HPX by example

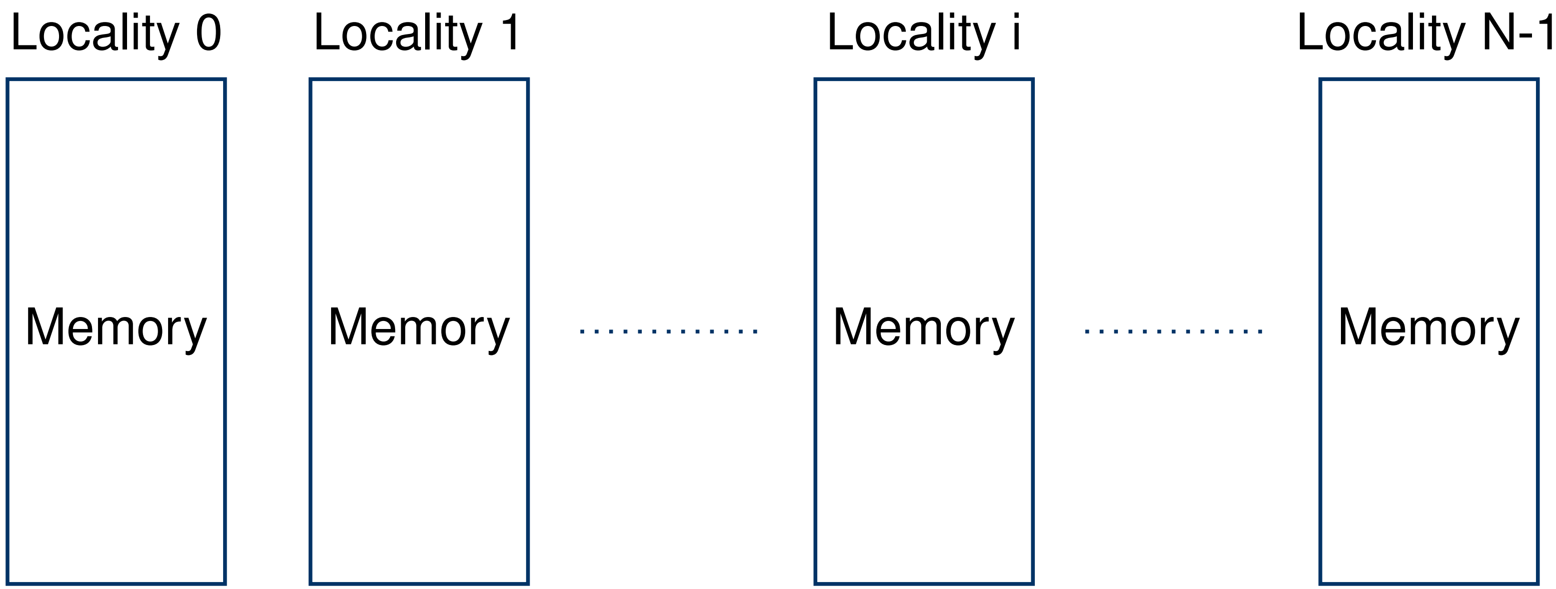
Thomas Heller (thomas.heller@cs.fau.de)
25.10.2014
FAU

LOUISIANA STATE UNIVERSITY
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

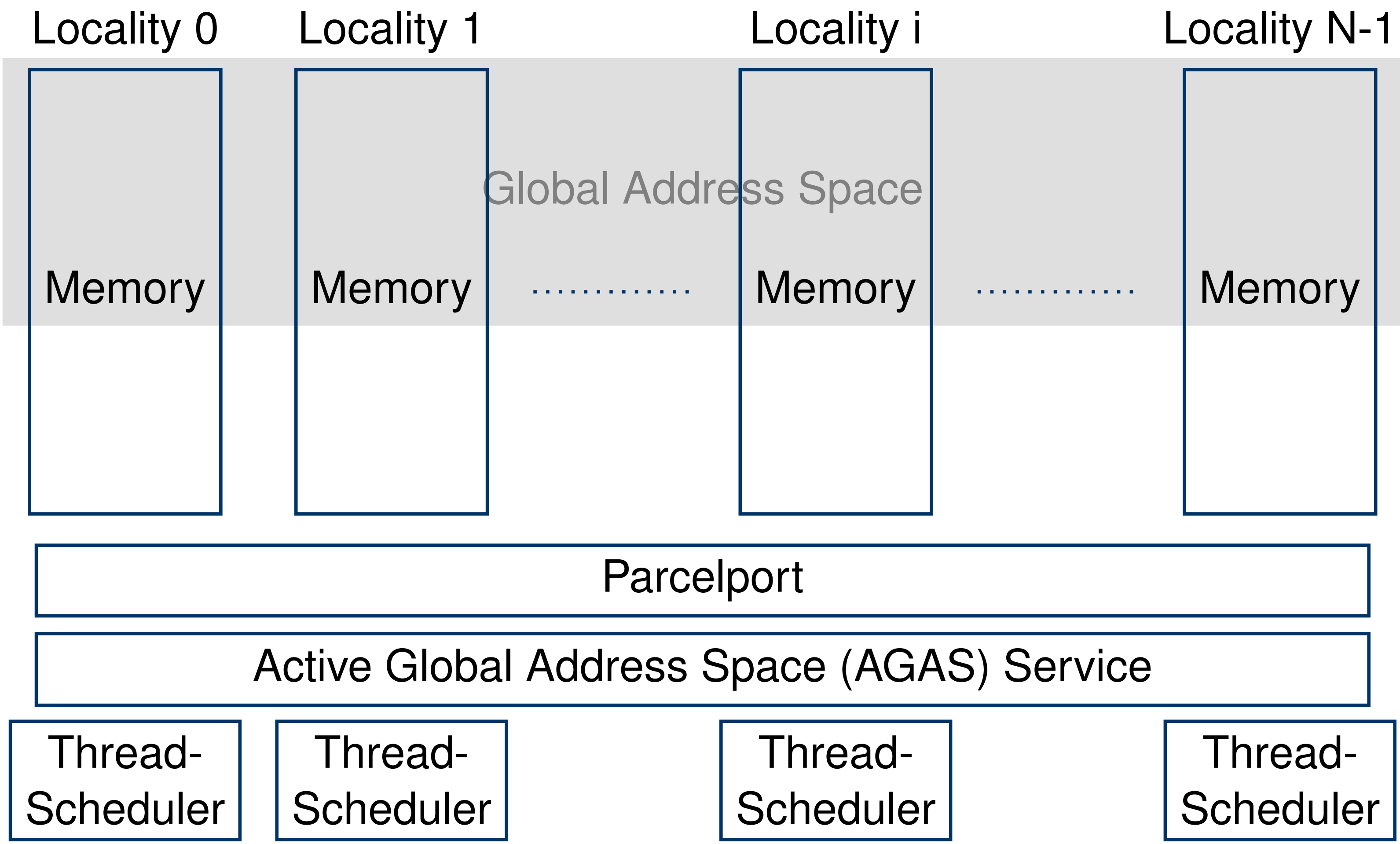
HPX – A general purpose parallel runtime system

HPX is a parallel runtime system which extends the C++11/14 standard to facilitate distributed operations, enable fine-grained constraint based parallelism, and support runtime adaptive resource management.

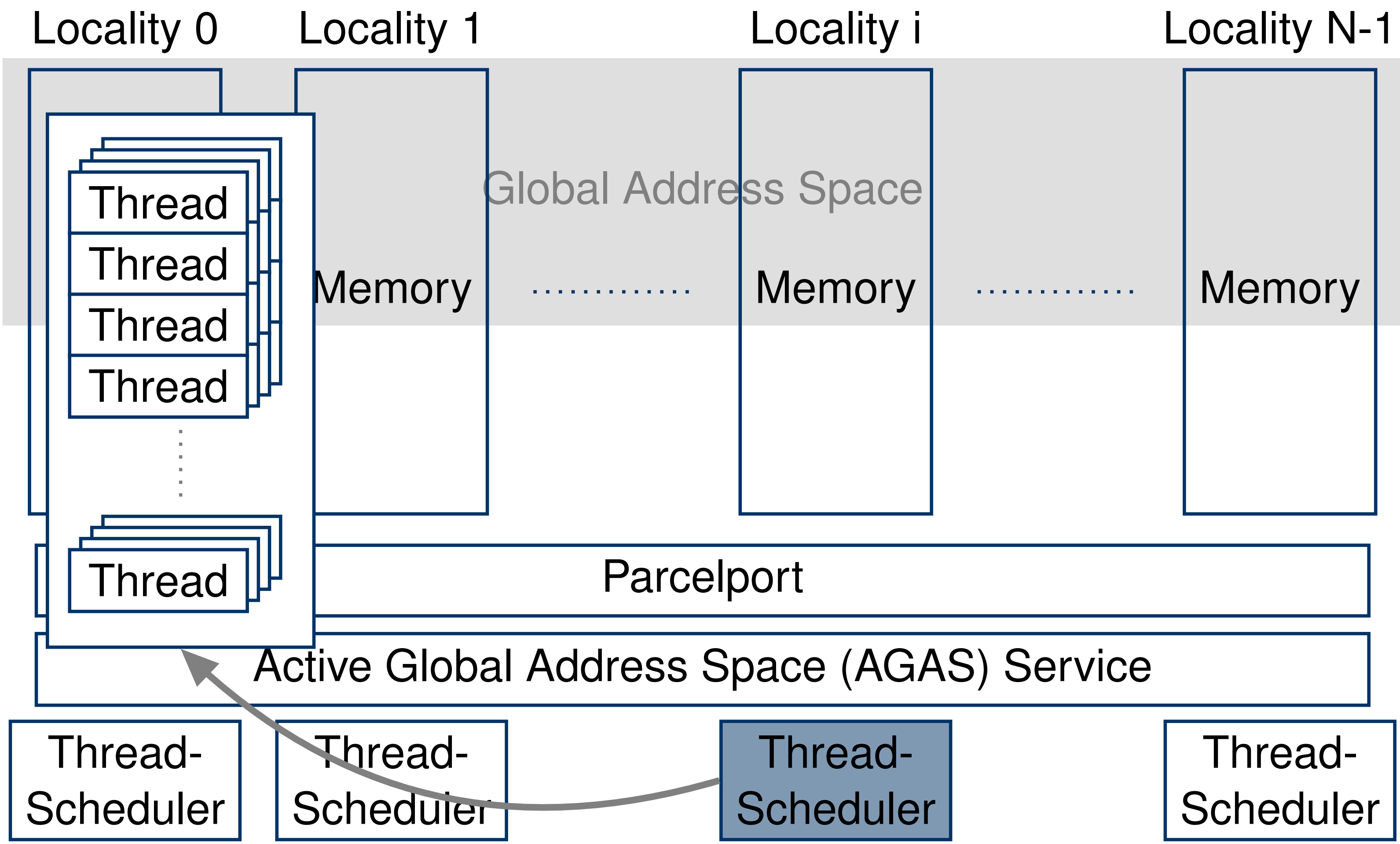
HPX – A general purpose parallel runtime system



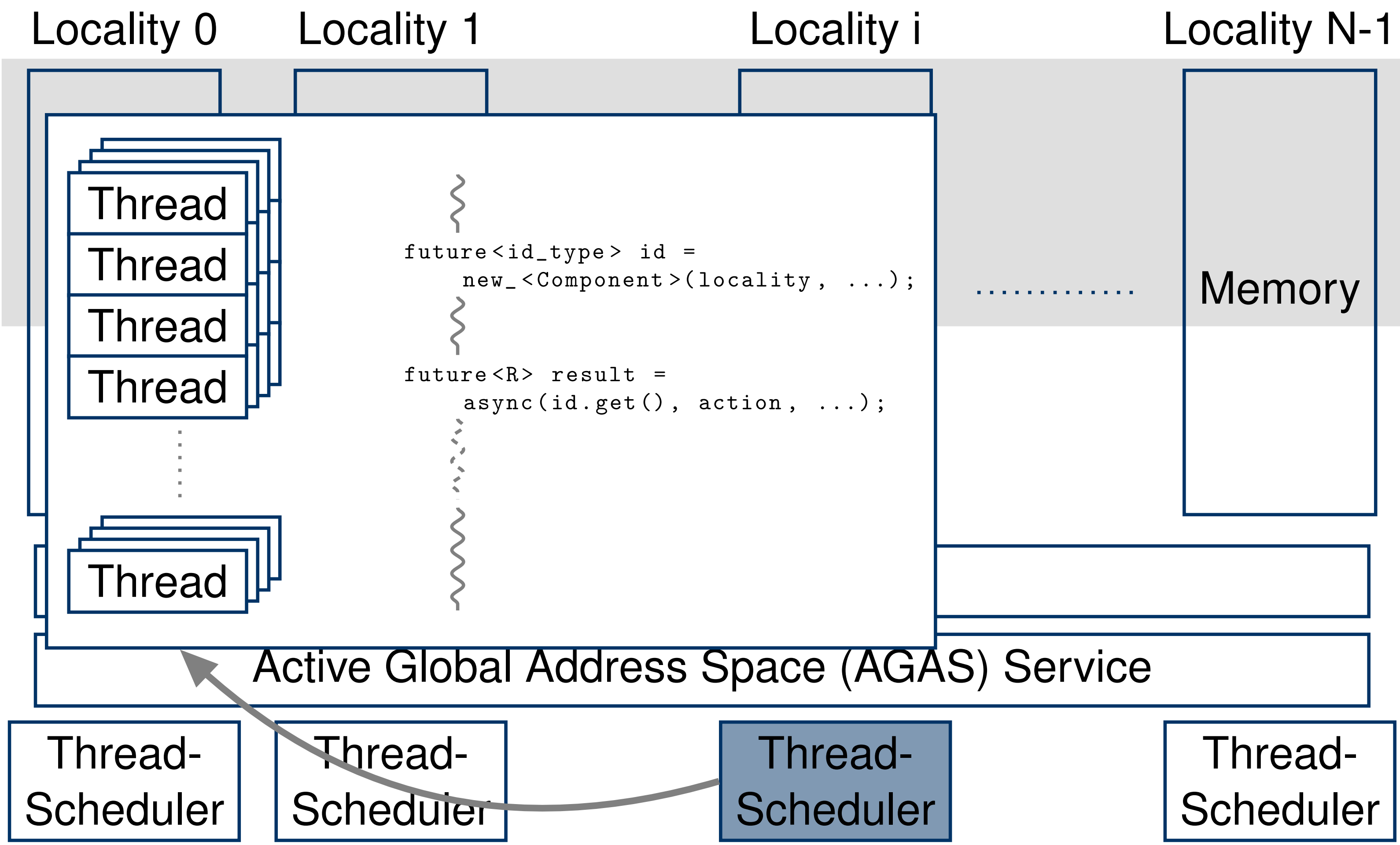
HPX – A general purpose parallel runtime system



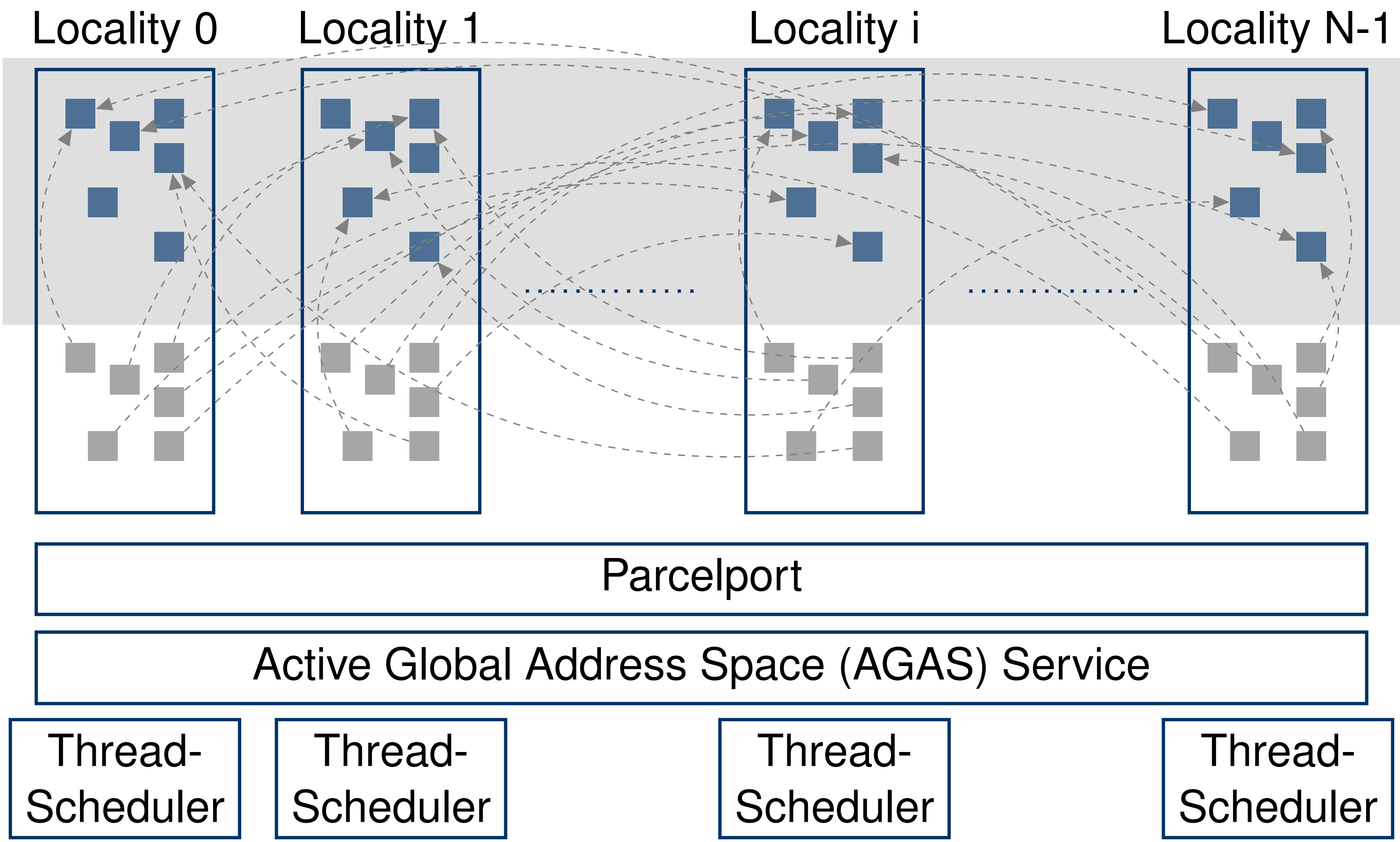
HPX – A general purpose parallel runtime system



HPX – A general purpose parallel runtime system



HPX – A general purpose parallel runtime system



Governing Principles

- Active global address space (AGAS)
- Message driven
- Lightweight Control Objects
- Adaptive locality control
- Moving work to data
- Fine grained parallelism of lightweight threads

HPX Threads

`R f(p...)`	Synchronous (returns `R`)	Asynchronous (returns `future<R>`)	Fire & Forget (returns `void`)
Functions (direct)	`f(p...)` C++	`async(f, p...)`	`apply(f, p...)`
Functions (lazy)	`bind(f, p...)(...)`	`async(bind(f, p...), ...)` C++ Standard Library	`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...), ...)` HPX

LOUISIANA STATE UNIVERSITY

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Future composability

Composable futures

- `hpx::when_all`, `hpx::when_any`, `hpx::when_n`
- `hpx::future<T>::then`
- `hpx::dataflow`

Expressing locality

- Executors let you specify where your tasks run and how they are scheduled

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::simple_component_base<
        hello_world_component
    >
{
    // ...
};
```

Components Interface: Writing a component

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

// Register component
typedef hpx::components::managed_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::simple_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

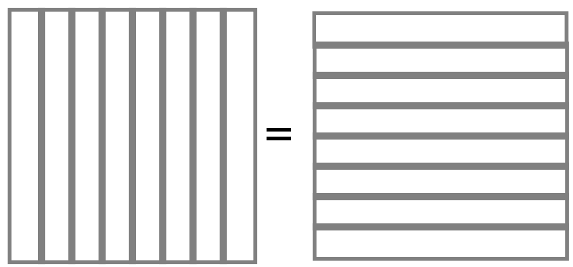
```
struct hello_world_component;

// 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();
}
```

Example 1: Matrix Transpose

$B = A^T \Rightarrow$



Example 1: Matrix Transpose

```
std::vector<double> A(order * order);
std::vector<double> B(order * order);

for(std::size_t i = 0; i < order; ++i)
{
  for(std::size_t j = 0; j < order; ++j)
  {
    B[i + order * j] = A[j + order * i];
  }
}
```

Example 1: Matrix Transpose

```
std::vector<double> A(order * order);
std::vector<double> B(order * order);

auto range = irange(0, order);
// parallel for
for_each(par, begin(range), end(range),
  [&](std::size_t i)
  {
    for(std::size_t j = 0; j < order; ++j)
    {
        B[i + order * j] = A[j + order * i];
    }
  }
);
```

Example 1: Matrix Transpose

```
std::size_t my_id = hpx::get_locality_id();
std::size_t num_blocks = hpx::get_num_localities().get();
std::size_t block_order = order / num_blocks;
std::vector<block> A(num_blocks);
std::vector<block> B(num_blocks);
```

Example 1: Matrix Transpose

```
for(std::size_t b = 0; b < num_blocks; ++b) {
  if(b == my_id) {
    A[b] = block(block_order * order);
    hpx::register_id_with_basename("A", get_gid(), b);
    B[b] = block(block_order * order);
    hpx::register_id_with_basename("B", get_gid(), b);
  }
  else {
    A[b] = hpx::find_id_from_basename("A", b);
    B[b] = hpx::find_id_from_basename("B", b);
  }
}
```

Example 1: Matrix Transpose

```
std::vector<hpx::future<void>> phases(num_blocks);
auto range = irange(0, num_blocks);
for_each(par, begin(range), end(range),
  [&](std::size_t phase)
  {
    std::size_t block_size = block_order * block_order;
    phases[b] = hpx::lcos::dataflow(
      transpose,
      A[phase].get_sub_block(my_id * block_size, block_size)
      B[my_id].get_sub_block(phase * block_size, block_size)
    );
  });
hpx::when_all(phases);
```

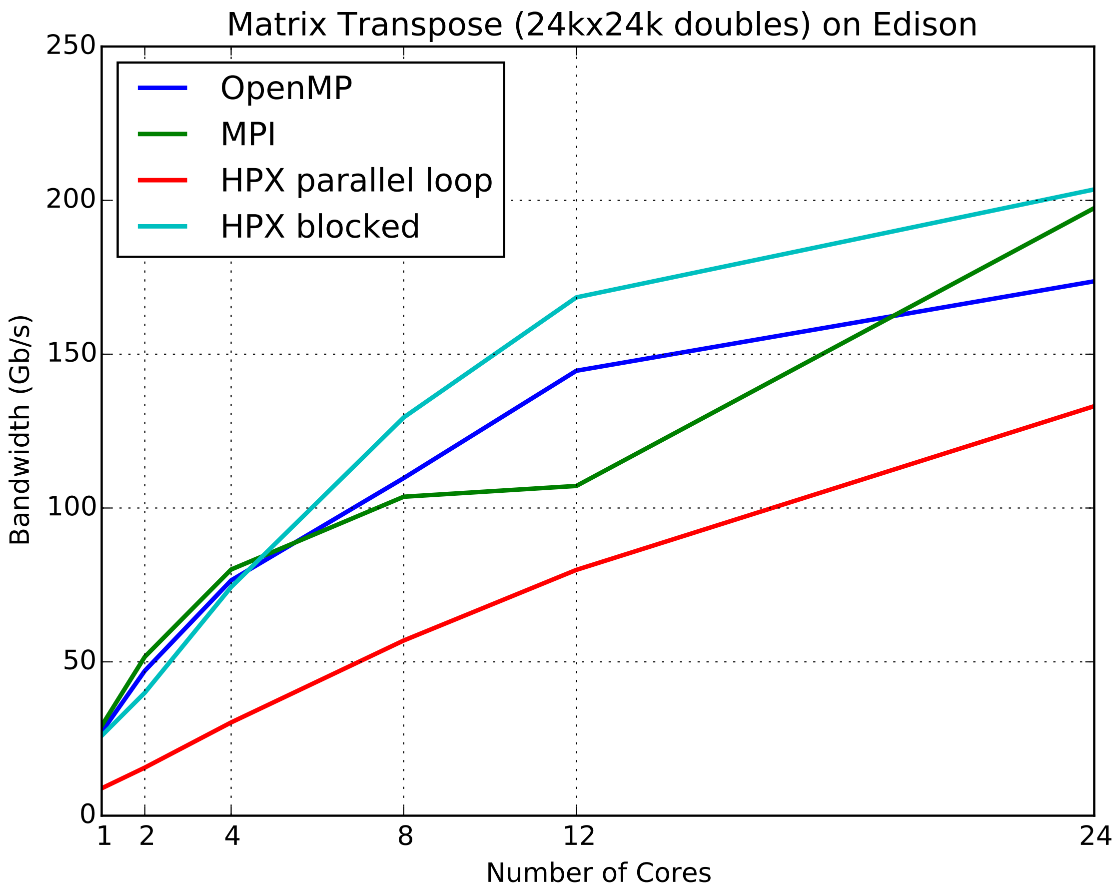
Example 1: Matrix Transpose

```
void transpose(hpx::future<sub_block> Af, hpx::future<sub_block> Bf)
{
  sub_block A = Af.get();
  sub_block B = Bf.get();
  for(std::size_t i = 0; i < block_order; ++i)
  {
    for(std::size_t j = 0; j < block_order; ++j)
    {
        B[i + block_order * j] = A[j + block_order * i];
    }
  }
}
```

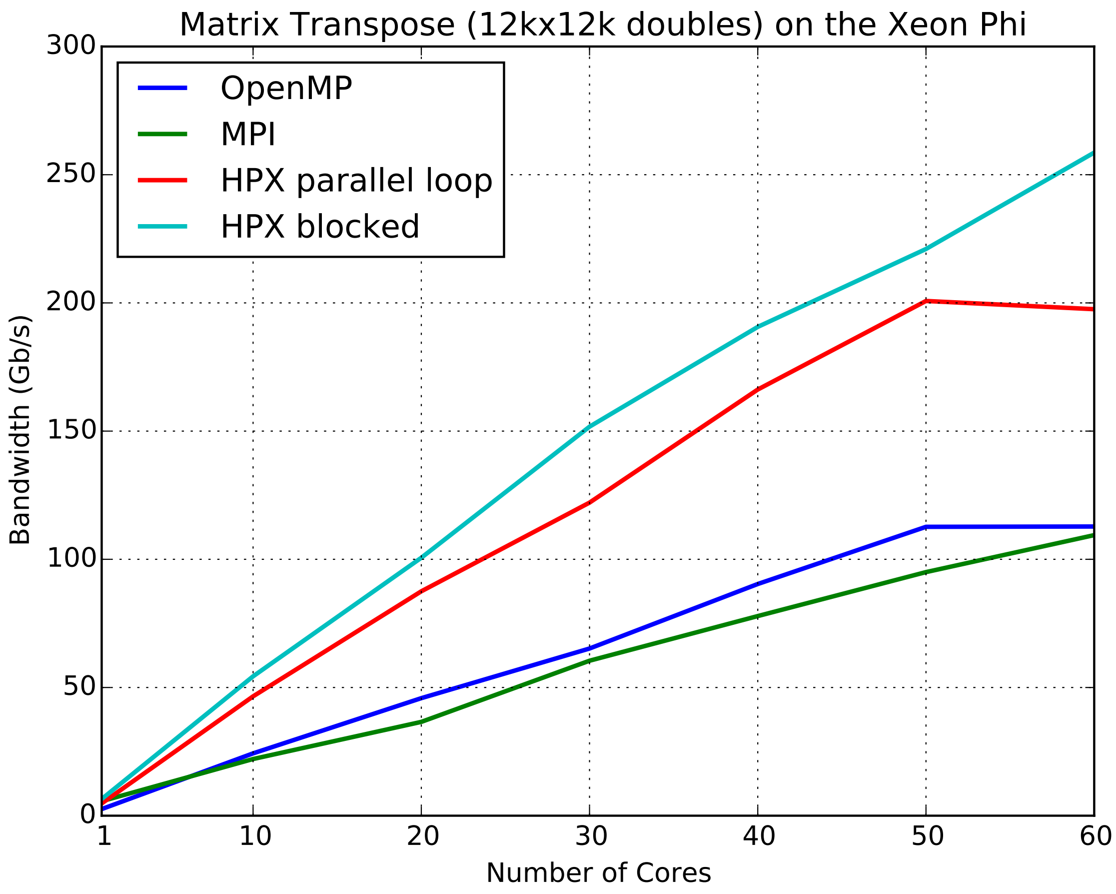
Example 1: Matrix Transpose

```
struct block_component
  : hpx::components::simple_component_base<block_component>
{
  block_component() {}
  block_component(std::size_t size)
    : data_(size) {}
  sub_block get_sub_block(std::size_t offset, std::size_t size)
  {
      return sub_block(&data_[offset], size);
  }
  HPX_DEFINE_COMPONENT_ACTION(block_component, get_sub_block);
  std::vector<double> data_;
};
```

Example 1: Matrix Transpose

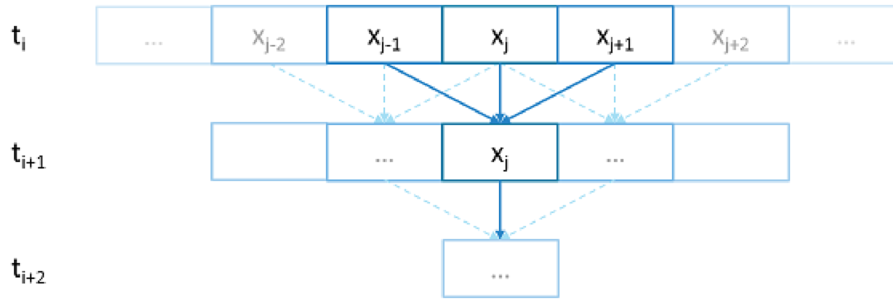


Example 1: Matrix Transpose



Example 2: 1D Heat equation

Solving $f = \Delta u$ using finite differences with a Jacobi-Solver:



Example 2: 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(grids[old]);
```

Example 2: 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 2: 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 2: 1D Heat equation

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

Example 2: 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 2: 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 2: 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 2: 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 2: 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);
}
```

Get in touch!

- Blog: `http://stellar-group.org`
- Code: `https://github.com/STEllAR-GROUP/hpx`
  Open Source (Boost Software License 1.0)
- Mailing List: `hpx-users@stellar.cct.lsu.edu`
- IRC: `#ste||ar @ irc.freenode.org`

LOUISIANA STATE UNIVERSITY

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT