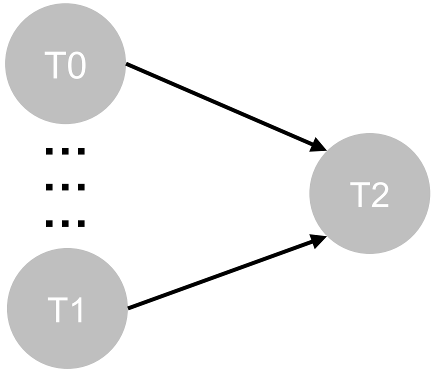
.then




```cpp
#include <hpx/future.hpp>
#include <hpx/hpx_main.hpp>

int main() {
  using namespace hpx;

  shared_future<int> future_value = make_ready_future<int>(13);

  future_value.then([](shared_future<int> value) {});
  future_value.then([](shared_future<int> value) {});
}
```

When will these be executed?

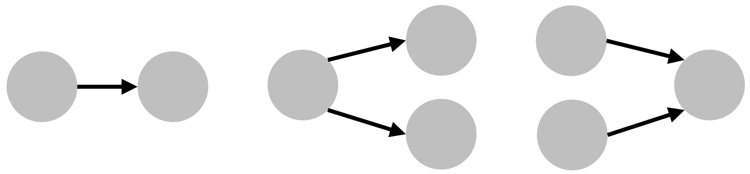
dataflow




```cpp
hpx::future<int>    task0_result = hpx::async([]() { return 26; });
hpx::future<float>  task1_result = hpx::async([]() { return 13.26f; });

hpx::dataflow(
  [](hpx::future<int> data0, hpx::future<> data1) { ... },
  task0_result, task1_result);
```

DAG Grammar

These allow you to describe any DAG



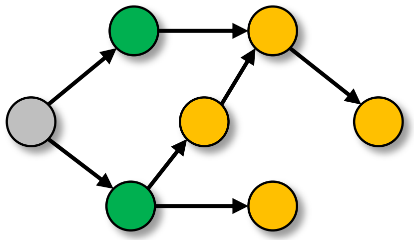
.then

.then
(shared_future)

dataflow

!!! IT’S NOT ABOUT FORK/JOIN !!!

these operations describe **<u>dependencies</u>** between tasks



DLAF

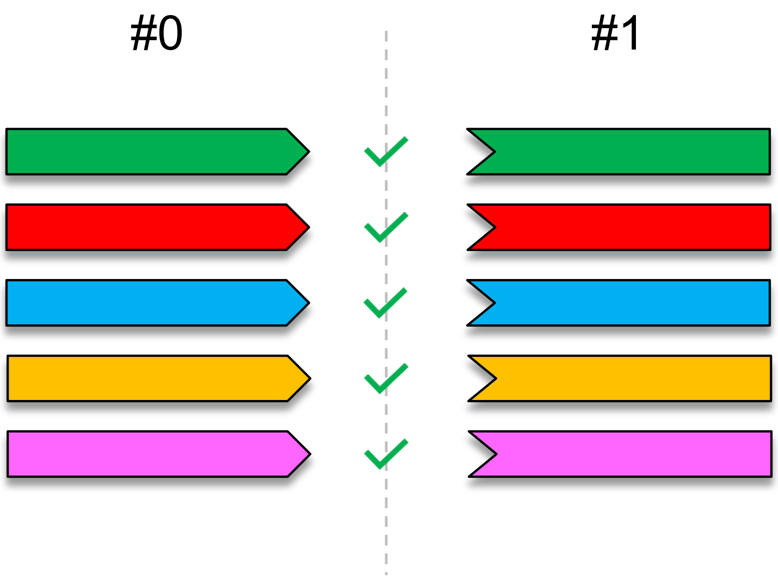
MPI Communication

The D in DLAF stands for DISTRIBUTED, which implies communication between nodes.

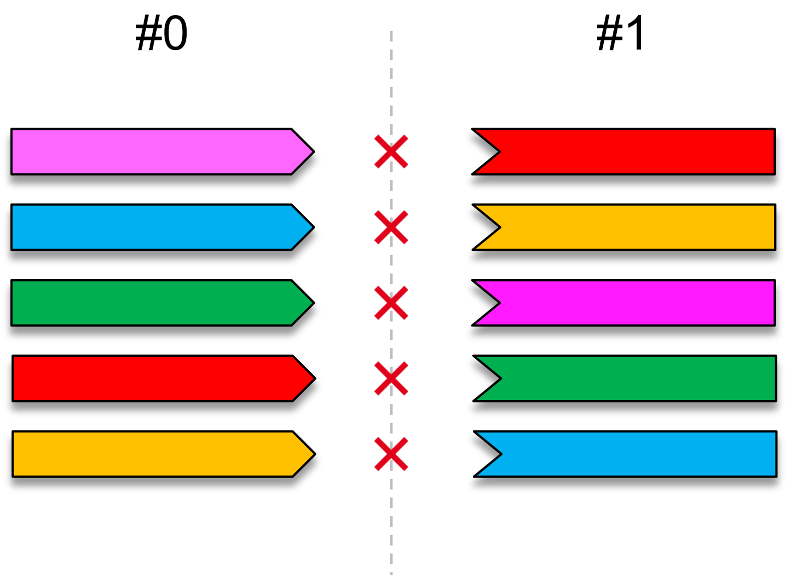
For the communication, our requirement is to use MPI, because DLAF aims at replacing ScaLAPACK in existing Fortran/C/C++ applications.

But using MPI poses a few challenges, especially in a task-based programming model…

Problem: Communication Order



Problem: Communication Order



Pipeline

What about adding a fictitious dependency that describe the ordering requirement?

What do they share?

❌ the data/tile communicated is likely not shared

✅ the communicator does!

Actually, we want to serialize the communications inside the same communicator, because MPI already preserves the independency between different communicators.

So, if we "serialize" the access to the communicator resource, we can impose an order on them!

object and promise ownership

just movable, enforce unique ownership

On destruction it sets the promise by moving in the value owned

Access to the object via ref()

```cpp
template <class T> class PromiseGuard {
public:
  PromiseGuard(T object, hpx::lcos::local::promise<T> next)
      : object_(std::move(object)), promise_(std::move(next)) {}

  PromiseGuard(PromiseGuard &&) = default;
  PromiseGuard &operator=(PromiseGuard &&) = default;

  PromiseGuard(const PromiseGuard &) = delete;
  PromiseGuard &operator=(const PromiseGuard &) = delete;

  ~PromiseGuard() {
    if (promise_.valid())
      promise_.set_value(std::move(object_));
  }

  T &ref() { return object_; }
  const T &ref() const { return object_; };

private:
  T object_;
  hpx::lcos::local::promise<T> promise_;
};
```

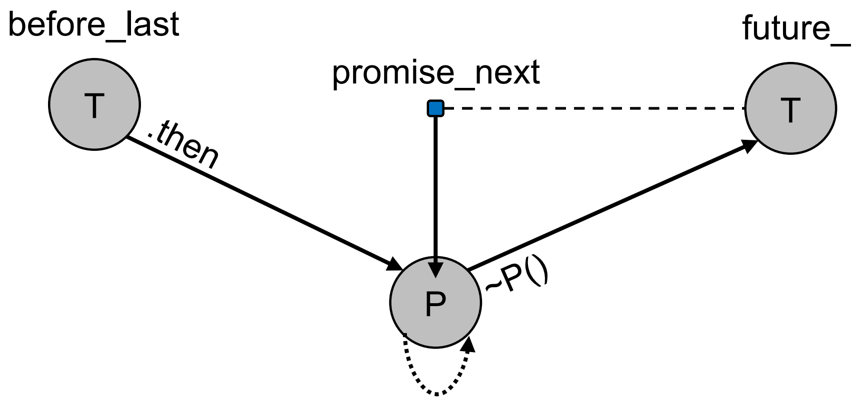
Get the ownership of the resource

The first element in the pipeline is ready

```
template <class T> class Pipeline {
public:
  Pipeline(T object) : future_(hpx::make_ready_future(std::move(object))) {}

  hpx::future<PromiseGuard<T>> operator()() {
    auto before_last = std::move(future_);

    hpx::lcos::local::promise<T> promise_next;
    future_ = promise_next.get_future();

    return before_last.then(
        hpx::launch::sync,
        hpx::unwrapping(
          [promise_next = std::move(promise_next)](T &&object) mutable {
            return PromiseGuard<T>{std::move(object), std::move(promise_next)};
        }));
  }

private:
  hpx::future<T> future_;
};
```

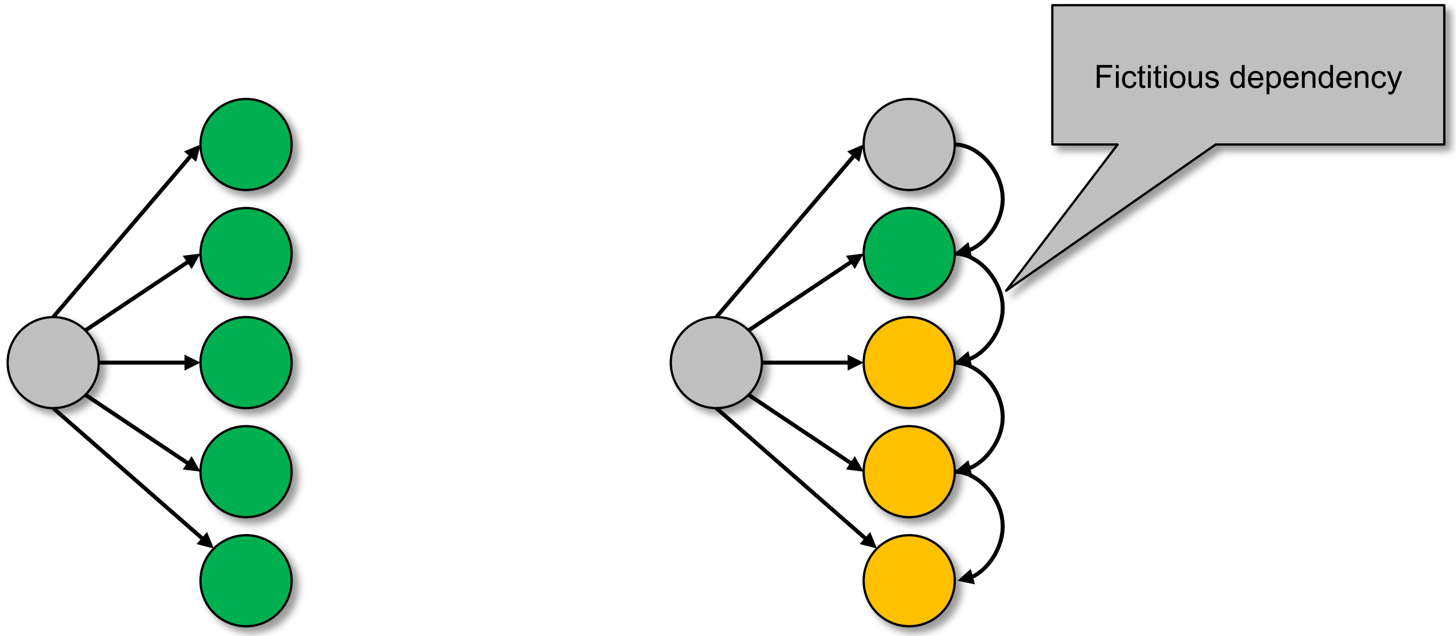
class invariant:
this stores the last element of the pipeline

```cpp
hpx::future<PromiseGuard<T>> operator()() {
  auto before_last = std::move(future_);

  hpx::lcos::local::promise<T> promise_next;
  future_ = promise_next.get_future();

  return before_last.then(
      hpx::launch::sync,
      hpx::unwrapping(
        [promise_next = std::move(promise_next)](T &&object) mutable {
          return PromiseGuard<T>{std::move(object), std::move(promise_next)};
      }));
}
```



Note

Pipeline is just a partial solution, since communications that can in theory run in parallel, are forced to run in a specific order, and this may cause performance degradation.



Parallelism

Exploiting new node architectures, which are capable of running multiple operations in parallel, requires a finer grained parallelism.

For what concerns linear algebra this is possible thanks to the “tiled” version of algorithms, i.e. divide the problem in smaller blocks.

These blocks can then be run in parallel, respecting the dependencies defined.

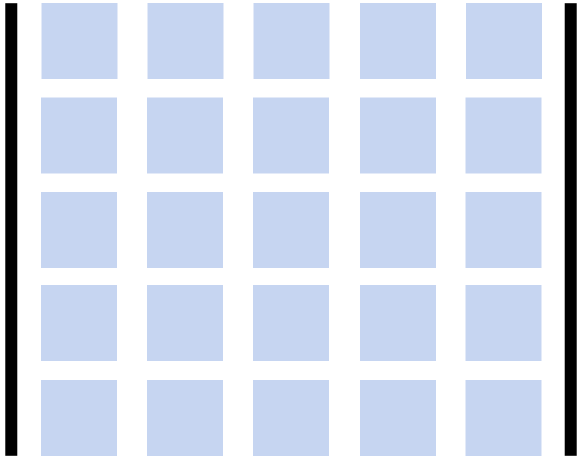
It sounds like a good use-case for HPX and task-based programming, and it is…

That’s what we are doing with DLAF

DATA-DRIVEN TASK BASED IMPLEMENTATION

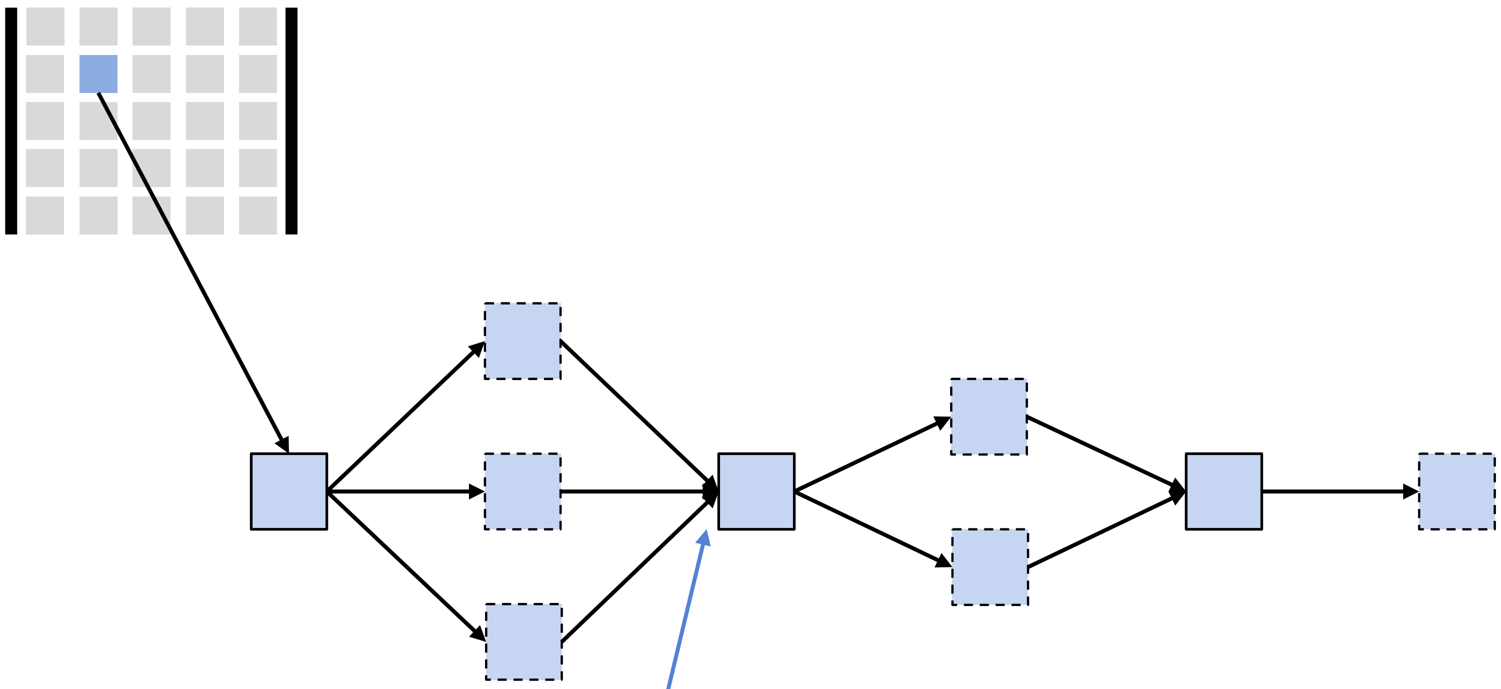
The Matrix

In a linear algebra library, the matrix plays a central role by definition, since it is the data structure on which algorithms runs.



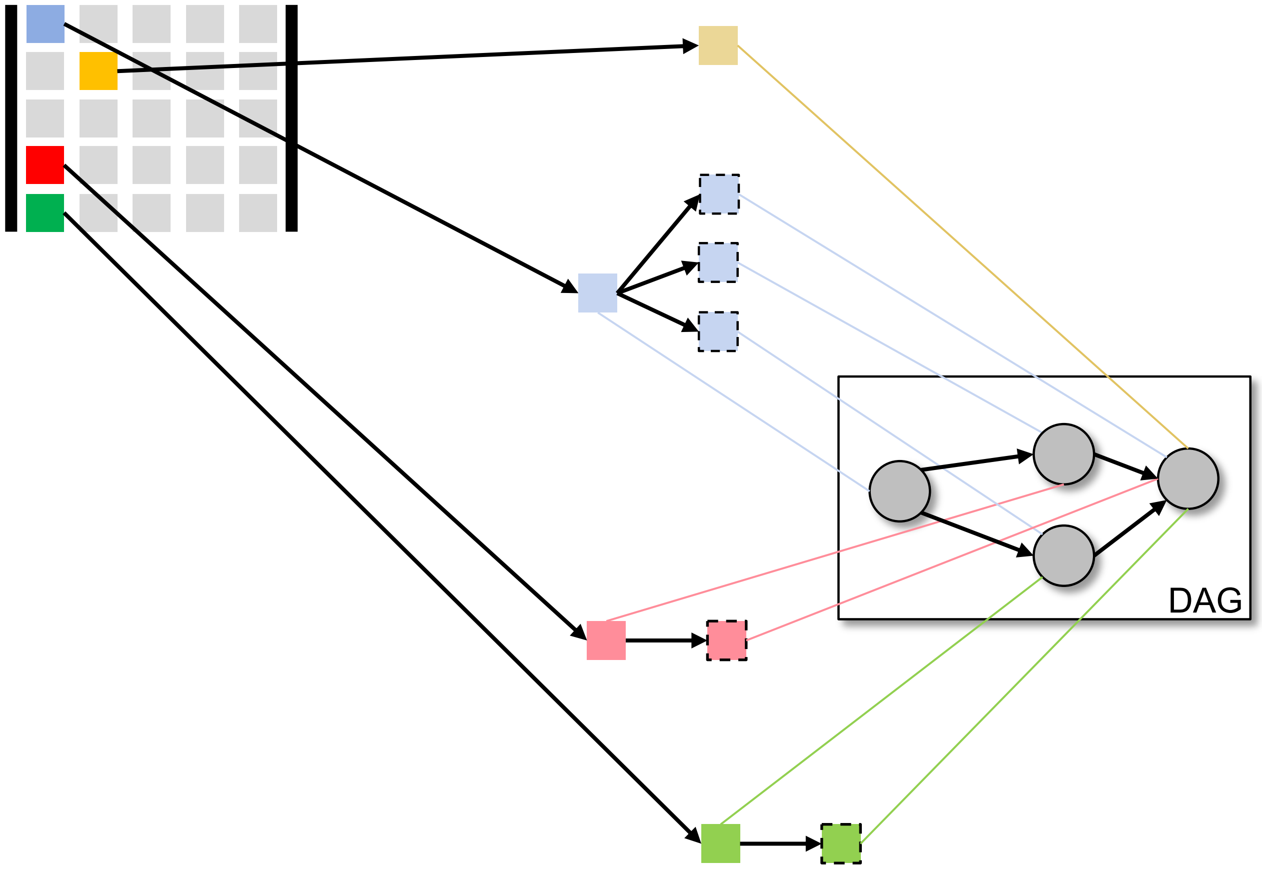
Algorithms are defined in terms of small tasks that run on tiles.

Tile as orchestrator of tasks



RW future value won't be set until the last "previous" RO one is released.

Tile as orchestrator of tasks



What's next?

The future of futures: sender-receiver

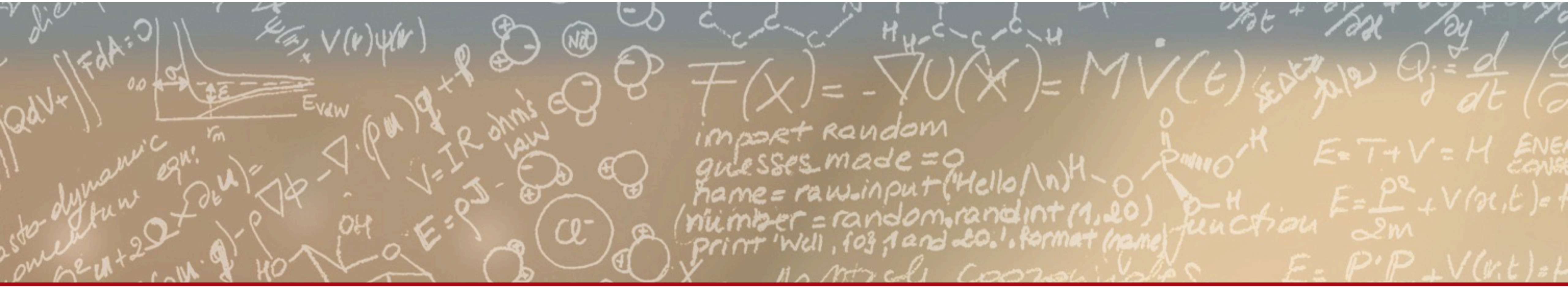
- std::future is largely considered a mistake; it requires heap allocation, does not support continuations, and is largely tied to std::thread
  - hpx::future inherits many of the downsides of std::future
- senders and receivers are a generalization of futures and promises based on concepts, proposed for standardization
- concept-based design allows different asynchronous libraries to interoperate; aims to be for asynchrony what iterators are for iteration
- allows for low-level customization based on the execution context, predecessor, and successor operations

Thanks to Mikael Simberg (github @msimberg)
HPX Maintainer

For more information and insights, you can:

✓ see the details in the proposal at wg21.link/p2300

✓ look at the presentation given by E.Niebler @ CppCon19
https://www.youtube.com/watch?v=tF-Nz4aRWAM&t=765s



Thank you for your attention.