# Math 4997-3

## Lecture 14: Serial partition-based 1D heat equation

Patrick Diehl 

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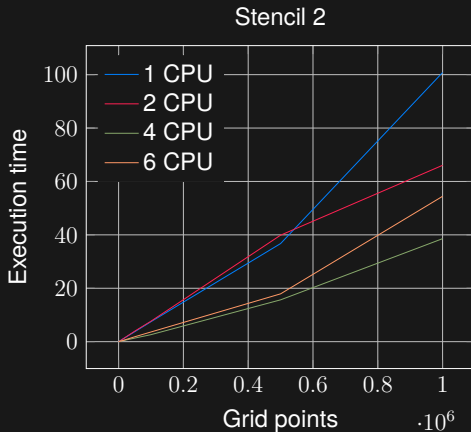
Reminder

# Lecture 13

## What you should know from last lecture

- ► `hpx::make_reday_future`
- ► `hpx::dataflow`
- ► `hpx::unwrapping`
- ► `hpx::shared_future`

Recap of the previous implementation

# Scaling results from previous example



Stencil 2

Note that we need to control the grain size (the amount of work) to get better scalability.

# Introducing partitions

# Goal of this week's lectures

We will reuse the serial implementation of Lecture 12 and

- ▶ introduces a partitioning of the 1D grid into groups of grid partitions
- ▶ which are handled at the same time.

So we can control the the amount of work performed (grain size) to improve the scalability of our program.

Note that before, we had exact one discrete mesh point handle per `hpx::future` and now we want to have multiple discrete mesh points.

# Generating partitions
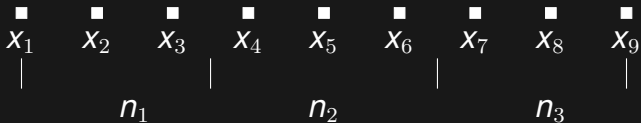


## Struct holding the data for each partition

```
struct partition_data
{
private:
    std::vector<double> data_;
};
```

# Data structure for the partitions I

```cpp
struct partition_data
{
  partition_data(std::size_t size = 0)
      : data_(size)
    {}

  partition_data(std::size_t size, double int_value)
      : data_(size)
    {
        double base_value =
            double(int_value * size);
        for (std::size_t i = 0; i != size; ++i)
            data_[i] = base_value + double(i);
    }
}
```

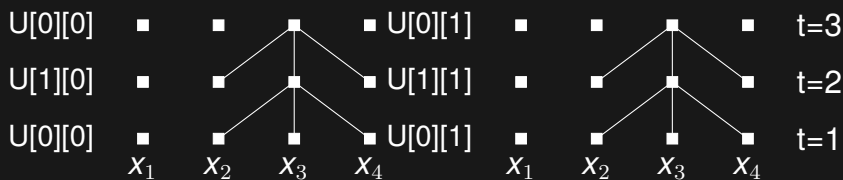# Data structure for the partitions II

```cpp
struct partition_data
{
  double& operator[](std::size_t idx) {
     return data_[idx];
  }

  double operator[](std::size_t idx) const {
      return data_[idx];
    }

    std::size_t size() const
    {
      return data_.size();
    }
}
```

Swapping the partitions

# New swapping scheme

# Adapted class stepper

```
class stepper
{
    // Our data for one time step
    typedef partition_data partition;
    typedef std::vector<partition> space;

    std::vector<space> U(2);
    for (space& s: U)
        // np is the number of partitions
        s.resize(np);

    // Initial conditions: f(0, i) = i
    for (std::size_t i = 0; i != np; ++i)
        U[0][i] = partition_data(nx, double(i));

    // Return the solution at time-step 'nt'.
    return U[nt % 2];

}
```

# New C++ and HPX features

# Moving objects with `std::move`[1]

We use `std::move` to indicate that the object may be moved to another object. This allows the efficient transfer of resources to another object.

## Example

```
std::string str = "Hello";
std::vector<std::string> v;

v.push_back(std::move(str));
```

## Be aware of undefined states, but valid states

```
str.clear{} //Ok, since clear has no preconditions
str.back() //Undefined behavior if size()==0
```

[1] https://en.cppreference.com/w/cpp/utility/move

# Semaphore

## Analogy

Imagine a public library lending books with no late fee. The have 5 copies of the Hitchhiker's Guide to the Galaxy [1] to borrow. Five people can borrow these copies and the library does not care to get them back in a feasible amount of time, since they avoided to introduce late fees. If one is waiting for a copy the copy will be assigned to this person, but if none is waiting the copy just goes back to the shelf until one asks for it.

The concept of semaphores was introduced by E. Dijkstra [2]. More details [3].

# P and V operations on a semaphore objects

## Variables

- maximum count
- current count

## Operations

- Taking ownership with the `wait` function, which decrements the semaphore. The so–called *P* operation from Dijkstra's paper.
- Releasing ownership with the `signal` function, the increments the semaphore. The so–called *V* operation from Dijkstra's paper.

# More details

### *P*-Operation

If the `wait` function is called, the current count is decreased. If the count is $\geq$ zero then the decrement just happens and the function will return. If the count is zero the function will wait until one other thread called the `signal` function.

### *V*-Operation

If the `signal` function is called, the current count is increased. If the count was zero before you called `signal` function and another thread was blocked in `wait` then that thread will be executed. If multiple threads are waiting, only one will be executed and the reaming ones have to wait for another increment of the counter.

# Semaphores in the C++ standard

We looked into `std::mutex` in Lecture 6, which is tied to one thread and only one thread can lock or unlock the mutex. Any thread can access the ownership on a semaphore.

The C++ standard does not define semaphores.

```cpp
// Generate a semaphore with maximal count nd
hpx::lcos::local::sliding_semaphore sem(nd);

// Release ownership for t
sem.signal(t);

// Obtain ownership for t
sem.wait(t);
```

# Summary

# Summary

After this lecture, you should know

▶ Using the partitions to control the grain size

▶ `std::move` for moving objects

▶ Semaphores and `hpx::lcos::local::sliding_semaphore`

# References

# References I

[1] Douglas Adams.
*The Hitchhiker's Guide to the Galaxy Omnibus: A Trilogy in Five Parts*, volume 6.
Pan Macmillan, 2017.

[2] Edsger W Dijkstra.
Over de sequentialiteit van procesbeschrijvingen.
*Trans. by Martien van der Burgt and Heather Lawrence. In*, 1962.

[3] Allen Downey.
*The little book of semaphores*.
Green Tea Press, 2008.