Back to the Future

Thomas Heller[1], Agustín Bergé[2], Hartmut Kaiser[3]
May 13, 2015
The STE||AR Group

[1] thomas.heller@cs.fau.de, [2] agustinberge@gmail.com, [3] hkaiser@cct.lsu.edu

HPX – A C++ runtime system for applications of any scale

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.

Futures

- The Future in C++11/14
- The Future of Futures

The HPX Future Landscape

- Overview
- Local Tasks
- Remote Tasks
- GPU Tasks

Futures


STE||AR GROUP


FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Futures – Async return object

```
int sequential() {
  int x = compute_x();
  int y = compute_y();
  return x + y;
}

int concurrent() {
  future<int> x = async(compute_x);
  int y = compute_y();
  return x.get() + y;
}
```

Futures – Async provider

```
promise<int> p;
future<int> f = p.get_future();
thread t([p = move(p)]{
  /*...*/
  p.set_value(42);
});
/*...*/
int r = f.get();
/*...*/
t.join();
```

Futures – Shared State

```
template <class R>
struct state {
  enum {
    empty, has_value, has_exception
  } status = empty;

  union { R value; exception_ptr exception; };
  mutable mutex m;
  mutable condition_variable cv;
};

template <class R>
using shared_state = shared_ptr<state<R>>;
```

Futures – Shared State (provider)

```
template <class R> struct state {
  void set_value(R const& r) {
    {
      lock_guard<mutex> lock(m);
      new(&value) R(r);
      status = has_value;
    }
    cv.notify_all();
  }
  void set_value(R&& r) { /*...*/ }
  void set_exception(exception_ptr e) { /*...*/ }
};
```

Futures – Shared State (provider)

```
template <class R> class promise {
  shared_state<R> state = make_shared<state<R>>();

public:
  void set_value(R const& r) {
    state->set_value(r);
  }
  void set_value(R&& r) { /*...*/ }
  void set_exception(exception_ptr e) { /*...*/ }

  future<R> get_future() { /*...*/ }
};
```

Futures – Shared State (return object)

```
template <class R> struct state {
  R& get() {
    unique_lock<mutex> lock(m);
    cv.wait(lock, [&]{
      return status == has_value
          || status == has_exception;
    });
    if (status == has_exception)
      rethrow_exception(exception);
    return value;
  }
  void wait() const { /*...*/ }
};
```

Futures – Shared State (return object)

```
template <class R> class future {
  shared_state<R> state = nullptr;

public:
  bool valid() const noexcept { return state !=
  nullptr; }
  R get() {
    shared_state<R> state = move(this->state);
    return move(state->get());
  }
  void wait() const { /*...*/ }
};
```

The Future of Futures

Composition

```
int concurrent() {
  future<int> x = async(compute_x);
  int y = compute_y();
  return x.get() + y;
}

future<int> futurized() {
  future<int> x = async(compute_x);
  int y = compute_y();
  return
    async([x = move(x), y](){return x.get() + y;});
}
```

Composition

```
int concurrent() {
  future<int> x = async(compute_x);
  int y = compute_y();
  return x.get() + y;
}

future<int> futurized() {
  future<int> x = async(compute_x);
  int y = compute_y();
  return
    x.then([y](future<int> x){return x.get()+y;});
}
```

Composition – Shared State

```
template <class R> struct state {
  vector<function<void()>> callbacks;
  void attach_callback(function<void()> c) {
    {
      lock_guard<mutex> lock(m);
      if (status == empty) {
        callbacks.push_back(move(c));
        return;
      }
    }
    c();
  }
};
```

Composition – Shared State

```
template <class R> struct state {
  void set_value(R const& r) {
    {
      lock_guard<mutex> lock(m);
      new(&value) R(r);
      status = has_value;
    }
    cv.notify_all();
    for (auto& c : callbacks) c();
  }
};
```

Composition – Then

```
template <class Cont, class Fut>
struct then_state : state<result_of_t<Cont(Fut)>> {
  Cont c; Fut f;

  then_state(Cont const& c, Fut&& f) : c(c), f(move
    (f)) {
    f.state->attach_callback([&]{ run(); });
  }
  then_state(Cont&& c, Fut&& f) { /*...*/ }
};
```

Composition – Then

```
template <class Cont, class Fut>
struct then_state : state<result_of_t<Cont(Fut)>> {
  Cont c; Fut f;

  void run() {
    try {
      set_value(c(move(f)));
    } catch (...) {
      set_exception(current_exception());
    }
  }
};
```

Composition – Then

```
template <class R> class future {
public:
  template <class Cont> auto then(Cont&& c) {
    using CS
      = then_state<decay_t<Cont>, future<R>>;
    using CR
      = result_of_t<decay_t<Cont>(future<R>)>;
    return future<CR>(
      make_shared<CS>(forward<Cont>(c), move(*this)
        ));
  }
};
```

Composition – Then

```
future<int> futurized() {
  future<int> x = async(compute_x);
  int y = compute_y();
  return
    x.then([y](future<int> x){return x.get()+y;});
}

future<int> futurized() {
  future<int> x = async(compute_x);
  int y = compute_y();
  return await x + y;
}
```

Composition

```
vector<future<int>> fs;
for (int x : {1, 2, 3})
  fs.push_back(async(compute, x));

// parallel composition
future<vector<future<int>>> all =
  when_all(fs.begin(), fs.end());
```

Composition – When All

```
template <class Fut> struct when_all_state
  : state<vector<Fut>> {
  size_t index = 0;
  vector<Fut> result;

  template <class Iter>
  when_all_state(Iter first, Iter last)
    : result(first, last) {
    result[0].state->attach_callback(
      [&]{ on_future_ready(); });
    }
};
```

Composition – When All

```
template <class Fut> struct when_all_state
  : state<vector<Fut>> {

  void on_future_ready() {
    if (++index < result.size()) {
      result[index].state->attach_callback(
        [&]{ on_future_ready(); });
    } else {
      set_value(move(result));
    }
  }
};
```

Composition – When All

```
template <class Iter>
auto when_all(Iter first, Iter last) {
  using Fut = typename iterator_traits<Iter>::
    value_type;
  return future<vector<Fut>>(
    make_shared<when_all_state<Fut>>(first, last));
}

template <class ...Fs>
future<tuple<decay_t<Fs>...>>
when_all(Fs&&... fs) { /*...*/ }
```

Composition – When All

```
vector<future<int>> fs;
for (int x : {1, 2, 3})
  fs.push_back(async(compute, x));

// wait?
when_all(fs.begin(), fs.end()).wait();
fs = when_all(fs.begin(), fs.end()).get();

// wait
wait_all(fs.begin(), fs.end());
```

Composition – When Any

```
vector<future<int>> fs;
for (int x : {1, 2, 3})
  fs.push_back(async(compute, x));

// parallel composition
future<when_any_result<vector<future<int>>>> any
  = when_any(fs.begin(), fs.end());
```

Composition – When Some

```
vector<future<int>> fs;
for (int x : {1, 2, 3})
  fs.push_back(async(compute, x));

// parallel composition
future<when_some_result<vector<future<int>>>> some
  = when_some(1, fs.begin(), fs.end());
```

Composition – When Some

```
template <class Fut> struct when_some_state
  : state<when_some_result<vector<Fut>>> {
  atomic<size_t> count = 0; size_t goal;
  when_some_result<vector<Fut>> result;

  template <class Iter>
  when_some_state(size_t goal, Iter first, Iter
    last)
    : goal(goal), result(first, last) {
    for (size_t i = 0; i < result.futures.size();
      ++i)
      result.futures[i].state->attach_callback(
        [i]{ on_future_ready(i); });
  }
};
```


[1] thomas.heller@cs.fau.de, [2] agustinberge@gmail.com, [3] hkaiser@cct.lsu.edu

Composition – When Some

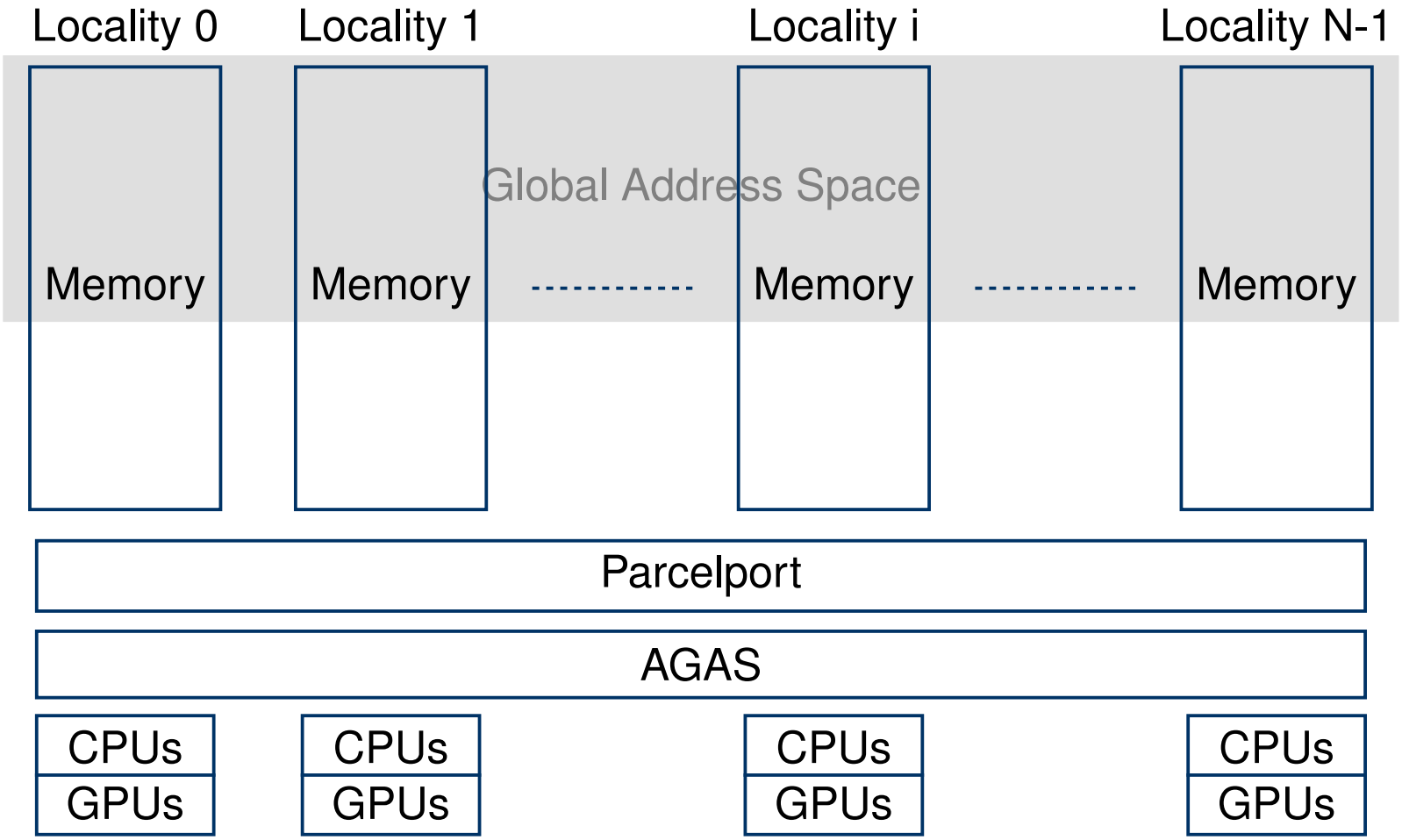
```
template <class Fut> struct when_some_state
  : state<when_some_result<vector<Fut>>> {

  void on_future_ready(size_t index) {
    if (count.fetch_add(1) + 1 <= goal) {
      lock_guard<mutex> lock(m);
        result.indices.push_back(index);
        if (result.indices.size() == goal)
          set_value(move(result));
    }
  }
};
```

Composition – When Some

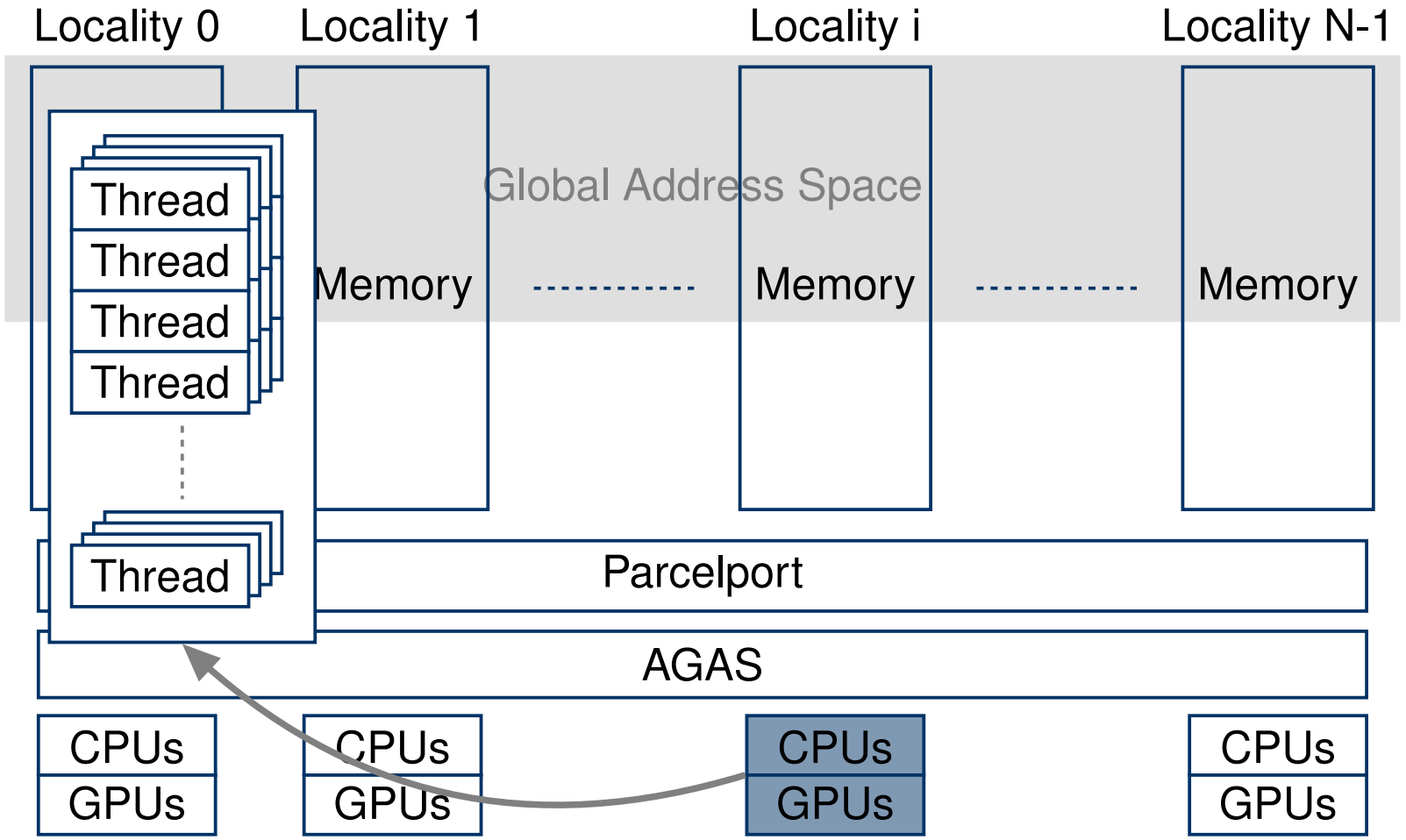
```
template <class Iter>
auto when_some(size_t goal, Iter first, Iter last)
  {
   using Fut = typename iterator_traits<Iter>::
     value_type;
   return future<when_some_result<vector<Fut>>>(
     make_shared<when_some_state<Fut>>(goal, first,
       last));
}

template <class ...Fs>
future<when_some_result<tuple<decay_t<Fs>...>>>
when_some(size_t goal, Fs&&... fs) { /*...*/ }
```

The HPX Future Landscape

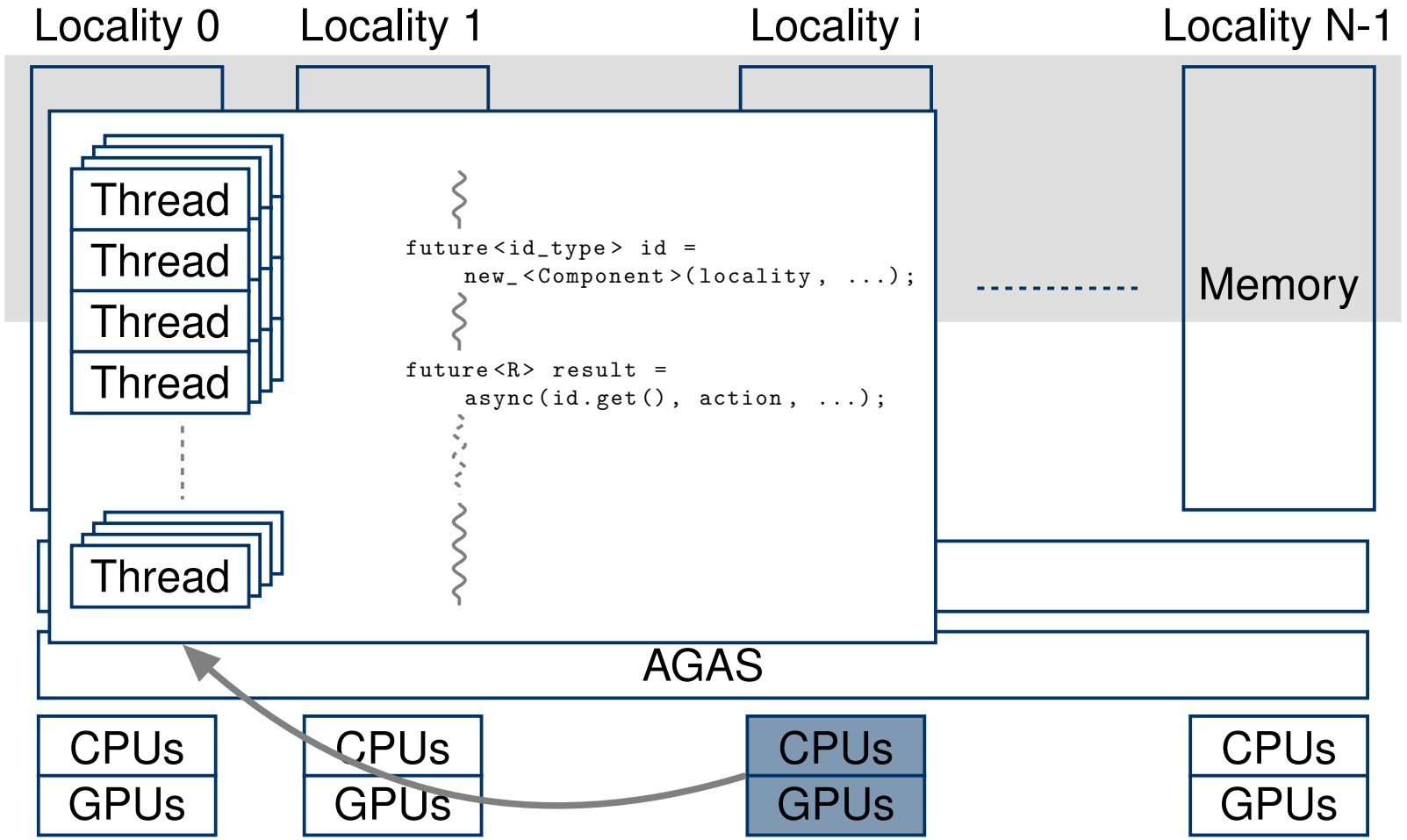
HPX – Overview



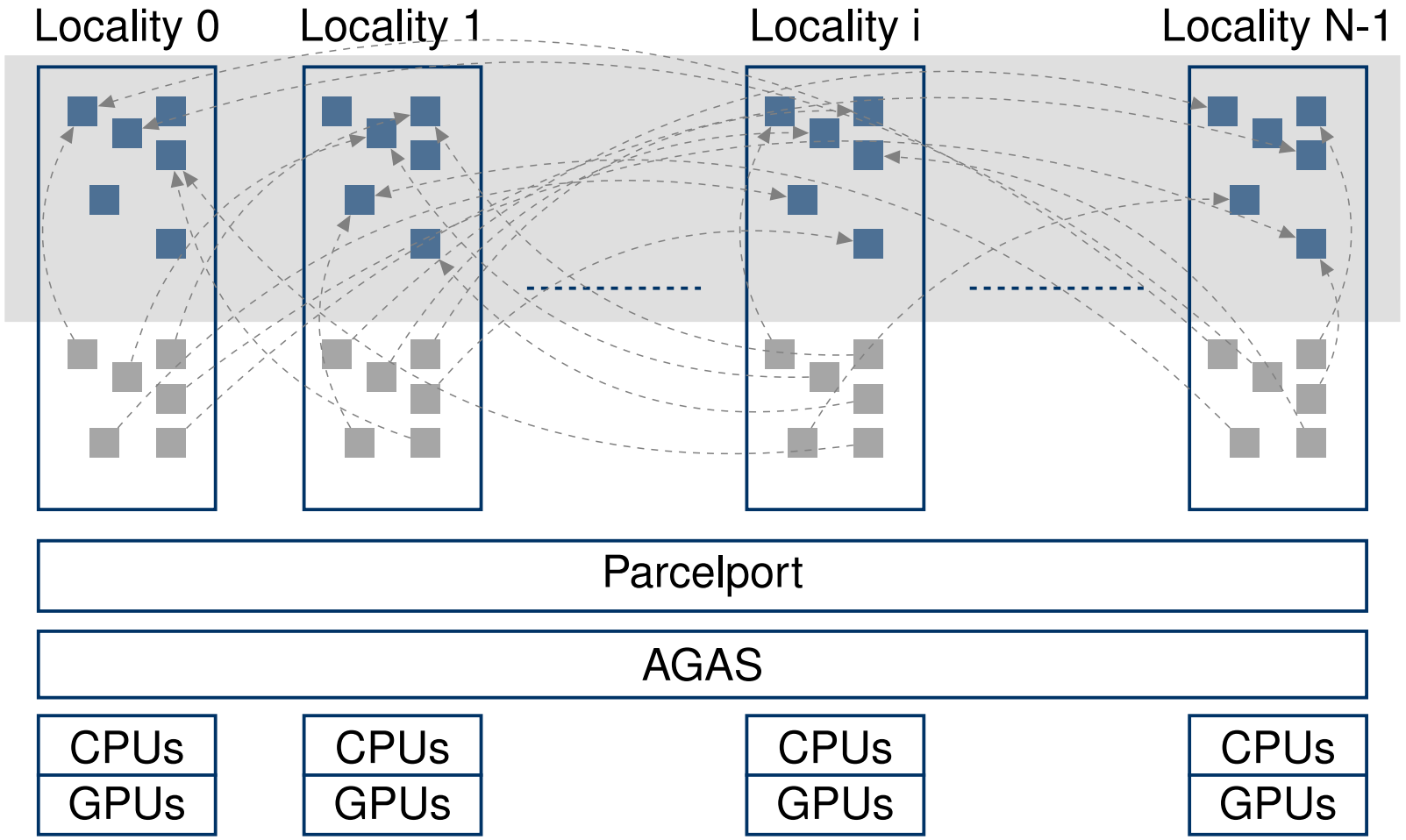
HPX – Overview



HPX – Overview



HPX – Overview



HPX Task Invocation 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...), ...)`

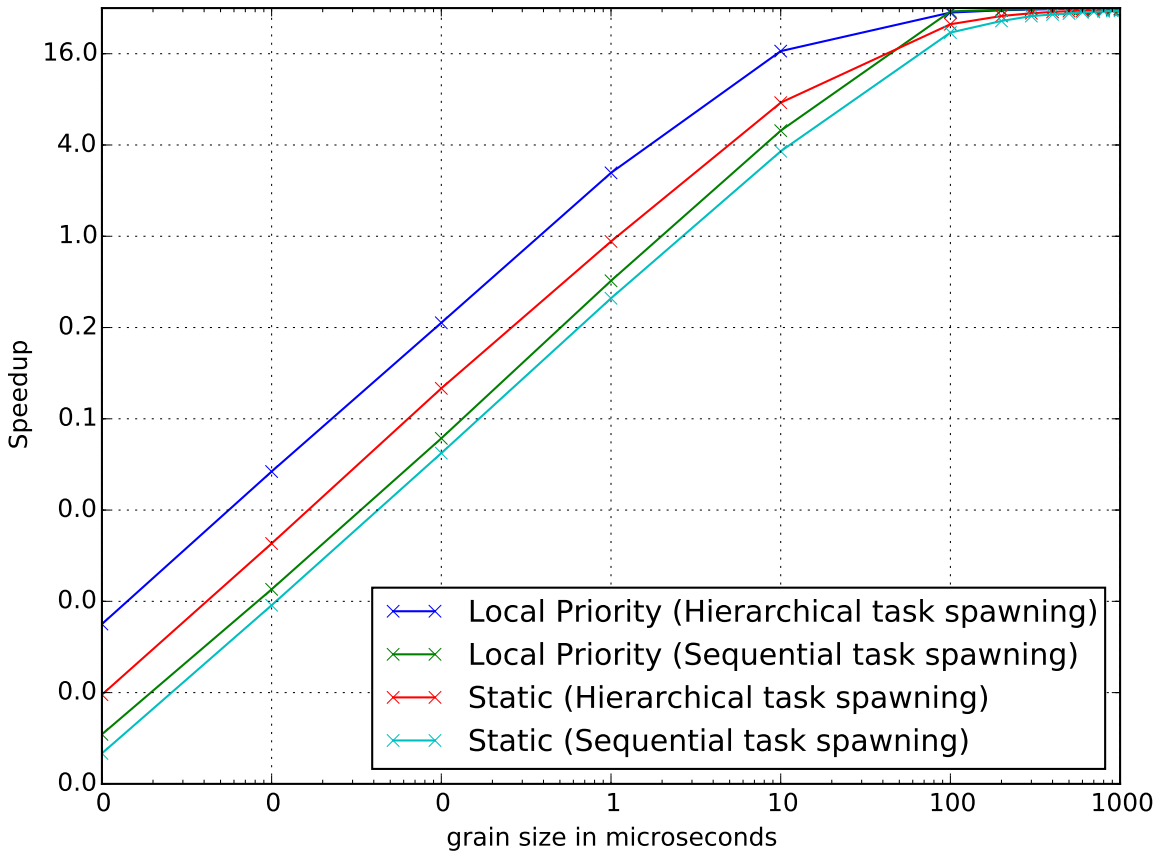
C++ | C++ Standard Library | HPX

Local Tasks

Lightweight tasks and scheduling

- Leightweight Task
  - User level context switching (think Boost.Context)
  - Each task has its own stack
- Task scheduling
  - Weakly parallel forward progress guarantee
  - Possible work stealing between cores
  - User defined (FIFO, LIFO, lockfree etc.)
  - Determined by executors

Grain Size


Speedup
16.0
4.0
1.0
0.2
0.1
0.0
0.0
0.0
0.0
0
0
0
1
10
100
1000
grain size in microseconds
Local Priority (Hierarchical task spawning)
Local Priority (Sequential task spawning)
Static (Hierarchical task spawning)
Static (Sequential task spawning)

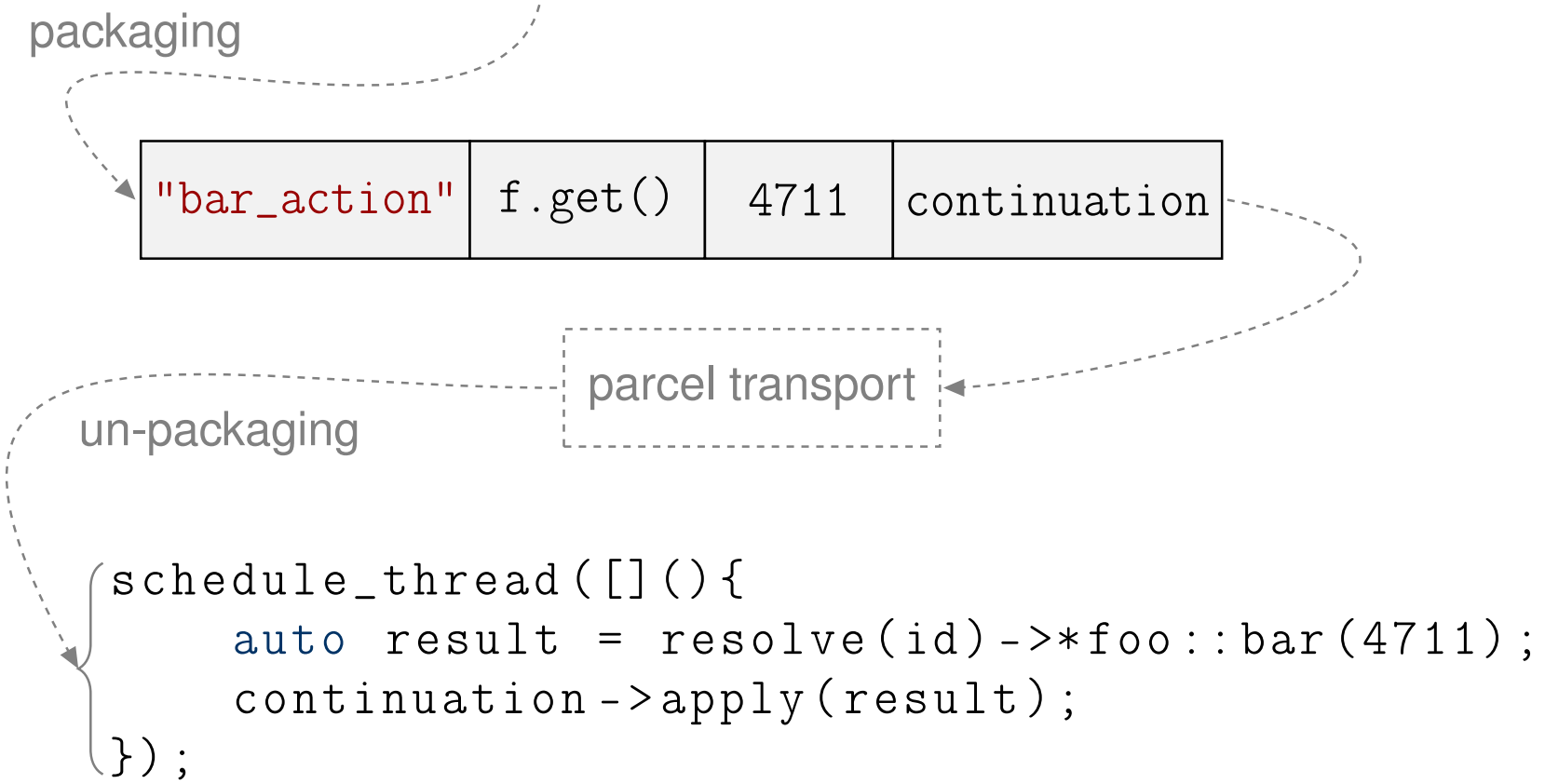
Remote Tasks

From `async` to distributed memory

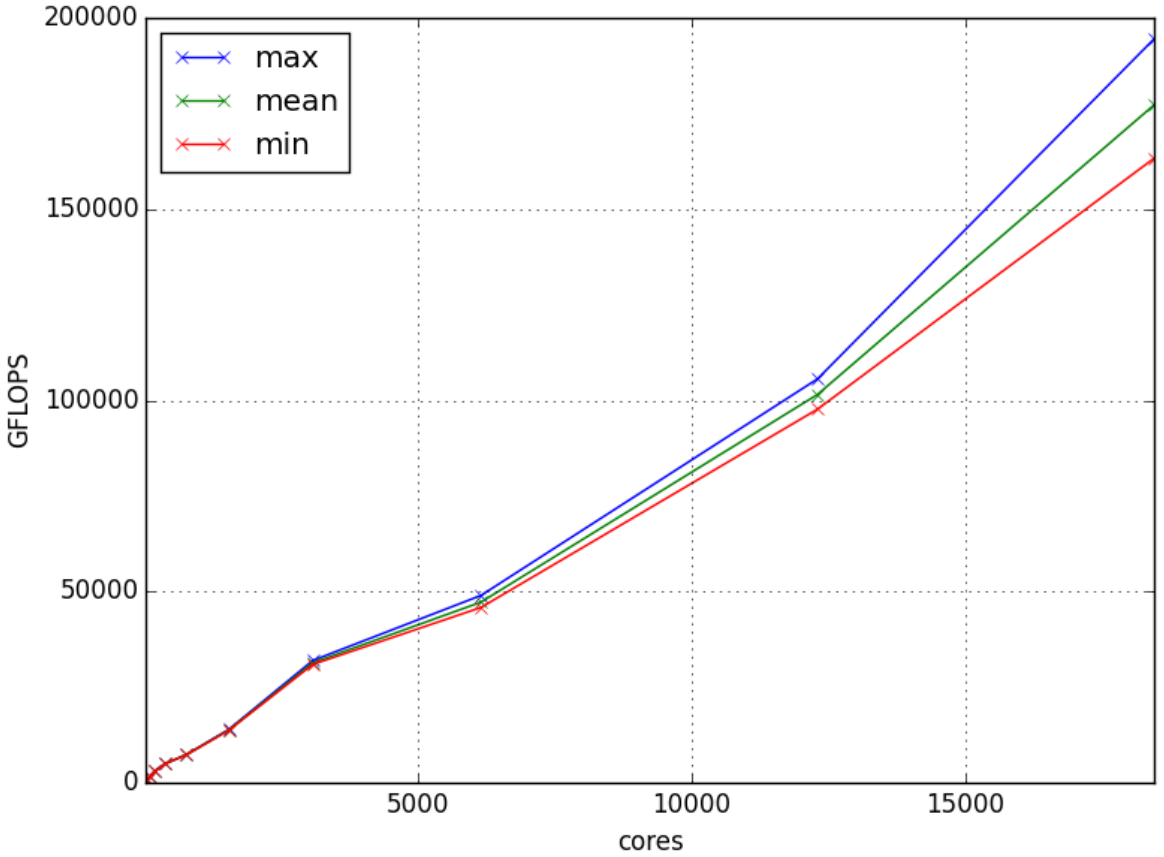
```
// local task calling a member function
foo f;
async(&foo::bar, &f, 4711);

// (possibly) remote task calling a member function
HPX_COMPONENT_ACTION(foo, bar);
future<id_type> f = new_<foo>(somewhere);
async(bar_action, f.get(), 4711);
```

Getting stuff over the wire

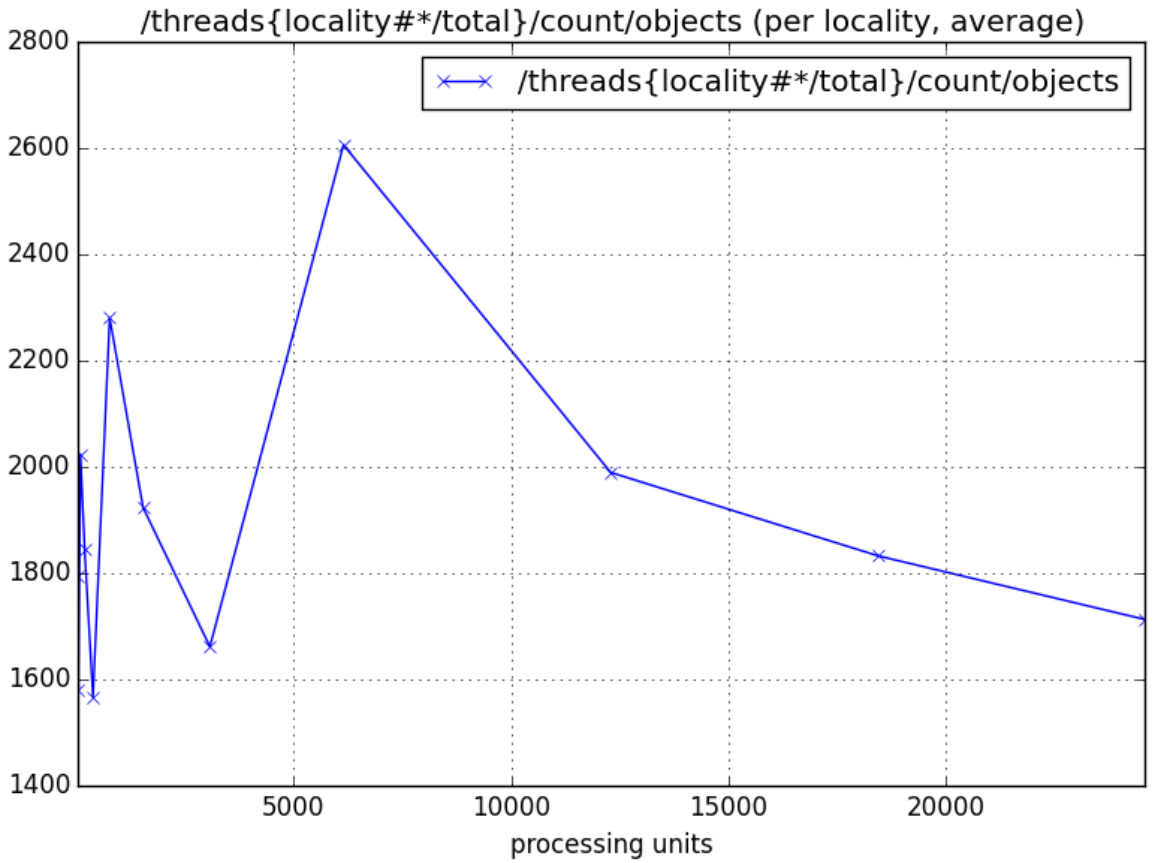
```
async(bar_action, f.get(), 4711);
```



Performant implementation for large scale applications



Performant implementation for large scale applications



GPU Tasks

Extending the Global Adress Space

- All GPU devices are addressable globally
- GPU memory can be allocated and referenced remotely
- Events are extensions of the shared state

$\Rightarrow$ API embedded into the already existing future facilities

From `async` to GPUs

Spawning single tasks not feasible

$\Rightarrow$ offload a `work group` (Think of `parallel::for_each`)

```
auto devices
    = hpx::opencl::find_devices(hpx::find_here(),
      CL_DEVICE_TYPE_GPU).get();
// create buffers, programs and kernels ...
hpx::opencl::buffer buf = devices[0].create_buffer(
  CL_MEM_READ_WRITE, 4711);
auto write_future = buf.enqueue_write(some_vec.
  begin(), some_vec.end());
auto kernel_future = kernel.enqueue(dim,
  write_future);
```

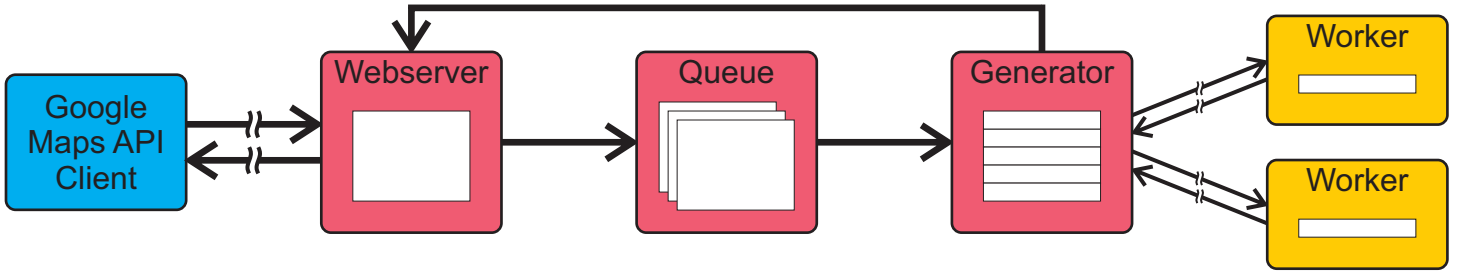
From `async` to GPUs

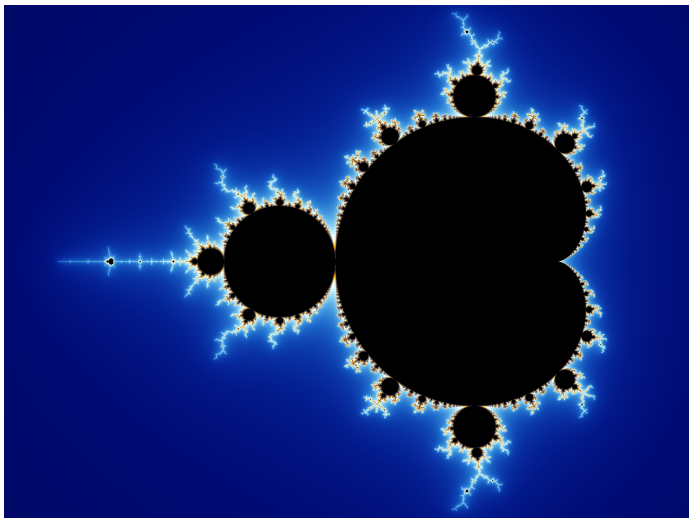
Spawning single tasks not feasible

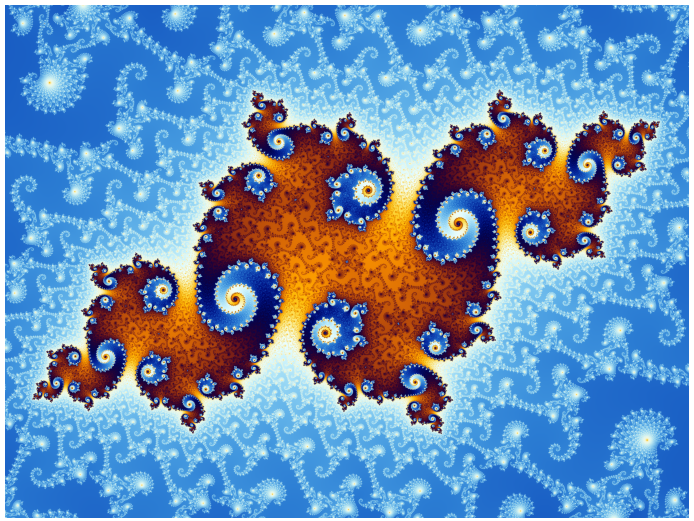
$\Rightarrow$ offload a `work group` (Think of `parallel::for_each`)

- Proof of Concept
- Future Directions:
  - Embedd OpenCL devices behind Execution Policies and Executors
  - Hide OpenCL stuff behind parallel algorithms
  - Hide OpenCL buffer management behind "distributed data structures"

Mandelbrot example

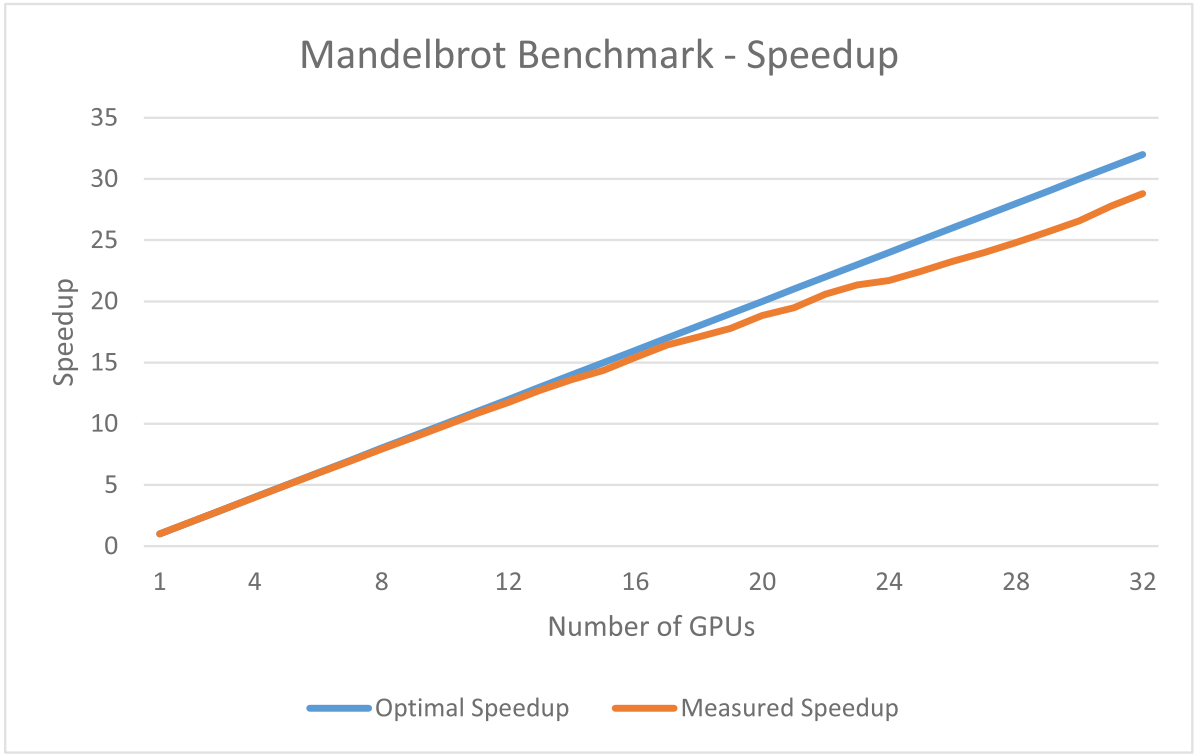






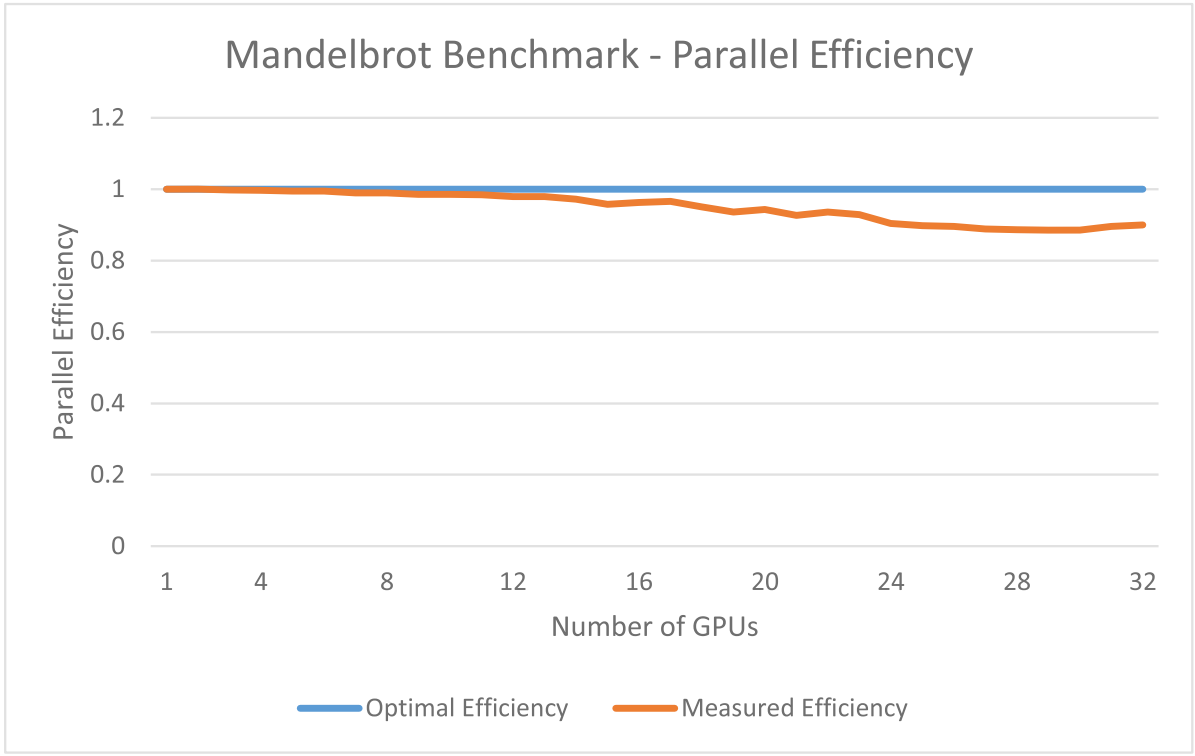
Acknowledgements to Martin Stumpf

Mandelbrot example



Acknowledgements to Martin Stumpf

Mandelbrot example



Acknowledgements to Martin Stumpf

More Information

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

- Visit Grant Mercers' talk on Parallelizing the STL