# Math 4997-3

## Lecture 17: Distributed implementation of the 1D heat equation

Patrick Diehl

https://www.cct.lsu.edu/~pdiehl/teaching/2021/4997/

This work is licensed under a Creative Commons “Attribution-NonCommercial-NoDerivatives 4.0 International” license.

## Reminder

# Lecture 16

## What you should know from last lecture

- Serialization
- Distributed computing
  - Plain actions
  - Components
  - Components actions

## Serialization

## Adding serialization

Note even if all the code in this iteration runs for now only one one locality (node), we have to add serialization since all arguments passed to an action have to support serialization.

```
// We have to add this class as a friend
friend class hpx::serialization::access;

// Add a function to serialize all arguments
template <typename Archive>
void serialize(Archive& ar,
     const unsigned int version) const {}
```

Note that we will implement this function in the next lecture when we prepare the code to run on multiple localities.

## Extending the partition data with serialization

```cpp
class partition_data
{
private:
    // Note we have to make the unique_ptr to
    // a shared_ptr since we share the data
    std::shared_ptr<double[]> data_;
    std::size_t size_;

    friend class hpx::serialization::access;

    template <typename Archive>
    void serialize(Archive& ar,
        const unsigned int version) const {}
};
```

## Adding components and actions

## Adding a server to handle the partitions I

```
struct partition_server
  : hpx::components::component_base<partition_server>
{
// construct new instances
    partition_server() {}

    partition_server(partition_data const& data)
      : data_(data)
    {}

private:
    partition_data data_;
};
```

## Adding component action to the server II

```
struct partition_server
  : hpx::components::component_base<partition_server>
{
 // access data
    partition_data get_data() const
    {
        return data_;
    }

    // Every member function which has to be invoked
    // remotely needs to be wrapped into a
    // component action.
    HPX_DEFINE_COMPONENT_DIRECT_ACTION(
        partition_server,
        get_data, get_data_action);
};
```

## Register the component and component action

```
// Register the component

typedef hpx::components::component<partition_server>
    partition_server_type;
HPX_REGISTER_COMPONENT(partition_server_type,
    partition_server);

// Register the component action

typedef partition_server::get_data_action
    get_data_action;
HPX_REGISTER_ACTION(get_data_action);
```

## Writing the client helper class I

```
struct partition
      : hpx::components::client_base<partition,
        partition_server>
{

    typedef hpx::components::client_base<partition,
        partition_server> base_type;

    partition() {}

    // Create new component on locality 'where'
    // and initialize the held data
    partition(hpx::id_type where, std::size_t size,
        double initial_value)
      : base_type(hpx::new_
           <partition_server>(where, size,
                initial_value))
    {}
};
```

## Writing the client helper class II

```
struct partition
        : hpx::components::client_base<partition,
          partition_server>
{

// Create a new component on the locality co-located
// to the id 'where'. The new instance will be
// initialized from the given partition_data.
    partition(hpx::id_type where,
        partition_data && data)
      : base_type(hpx::new_<partition_server>
            (hpx::colocated(where), std::move(data)))
    {}
};
```

## Writing the client helper class III

```
struct partition
      : hpx::components::client_base<partition,
        partition_server>
{

// Attach a future representing a
// (possibly remote) partition.
    partition(hpx::future<hpx::id_type> && id)
      : base_type(std::move(id))
    {}

};
```

## Writing the client helper class IV

```
struct partition
        : hpx::components::client_base<partition,
          partition_server>
{

// Unwrap a future<partition> (a partition already
// holds a future to the id of the referenced object,
// thus unwrapping accesses this inner future).
partition(hpx::future<partition> && c)
      : base_type(std::move(c))
    {}

};
```

## Writing the client helper class V

```
struct partition
        : hpx::components::client_base<partition,
          partition_server>
{

// Invoke the (remote) member function which
// gives us access to the data.
hpx::future<partition_data> get_data() const
{
    return hpx::async(get_data_action(), get_id());
}

};
```

## Preparing the simulation control to be distributed

## Updating the computation of the heat

```
static partition heat_part(partition const& left,
   partition const& middle,partition const& right)
{
return dataflow(
  unwrapping(
      [middle](partition_data const& l,
          partition_data const& m,
          partition_data const& r)
          {
          // The new partition_data will be allocated
          //  on the same locality as 'middle'.
          return partition(middle.get_id(),
             heat_part_data(l, m, r));
          }
          ),
          left.get_data(), middle.get_data(),
             right.get_data());
}
```

## Updating the computation of the heat II

```
stepper::space stepper::do_work(std::size_t np,
   std::size_t nx, std::size_t nt)
{
    // Initial conditions: f(0, i) = i
    for (std::size_t i = 0; i != np; ++i)
        U[0][i] = partition(hpx::find_here(),
           nx, double(i));

}
```

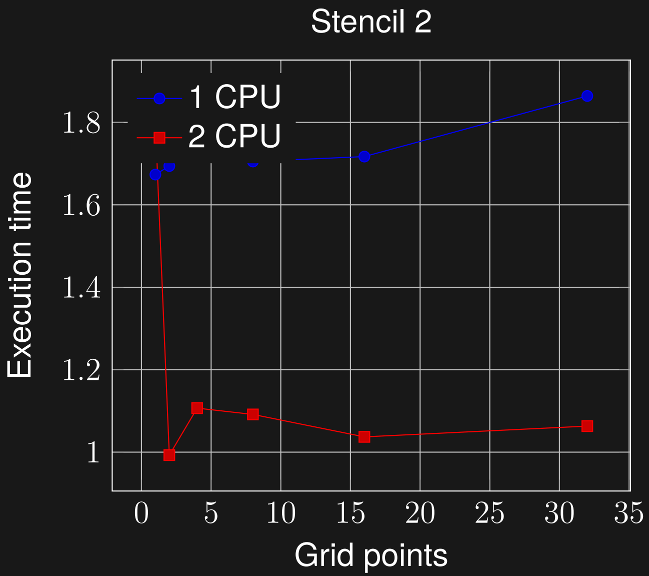
Note that we only use `hpx::find_here()` the code will run one one logicality only.

## Updating the computation of the heat III

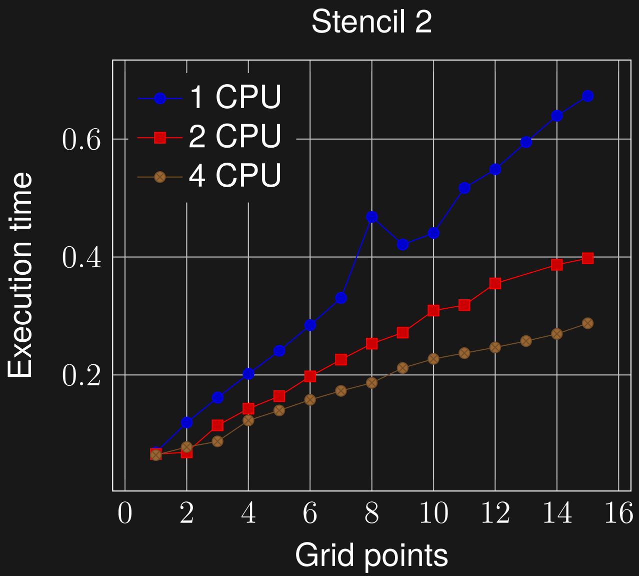
```
// Generate place holder for the changing arguments
using hpx::util::placeholders::_1;
using hpx::util::placeholders::_2;
using hpx::util::placeholders::_3;

// Pass the fixed argument to the function
auto Op = hpx::util::bind(heat_part_action(),
    hpx::find_here(), _1, _2, _3);

next[i] = dataflow(
    hpx::launch::async, Op,
    current[idx(i, -1, np)],
    current[i],
    current[idx(i, +1, np)]
);
```

## Scaling results

## Scaling for 10000000 discrete mesh points



# Scaling for increasing amount of work



## Summary

## Summary

### After this lecture, you should know

- How to use components and actions to make remote function calls