Using HPX with C++

Thomas Heller (thomas.heller@cs.fau.de), Bryce Lelbach (balelbach@lbl.gov), Alice Koniges (aekoniges@lbl.gov)

November 15, 2015

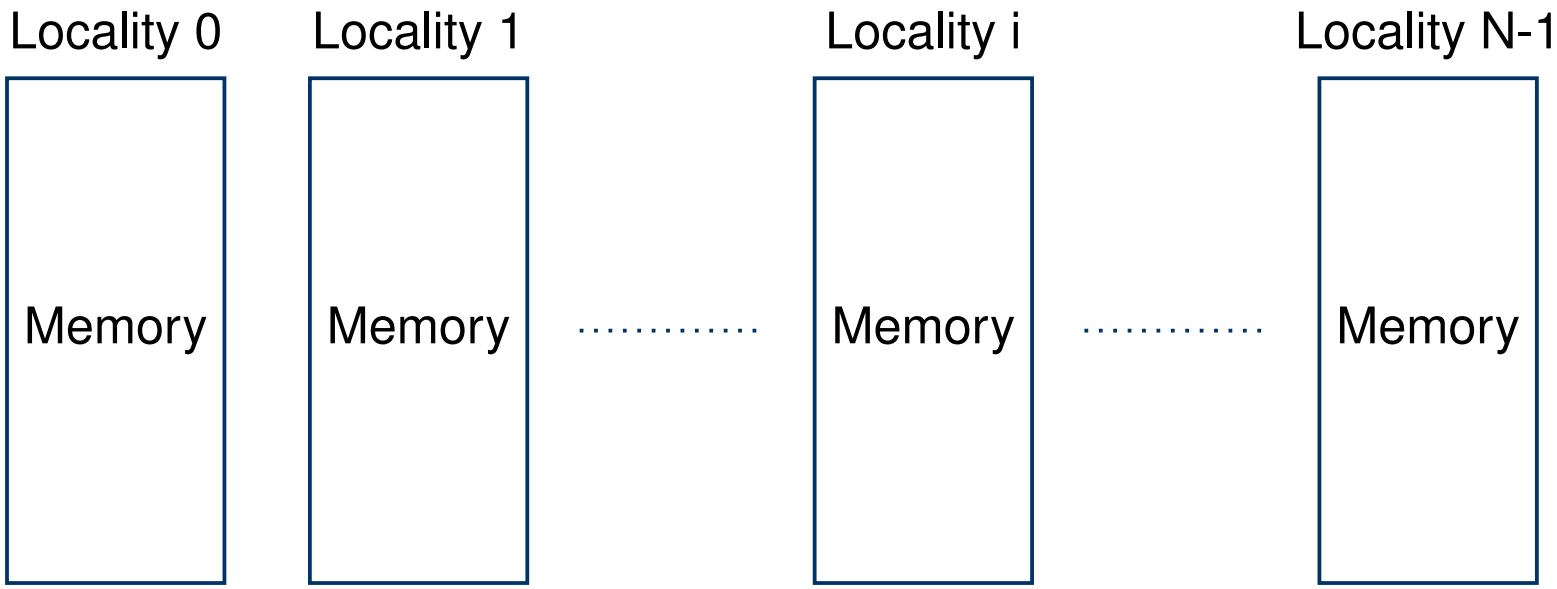
This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 671603

STE||AR GROUP

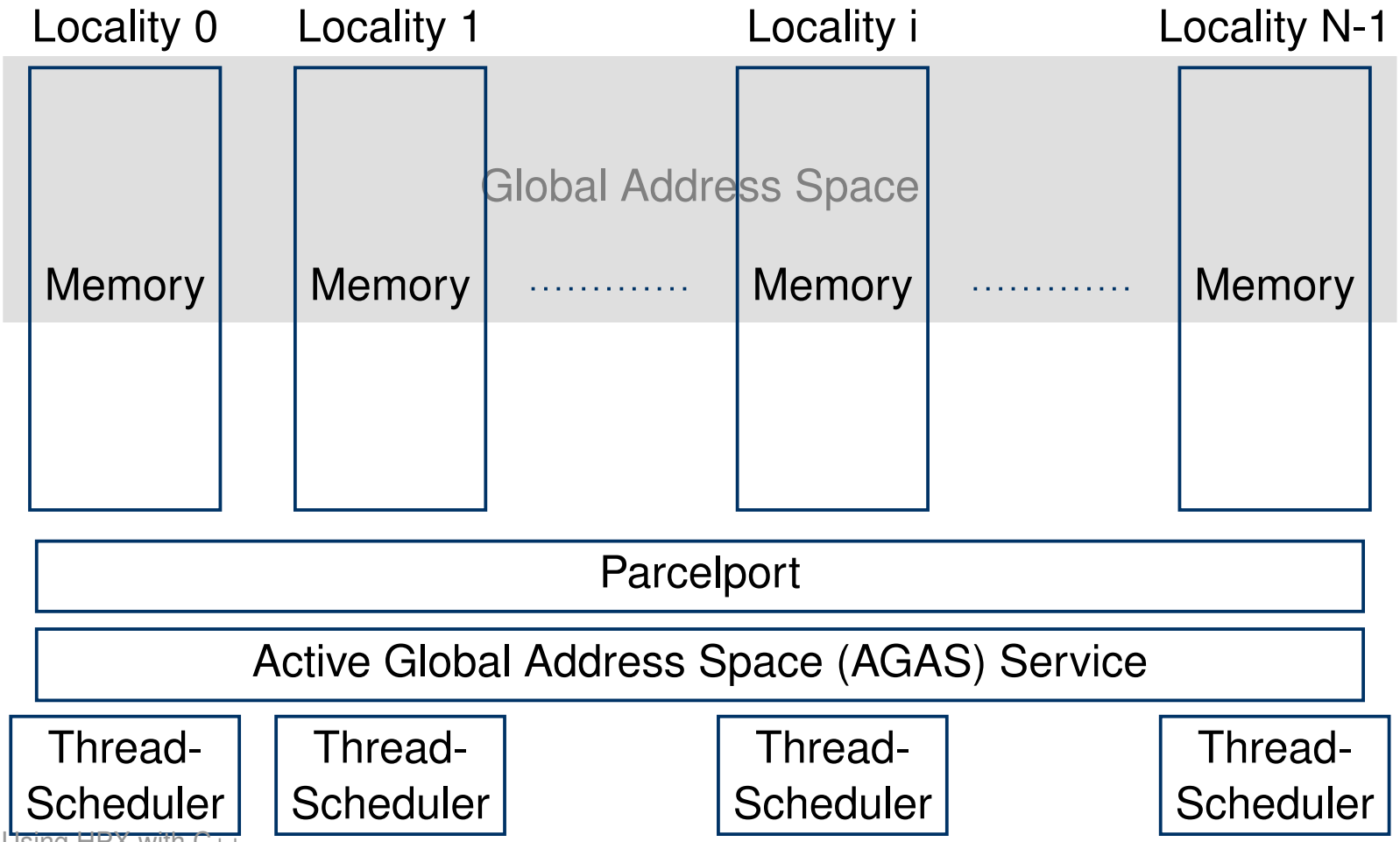
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

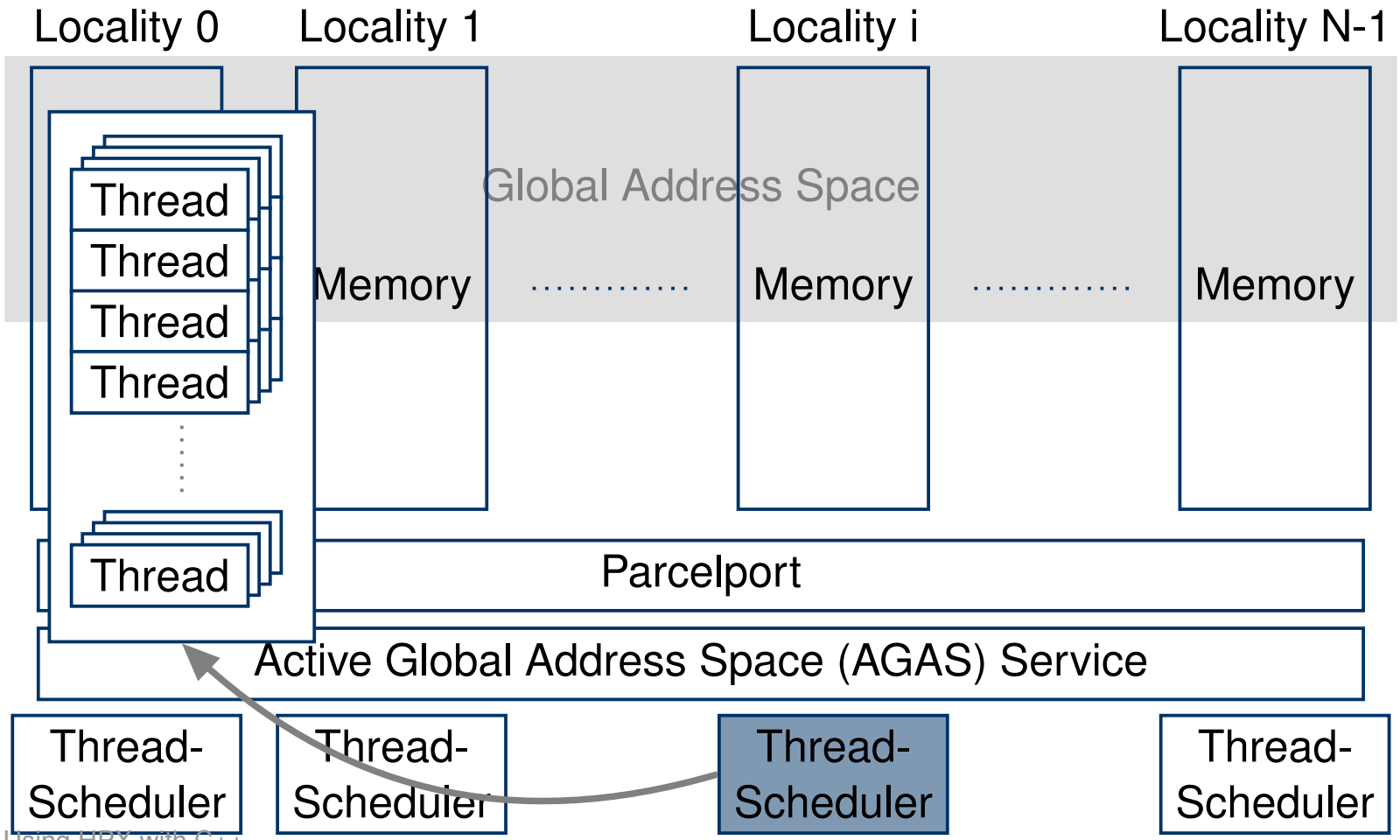
HPX 101 – The programming model



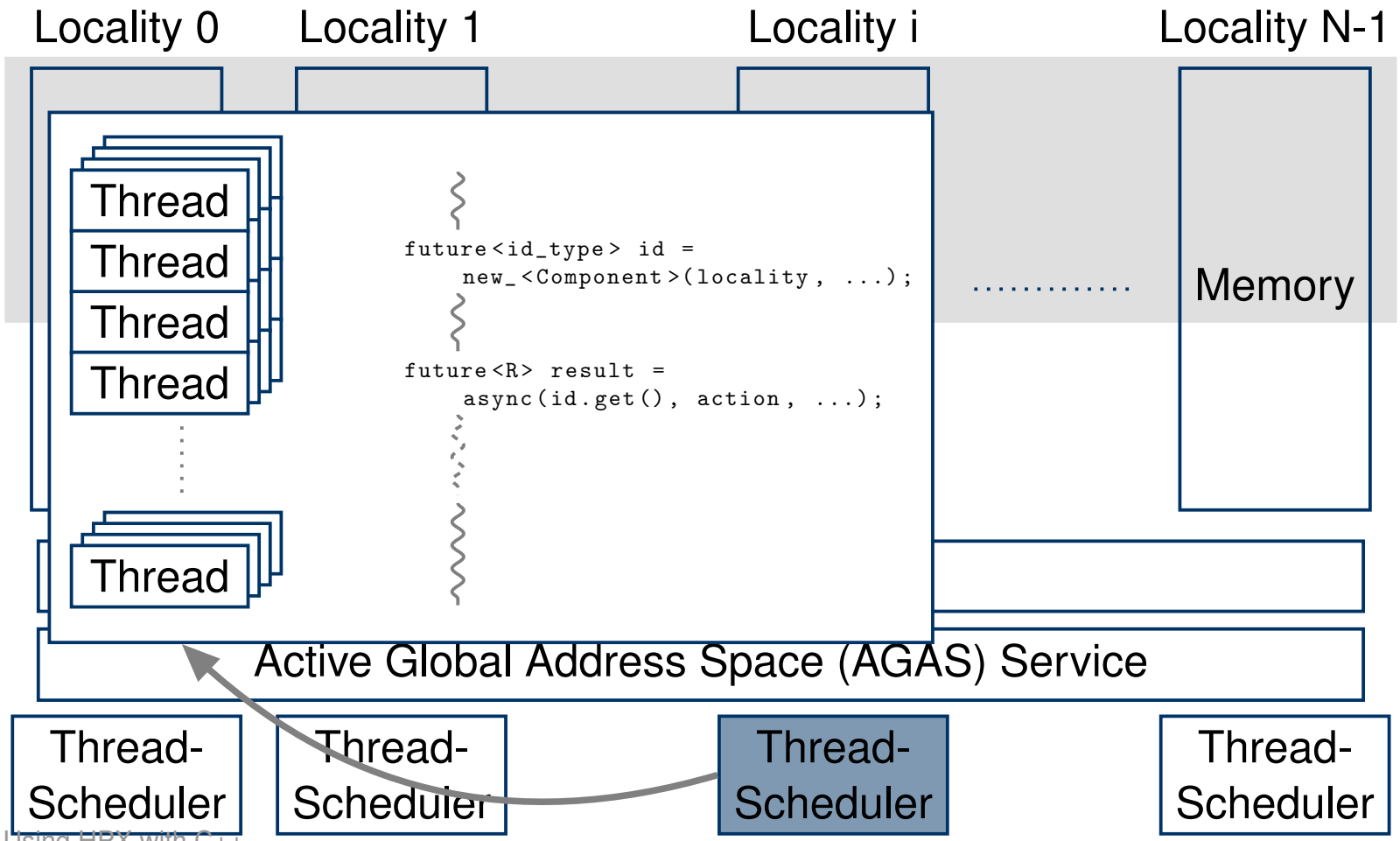
HPX 101 – The programming model



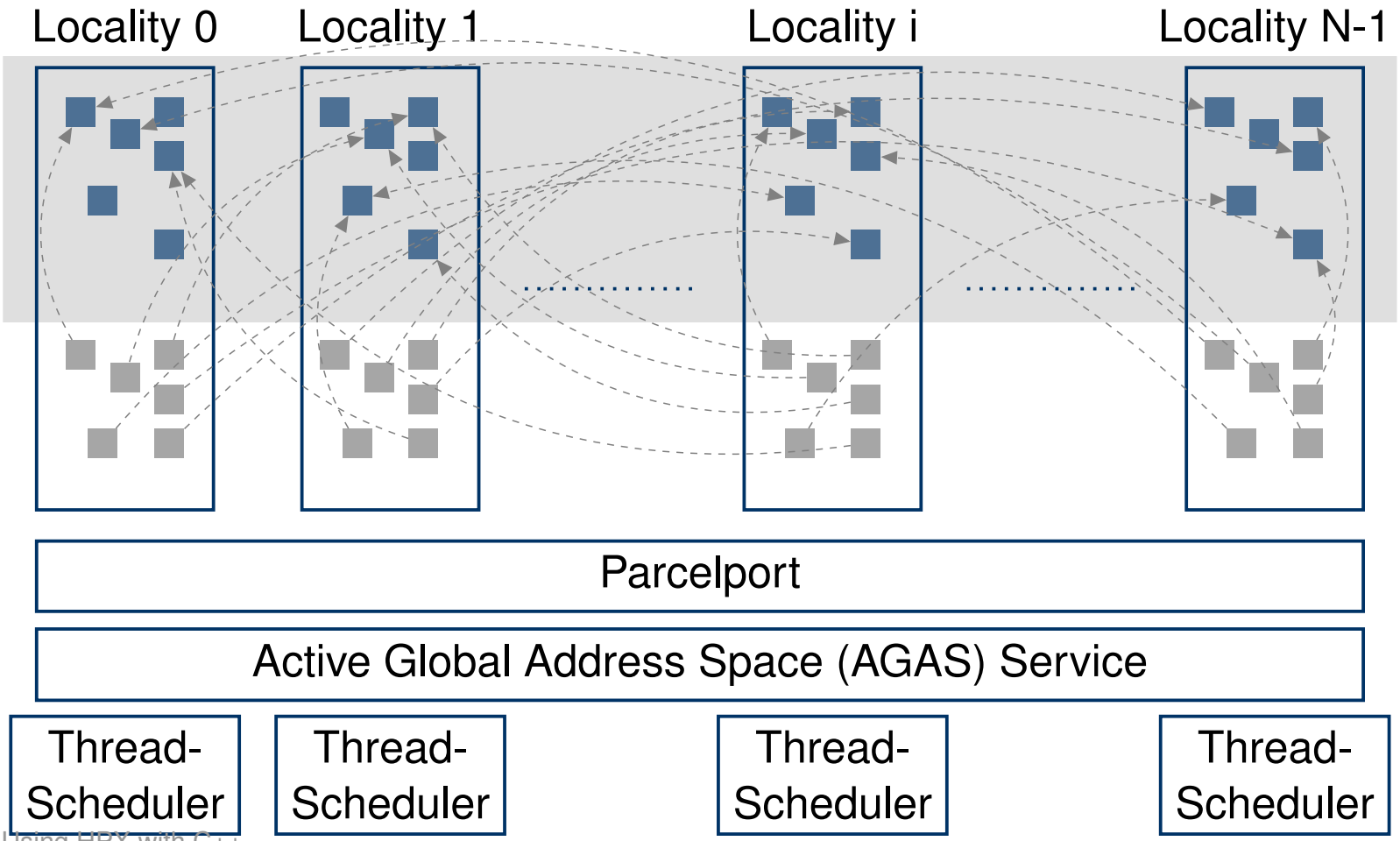
HPX 101 – The programming model



HPX 101 – The programming model



HPX 101 – The programming model



HPX 101 – Overview

`R f(p...)`	Synchronous (returns `R`)	Asynchronous (returns `future<R>`)	Fire & Forget (returns `void`)
Functions (direct)	`f(p...)`	`async(f, p...)`	`apply(f, p...)`
Functions (lazy)	`bind(f, p...)(...)`	`async(bind(f, p...), ...)`	`apply(bind(f, p...), ...)`
Actions (direct)	`HPX_ACTION(f, a)` `a()(id, p...)`	`HPX_ACTION(f, a)` `async(a(), id, p...)`	`HPX_ACTION(f, a)` `apply(a(), id, p...)`
Actions (lazy)	`HPX_ACTION(f, a)` `bind(a(), id, p...)(...)`	`HPX_ACTION(f, a)` `async(bind(a(), id, p...), ...)`	`HPX_ACTION(f, a)` `apply(bind(a(), id, p...), ...)`

C++

C++ Standard Library

HPX

In Addition: `dataflow(func, f1, f2);`

HPX 101 – Example

```
void hello_world(std::string msg)
{ std::cout << msg << '\n'; }

// Register hello_world as an action
HPX_PLAIN_ACTION(hello_world);

// Asynchronously call hello_world: Returns a future
hpx::future<void> f1
  = hpx::async(hello_world, "Hello HPX!");

// Asynchronously call hello_world: Fire & forget
hpx::apply(hello_world, "Forget me not!");

// Asynchronously call hello_world_action
hpx::future<void> f2
  = hpx::async(hello_world_action, hpx::find_here(), "Hello HPX!");

// Asynchronously call hello_world after f1 and f2 are ready
hpx::future<void> f3
  = dataflow(hello_world, f1, f2, "The world is in flow!");
```

HPX 101 – Future Composition

```
// Attach a Continuation to a future
future<R> ff = ...;
ff.then([](future<R> f){ do_work(f.get()) });

// Returns a future which becomes ready when all input become ready
hpx::when_all(...);

// Returns a future which becomes ready when n of the input futures become ready
hpx::when_some(...);

// Returns a future which becomes ready when one of the input futures become ready
hpx::when_any(...);
```

Fibonacci – serial

```
int fib(int n)
{
  if (n < 2) return n;
  return fib(n-1) + fib(n-2);
}
```

Fibonacci – parallel

```
int fib(int n)
{
  if (n < 2) return n;

  future<int> fib1 = hpx::async(fib, n-1);
  future<int> fib2 = hpx::async(fib, n-2);
  return fib1.get() + fib2.get();
}
```

Fibonacci – parallel, take 2

```
future<int> fib(int n)
{
  if(n < 2)
    return hpx::make_ready_future(n);

  if(n < 10)
    return hpx::make_ready_future(fib_serial(n));

  future<int> fib1 = hpx::async(fib, n-1);
  future<int> fib2 = hpx::async(fib, n-2);
  return
    dataflow(unwrapped([](int f1, int f2){
        return f1 + f2;
    }), fib1, fib2);
}
```

Fibonacci – parallel, take 3

```
future<int> fib(int n)
{
  if(n < 2)
    return hpx::make_ready_future(n);

  if(n < 10)
    return hpx::make_ready_future(fib_serial(n));

  future<int> fib1 = hpx::async(fib, n-1);
  future<int> fib2 = hpx::async(fib, n-2);
  return await fib1 + await fib2;
}
```

Loop parallelization

```
// Serial version

int lo = 1;
int hi = 1000;
auto range
  = boost::irange(lo, hi);
for(int i : range)
{
  do_work(i);
}
```

```
// Parallel version

int lo = 1;
int hi = 1000;
auto range
  = boost::irange(lo, hi);
using hpx::parallel::
   for_each;
using hpx::parallel::par;
for_each(
  par, begin(range), end(
     range),
  [](int i)
  {
    do_work(i);
  }
);
```

```
// Parallel version

int lo = 1;
int hi = 1000;
auto range
  = boost::irange(lo, hi);
using hpx::parallel::for_each;
using hpx::parallel::task;
future<void> f = for_each(
  task, begin(range), end(range),
  [](int i)
  {
    do_work(i);
  }
);
other_expensive_work();
// Wait for parallel loop to finish:
f.wait();
```

Parallel algorithms

- adjacent_difference
  inner_product
- adjacent_find
- all_of any_of none_of
- copy copy_if copy_n
- move
- count count_if
- equal mismatch
- exclusive_scan inclusive_scan
- reduce
- transform
- fill fill_n
- find find_end find_first_of
  find_if find_if_not
- for_each for_each_n
- generate generate_n
- is_heap is_heap_until
- is_partitioned is_sorted
  is_sorted_until
- lexicographical_compare
- max_element max_element
  minmax_element
- partial_sort partial_sort_copy
  nth_element sort stable_sort
- partition partition_copy
  stable_partition
- remove remove_if
- remove_copy remove_copy_if
- reverse reverse_copy
- replace
- rotate rotate_copy
- search search_n
- set_difference
  set_intersection
  set_symmetric_difference
  set_union includes
- inplace_merge merge
- swap_ranges
- uninitialized_copy
  uninitialized_copy_n
- uninitialized_fill
  uninitialized_fill_n
- unique unique_copy
- transform_reduce
- transform_exclusive_scan
  transform_inclusive_scan

Components Interface: Writing a component

```
struct hello_world_component;
struct hello_world;

int main()
{
  hello_world hw(hpx::find_here());

  hw.print();
}
```

Components Interface: Writing a component

```
// Component implementation
struct hello_world_component
  : hpx::components::component_base<
        hello_world_component
    >
{
    // ...
};
```

Components Interface: Writing a component

```
// Component implementation
struct hello_world_component
  : hpx::components::component_base<
        hello_world_component
    >
{
    void print() { std::cout << "Hello World!\n"; }
    // define print_action
    HPX_DEFINE_COMPONENT_ACTION(hello_world_component, print);
};
```

Components Interface: Writing a component

```
// Component implementation
struct hello_world_component
  : hpx::components::component_base<
        hello_world_component
    >
{
    // ...
};

// Register component
typedef hpx::components::component<
    hello_world_component
> hello_world_type;

HPX_REGISTER_MINIMAL_COMPONENT_FACTORY(hello_world_type, hello_world);
```

Components Interface: Writing a component

```
// Component implementation
struct hello_world_component
  : hpx::components::component_base<
        hello_world_component
    >
{
    // ...
};

// Register component ...

// Register action
HPX_REGISTER_ACTION(print_action);
```

Components Interface: Writing a component

```
struct hello_world_component;

// Client implementation
struct hello_world
  : hpx::components::client_base<hello_world, hello_world_component>
{
    // ...
};

int main()
{
    // ...
}
```

Components Interface: Writing a component

```
struct hello_world_component;

// Client implementation
struct hello_world
  : hpx::components::client_base<hello_world, hello_world_component>
{
    typedef
        hpx::components::client_base<hello_world, hello_world_component>
        base_type;

    hello_world(hpx::id_type where)
      : base_type(
          hpx::new_<hello_world_component>(where)
        )
    {}
};

int main()
{
    // ...
}
```

Components Interface: Writing a component

```
struct hello_world_component;

// Client implementation
struct hello_world
  : hpx::components::client_base<hello_world, hello_world_component>
{
    // base_type

    hello_world(hpx::id_type where);

    hpx::future<void> print()
    {
        hello_world_component::print_action act;
        return hpx::async(act, get_gid());
    }
};

int main()
{
    // ...
}
```

Components Interface: Writing a component

```
struct hello_world_component;

// Client implementation
struct hello_world
  : hpx::components::client_base<hello_world, hello_world_component>
{
    hello_world(hpx::id_type where);
    hpx::future<void> print();
};

int main()
{
    hello_world hw(hpx::find_here());
    hw.print();
}
```

Excercises

1. Quicksort – Insert futures to parallelize a quicksort function
2. Matmul – Insert futures to parallelize a matrix multiply function
3. 1D stencil – Insert futures to parallelize the main loop for a heat equation evolution
4. Integ – Parallelize the numerical integrator

Get the examples at:
`https://github.com/STEllAR-GROUP/SC15_Examples`

Example 1: 1D Heat equation

```
typedef hpx::future<double> partition;
std::vector<partition> grids[2];
std::size_t old = 0;
std::size_t cur = 1;
for(std::size_t t = 0; t < nt; ++t)
{
  for(std::size_t x = 1; x < nx-1; ++x)
    grids[cur] = hpx::lcos::dataflow(
        heat_diffusion
      , grids[old][x-1], grids[old][x], grids[old][x+1]
    );
  std::swap(old, cur);
}
wait_all(grids[old]);
```

Example 1: 1D Heat equation

```
struct partition_component // component (details omitted for clarity)
{
  typedef std::vector<double> partition_data;

  partition_data get_data();

  partition_data data_;
};
```

Example 1: 1D Heat equation

```
std::vector<partition> grids[2];
hpx::id_type left_neighbor, right_neighbor;
std::size_t old = 0;
std::size_t cur = 1;
for (std::size_t t = 0; t != nt; ++t)
{
  for(std::size_t x = 1; x < num_parts-1; ++x)
    grids[cur] = hpx::lcos::dataflow(
        heat_part
      , grids[old][x-1], grids[old][x], grids[old][x+1]
    );
}
```

Example 1: 1D Heat equation

```
partition heat_part(partition left, partition middle, partition right)
{
  hpx::future<partition_data> middle_part = middle.get_part();
  // ...
}
```

Example 1: 1D Heat equation

```
partition heat_part(partition left, partition middle, partition right)
{
  hpx::future<partition_data> middle_part;
  hpx::future<partition> next_middle = middle_part.then(
    hpx::util::unwrapped([](partition_data old) {
      partition_data next(old.size());

      for(std::size_t x = 1; x < old.size()-1; ++x)
        grids[cur] = hpx::lcos::dataflow(
          heat_diffusion
        , old[x-1], old[x], old[x+1]
        );
    })
  );
}
```

Example 1: 1D Heat equation

```
partition heat_part(partition left, partition middle, partition right)
{
  hpx::future<partition_data> middle_part;
  hpx::future<partition> next_middle;
  return dataflow(
    unwrapped([left, middle, right](partition_data next, partition_data const& l,
      partition_data const& m, partition_data const& r) -> partition {
      std::size_t size = m.size();
      next[0] = heat(l[size-1], m[0], m[1]);
      next[size-1] = heat(m[size-2], m[size-1], r[0]);
      return partition(middle.get_gid(), next);
    }),
    std::move(next_middle),
    left.get_part(),
    middle_data, right.get_part());
}
```

Example 1: 1D Heat equation

```
std::vector<hpx::future<partition>> grids[2];
hpx::id_type left_neighbor, right_neighbor;
std::size_t old = 0;
std::size_t cur = 1;
for (std::size_t t = 0; t != nt; ++t)
{
  // receive ...
  if(id != 0)
    grids[cur][0] = receive_left(t);
  if(id != ranks-1)
    grids[cur][num_parts-1] = receive_right(t);

  for(std::size_t x = 1; x < num_parts-1; ++x)
      grids[cur][x] = hpx::lcos::dataflow(
            heat_part
          , grids[old][x-1], grids[old][x], grids[old][x+1]
        );
  // send ...
  if(id != 0)
    send_left(grids[1])
  if(id != ranks-1)
    send_right(grids[num_parts-2]);
}
```

Example 1: 1D Heat equation

```
hpx::lcos::local::receive_buffer<partition> left_receiver;
hpx::future<partition> receive_left(std::size_t t)
{
  return left_receiver.receive(t);
}

hpx::lcos::local::receive_buffer<partition> right_receiver;
hpx::future<partition> receive_left(std::size_t t)
{
  return right_receiver.receive(t);
}
```

Example 1: 1D Heat equation

```
void send_left(partition p, std::size_t t)
{
  store_right_action act;
  hpx::apply(act, left_neighbor, t, p);
}
void store_right(std::size_t t, partition p)
{
  right_receiver.store_received(t, p);
}

void send_right(partition p, std::size_t t)
{
  store_left_action act;
  hpx::apply(act, right_neighbor, t, p);
}
void store_left(std::size_t t, partition p)
{
  left_receiver.store_received(t, p);
}
```