# Diving into Parallelization in Modern C++ using HPX

Thomas Heller, Hartmut Kaiser, John Biddiscombe – CERN 2016
May 18, 2016
Computer Architecture – Department of Computer Science

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

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

State of the Art

The Future – The HPX Programming Model

Examples:

- Fibonacci
- Simple Loop Parallelization
- SAXPY routine
- Hello Distributed World!
- Matrix Transpose

# State of the Art

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

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

# State of the Art

- Modern architectures impose massive challenges on programmability in the context of performance portability
  - Massive increase in on-node parallelism
  - Deep memory hierarchies
- Only portable parallelization solution for C++ programmers (today): TBB, OpenMP and MPI
  - Hugely successful for years
  - Widely used and supported
  - Simple use for simple use cases
  - Very portable
  - Highly optimized

# The C++ Standard – Our Vision

Currently there is no overarching vision related to higher-level parallelism

- Goal is to standardize a ‘big story’ by 2020
- No need for OpenMP, OpenACC, OpenCL, MPI, etc.

# HPX – A general purpose parallel Runtime System

- Exposes a coherent and uniform, standards-oriented API for ease of programming parallel and distributed applications.
  - Enables to write fully asynchronous code using hundreds of millions of threads.
  - Provides unified syntax and semantics for local and remote operations.
- Developed to run at any scale
- Compliant C++ Standard implementation (and more)
- Open Source: Published under the Boost Software License

# HPX – A general purpose parallel Runtime System

HPX represents an innovative mixture of

- A global system-wide address space (AGAS - Active Global Address Space)
- Fine grain parallelism and lightweight synchronization
- Combined with implicit, work queue based, message driven computation
- Full semantic equivalence of local and remote execution, and
- Support for hardware accelerators

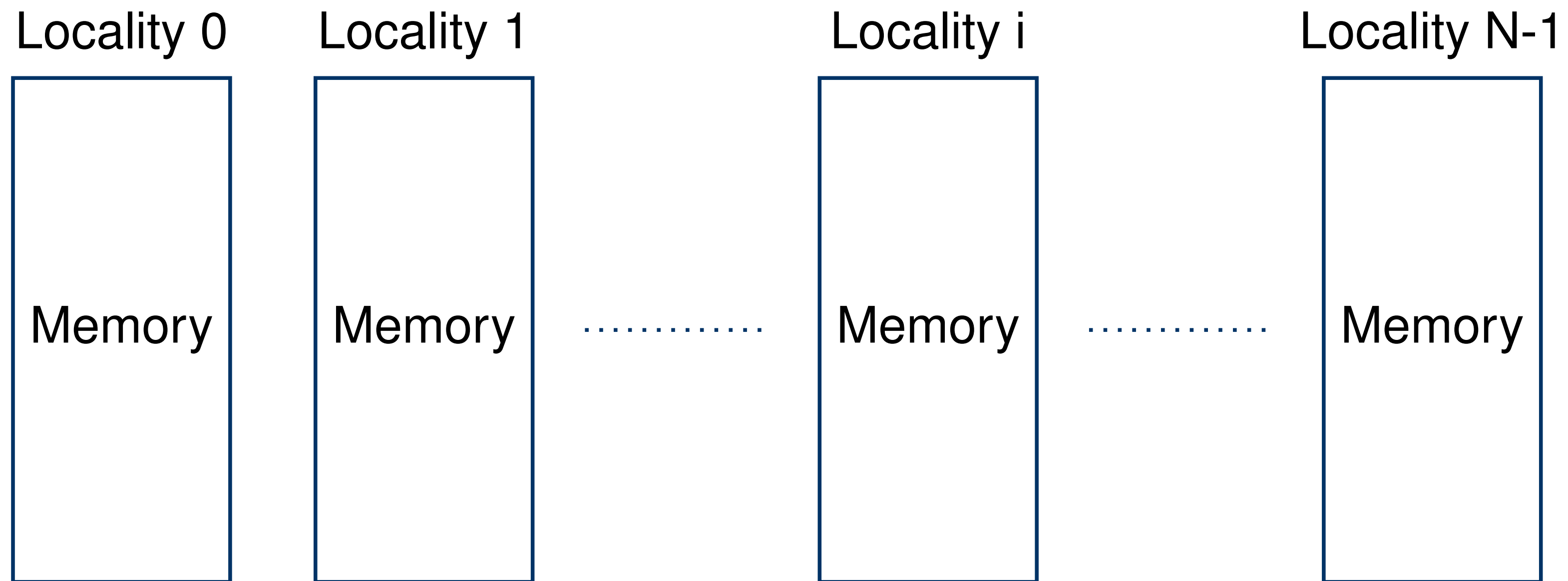
# The Future – The HPX Programming Model

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

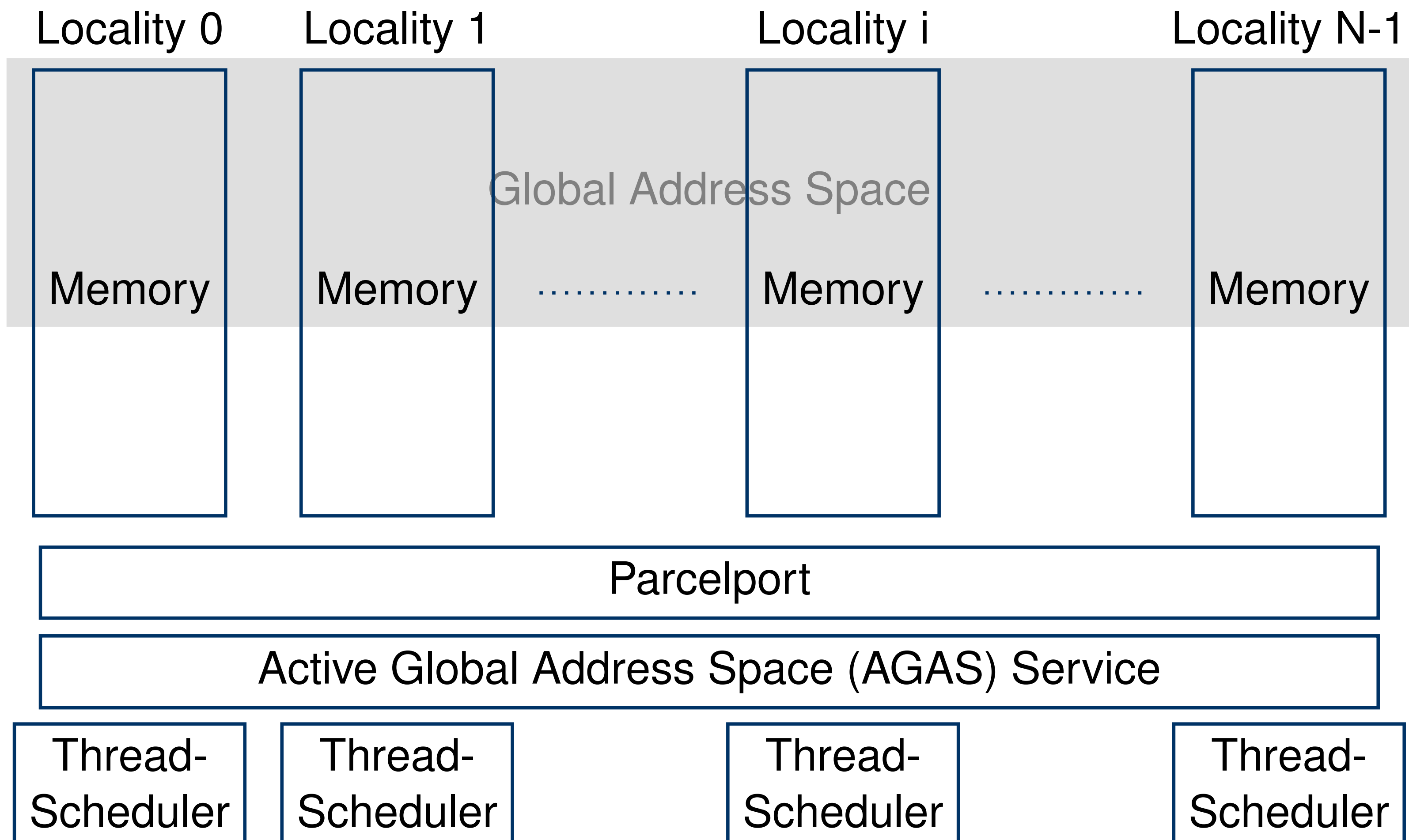
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

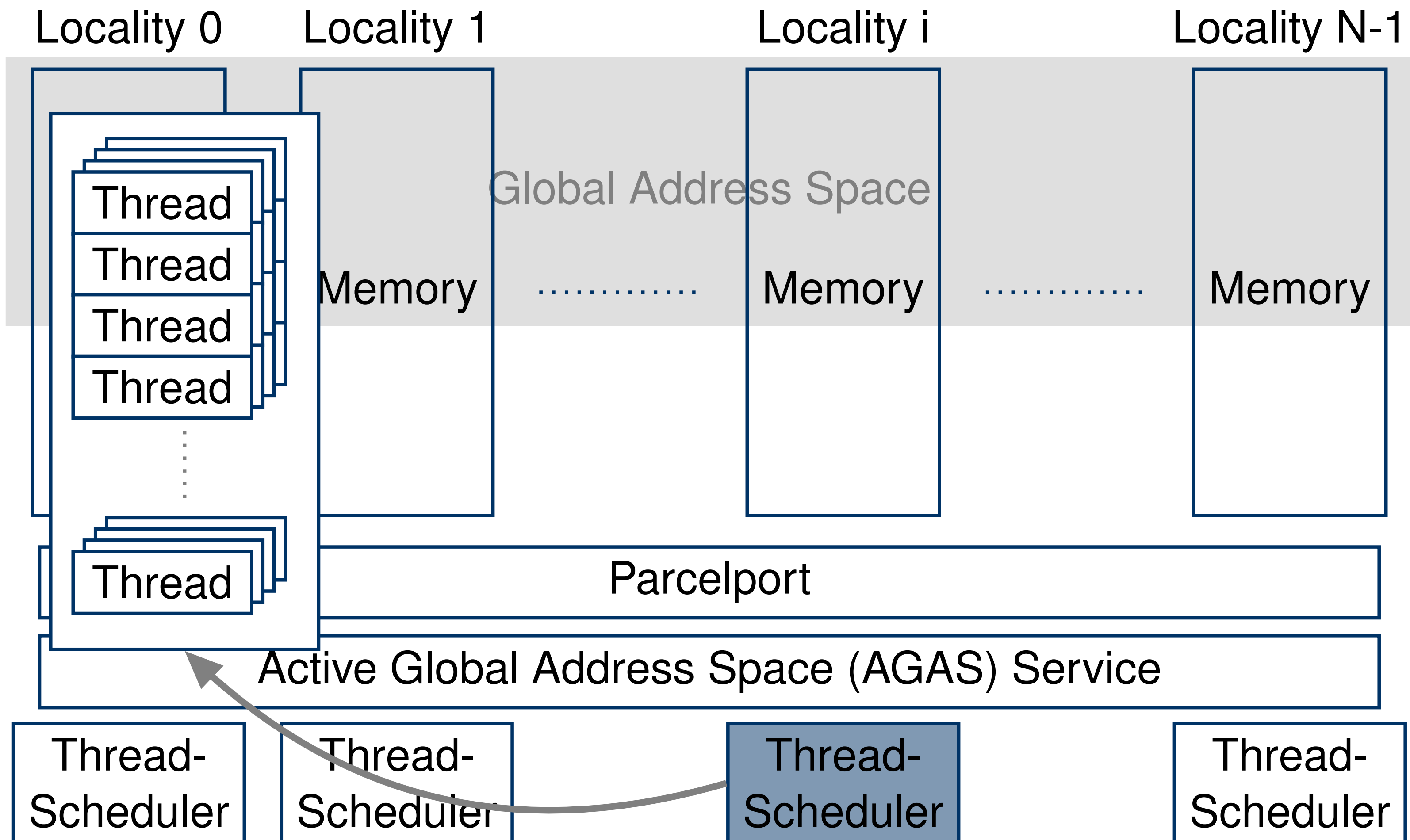
# HPX – The programming model



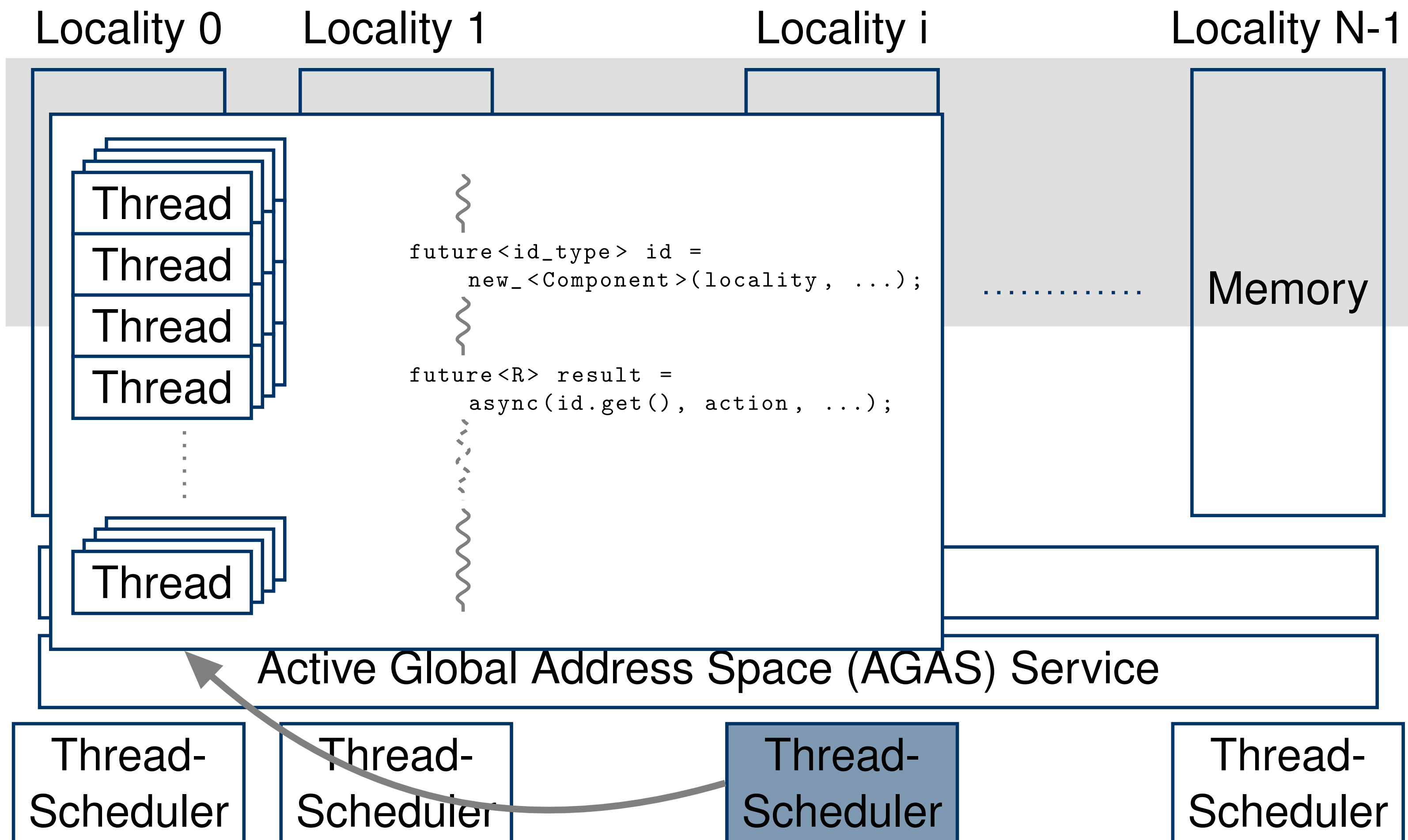
# HPX – The programming model



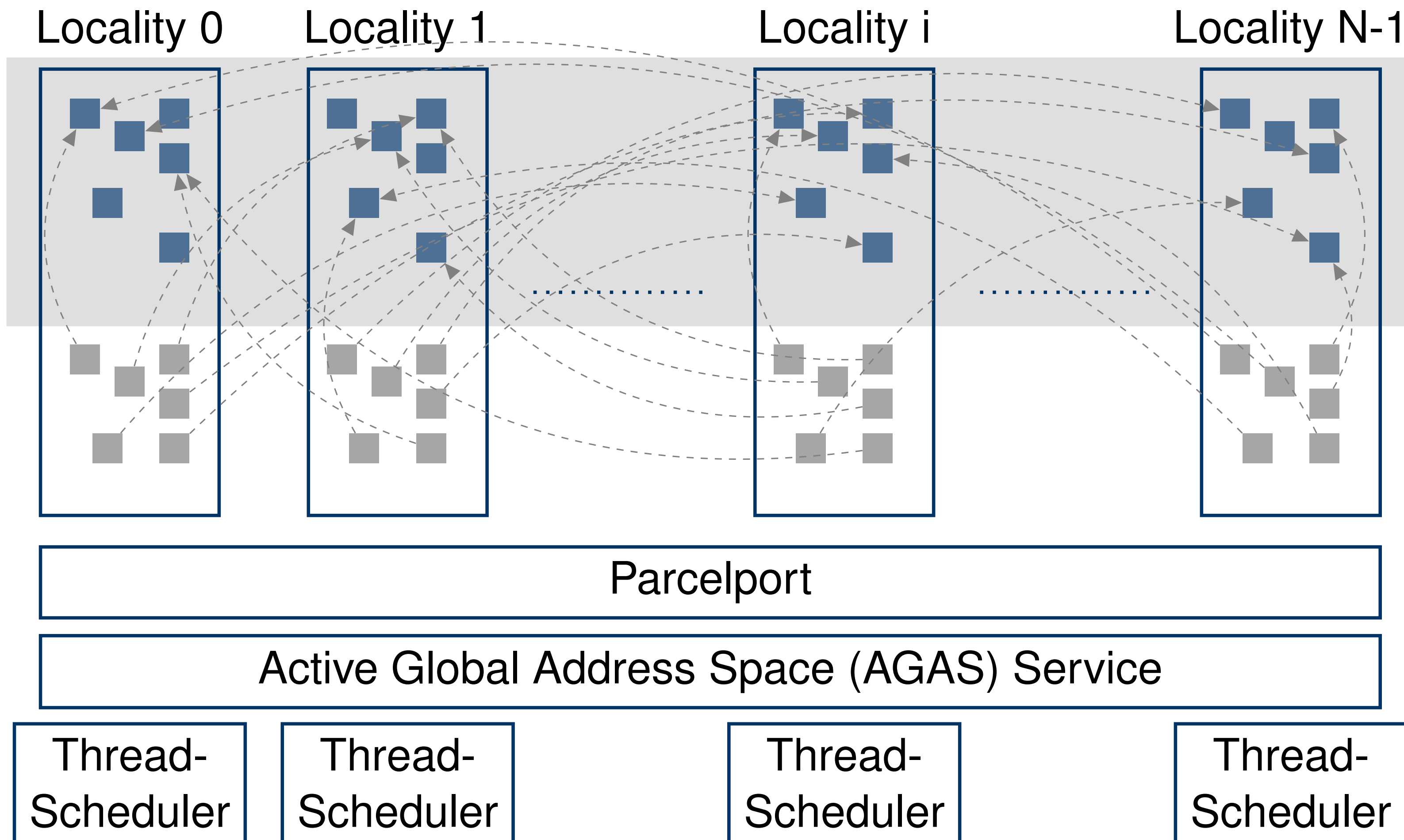
# HPX – The programming model



# HPX – The programming model



# HPX – The programming model



# HPX 101 – API Overview

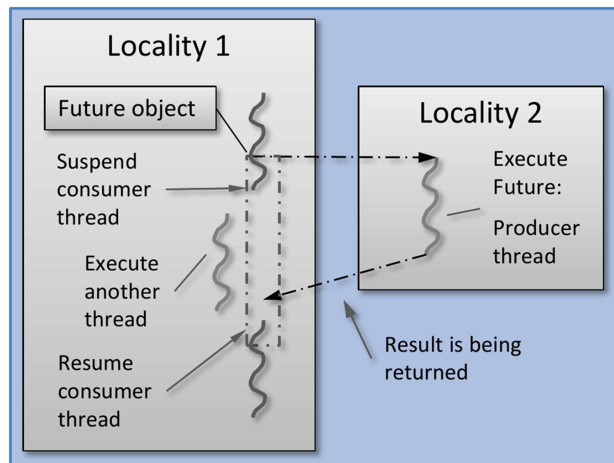
| `R f(p...)` | Synchronous (returns `R`) | Asynchronous (returns `future<R>`) | Fire & Forget (returns `void`) |
|---|---|---|---|
| Functions (direct) | `f(p...)` | `async(f, p...)` | `apply(f, p...)` |
| Functions (lazy) | `bind(f, p...)(...)` | `async(bind(f, p...), ...)` | `apply(bind(f, p...), ...)` |
| 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);`

# What is a (the) future

A future is an object representing a result which has not been calculated yet




- Enables transparent synchronization with producer
- Hides notion of dealing with threads
- Makes asynchrony manageable
- Allows for composition of several asynchronous operations
- Turns concurrency into parallelism

# What is a (the) future

Many ways to get hold of a future, simplest way is to use (std) async:

```
int universal_answer() { return 42; }
void deep_thought() {
    future<int> promised_answer
        = async(&universal_answer);
    // do other things for 7.5 million years
    cout << promised_answer.get() << endl;
    // prints 42, eventually
}
```

# The Future – Examples

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

# The Future – Examples

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

// 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!");
```

# The Future – Examples

```
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_action
hpx::future<void> f2
  = hpx::async(hello_world_action, hpx::find_here(), "Hello HPX!");
```

# Compositional facilities

Sequential composition of futures:

```
future<string> make_string() {
  future<int> f1 =
    async([]() -> int { return 123; });
  future<string> f2 = f1.then(
    [](future<int> f) -> string
    {
      // here .get() won't block
      return to_string(f.get());
    });
  return f2;
}
```

# Compositional facilities

## Parallel composition of futures

```
future<int> test_when_all() {
  future<int> future1 =
    async([]() -> int { return 125; });
  future<string> future2 =
    async([]() -> string { return string("hi"); });
  auto all_f = when_all(future1, future2);
  future<int> result = all_f.then(
    [](auto f) -> int {
      return do_work(f.get());
    });
  return result;
}
```

# Dataflow – The new 'async' (HPX)

- What if one or more arguments to 'async' are futures themselves?
- Normal behavior: pass futures through to function
- Extended behavior: wait for futures to become ready before invoking the function:

```
template <typename F, typename... Arg>
future<result_of_t<F(Args...)>>
// requires(is_callable<F(Arg...)>)
dataflow(F && f, Arg &&... arg);
```

- If ArgN is a future, then the invocation of F will be delayed
- Non-future arguments are passed through

# Examples:

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

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# Fibonacci

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

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# 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;
}
```

# Simple Loop Parallelization

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

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# Loop parallelization

```
// Serial version

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

# 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);
for_each(
  par, begin(range), end(range),
  [](int i) {
    do_work(i);
  });
```

# Loop parallelization

```
// Serial version

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

```
// Task parallel version

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

# SAXPY routine

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

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# SAXPY routine

- $a[i] = b[i] * x + c[i]$, for $i$ from 0 to $N - 1$
- Using parallel algorithms
- Explicit Control over data locality
- No raw Loops

# SAXPY routine

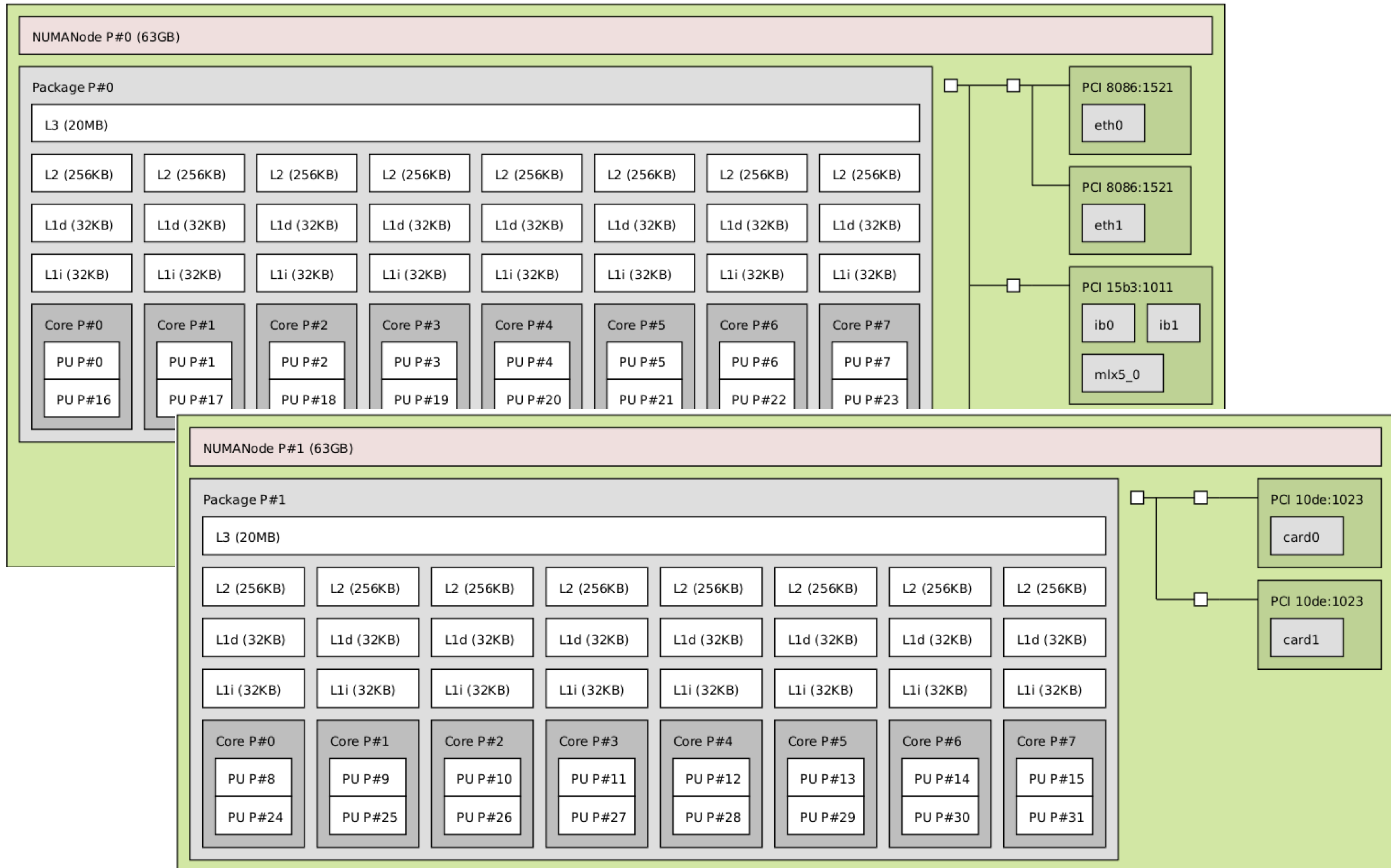
Complete serial version:

```
std::vector<double> a = ...;
std::vector<double> b = ...;
std::vector<double> c = ...;
double x = ...;

std::transform(b.begin(), b.end(),
  c.begin(), c.end(), a.begin(),
  [x](double bb, double cc)
  {
      return bb * x + cc;
  });
```

# SAXPY routine

Parallel version, no data locality:

```
std::vector<double> a = ...;
std::vector<double> b = ...;
std::vector<double> c = ...;
double x = ...;

parallel::transform(parallel::par,
  b.begin(), b.end(),
  c.begin(), c.end(), a.begin(),
  [x](double bb, double cc)
  {
      return bb * x + cc;
  });
```


NUMANode P#0 (63GB)
Package P#0
L3 (20MB)
L2 (256KB)
L1d (32KB)
L1i (32KB)
Core P#0
PU P#0
PU P#16
Core P#1
PU P#1
PU P#17
Core P#2
PU P#2
PU P#18
Core P#3
PU P#3
PU P#19
Core P#4
PU P#4
PU P#20
Core P#5
PU P#5
PU P#21
Core P#6
PU P#6
PU P#22
Core P#7
PU P#7
PU P#23
PCI 8086:1521
eth0
PCI 8086:1521
eth1
PCI 15b3:1011
ib0
ib1
mlx5_0
NUMANode P#1 (63GB)
Package P#1
L3 (20MB)
L2 (256KB)
L1d (32KB)
L1i (32KB)
Core P#0
PU P#8
PU P#24
Core P#1
PU P#9
PU P#25
Core P#2
PU P#10
PU P#26
Core P#3
PU P#11
PU P#27
Core P#4
PU P#12
PU P#28
Core P#5
PU P#13
PU P#29
Core P#6
PU P#14
PU P#30
Core P#7
PU P#15
PU P#31
PCI 10de:1023
card0
PCI 10de:1023
card1

# SAXPY routine

Parallel version, no data locality:

```
std::vector<hpx::compute::host::target> target = hpx::compute::host::get_numa_domains();

hpx::compute::host::block_allocator<double> alloc(targets);

hpx::compute::vector<double, block_allocator<double>> a(..., alloc);
hpx::compute::vector<double, block_allocator<double>> b(..., alloc);
hpx::compute::vector<double, block_allocator<double>> c(..., alloc);
double x = ...;

hpx::compute::host::block_executor<> numa_executor(targets);

auto policy = parallel::par.on(numa_executor)

parallel::transform(
    policy,
    b.begin(), b.begin(), c.begin(),
    a.begin(),
    [x](double bb, double cc)
    { return bb * x + cc; });
```

# SAXPY routine

Parallel version, running on the GPU:

```
hpx::compute::cuda::target target = hpx::compute::cuda::get_default_device();

hpx::compute::host::cuda_allocator<double> alloc(target);

hpx::compute::vector<double, block_allocator<double>> a(..., alloc);
hpx::compute::vector<double, block_allocator<double>> b(..., alloc);
hpx::compute::vector<double, block_allocator<double>> c(..., alloc);
double x = ...;

hpx::compute::host::cuda_executor executor(target);

auto policy = parallel::par.on(executor)

parallel::transform(
    policy,
    b.begin(), b.begin(), c.begin(),
    a.begin(),
    [x](double bb, double cc)
    { return bb * x + cc; });
```

# More on HPX GPU support

- Executors to modify behavior of how the warps are scheduled
- Executor Parameters to modify chunking (partitioning) of parallel work
- Dynamic parallelism: `hpx::parallel::sort(....); hpx::async(cuda_exec, [&]()`

# More on HPX data locality

- The goal is to be able to expose high level support for all kinds of memory:
  - Scratch Pads
  - High Bandwidth Memory (KNL)
  - Remote Targets (memory locations)
- Targets are the missing link between where data is executed, and where it is located

# Hello Distributed World!

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

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# Hello Distributed World!

```
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();
}
```

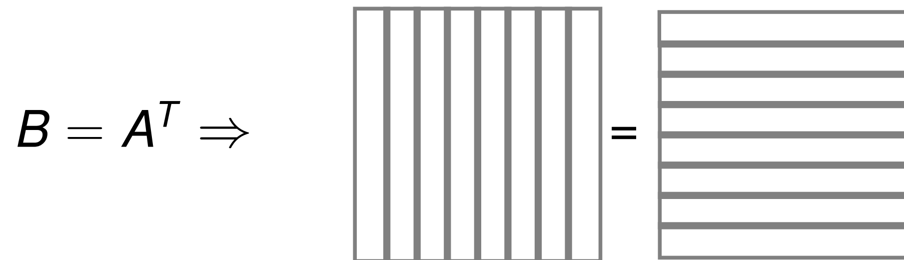
# Matrix Transpose

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

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# Matrix Transpose




Inspired by the Intel Parallel Research Kernels (https://github.com/ParRes/Kernels)

# 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: 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: 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: 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: 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::wait_all(phases);
```

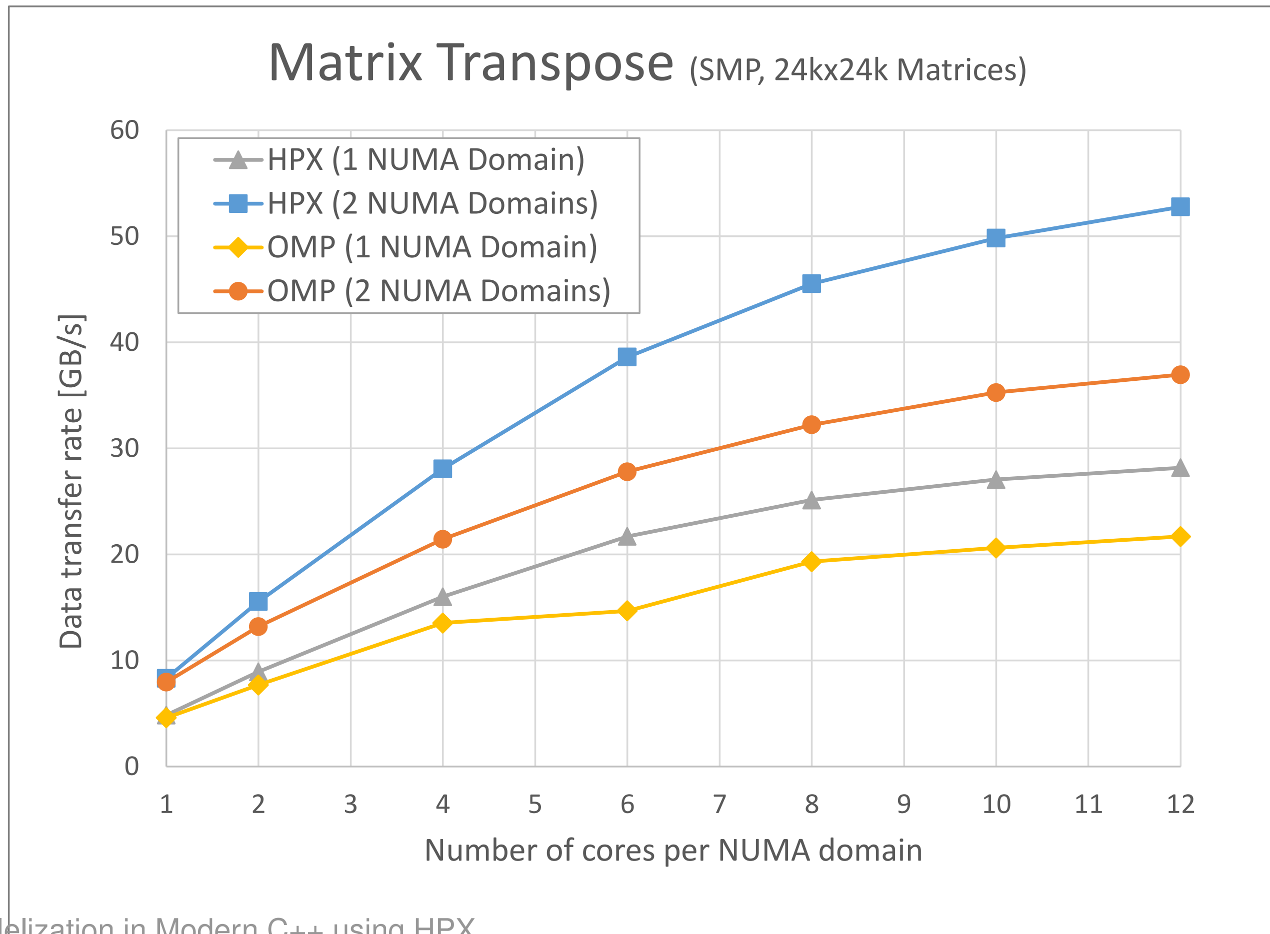
# Example: Matrix Transpose

```
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];
    }
  }
}
```

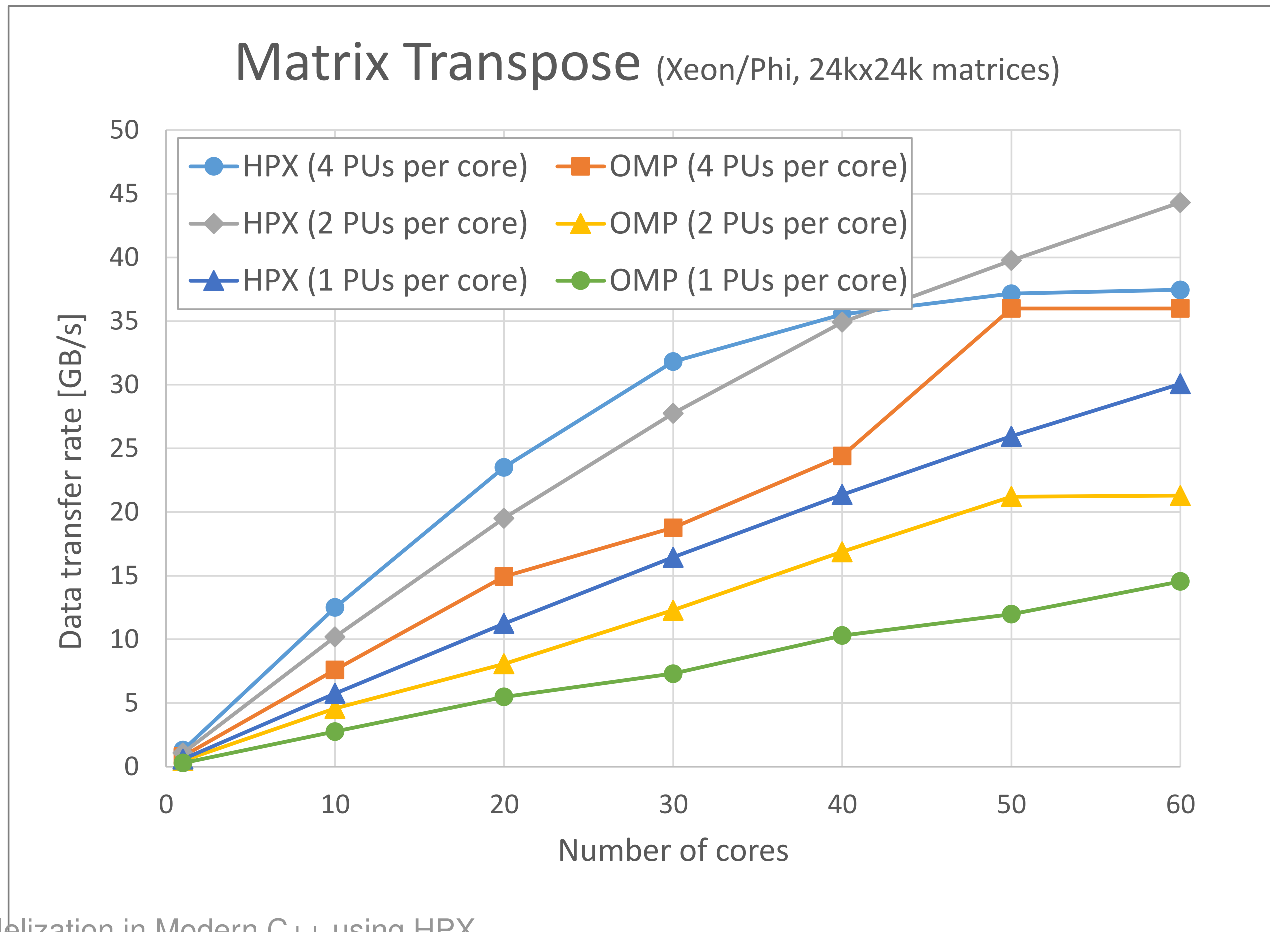
# Example: Matrix Transpose

```
struct block_component
  : hpx::components::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_;
};
```

# Matrix Transpose



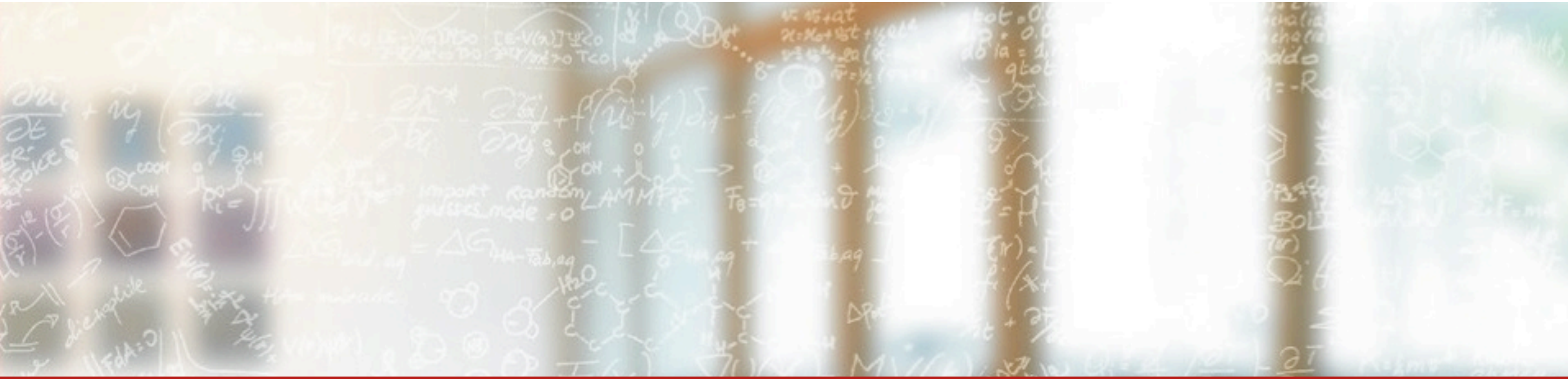
# Matrix Transpose



# HPX vs. TBB

| | HPX | TBB |
|---|---|---|
| Standards conformance | x | |
| Extension to distributed computing | x | |
| Extension to accelerators (GPU etc.) | x | |
| Parallel Algorithms | x | x |
| Dataflow Graphs | dynamic | static |
| User level threading | x | |
| Asynchronous latency hiding | x | |
| Concurrent containers | | x |

ETH zürich



# VTK-m backend using HPX

John Biddiscombe
CSCS / Future Systems
May 2016

# Background : What is VTK-m

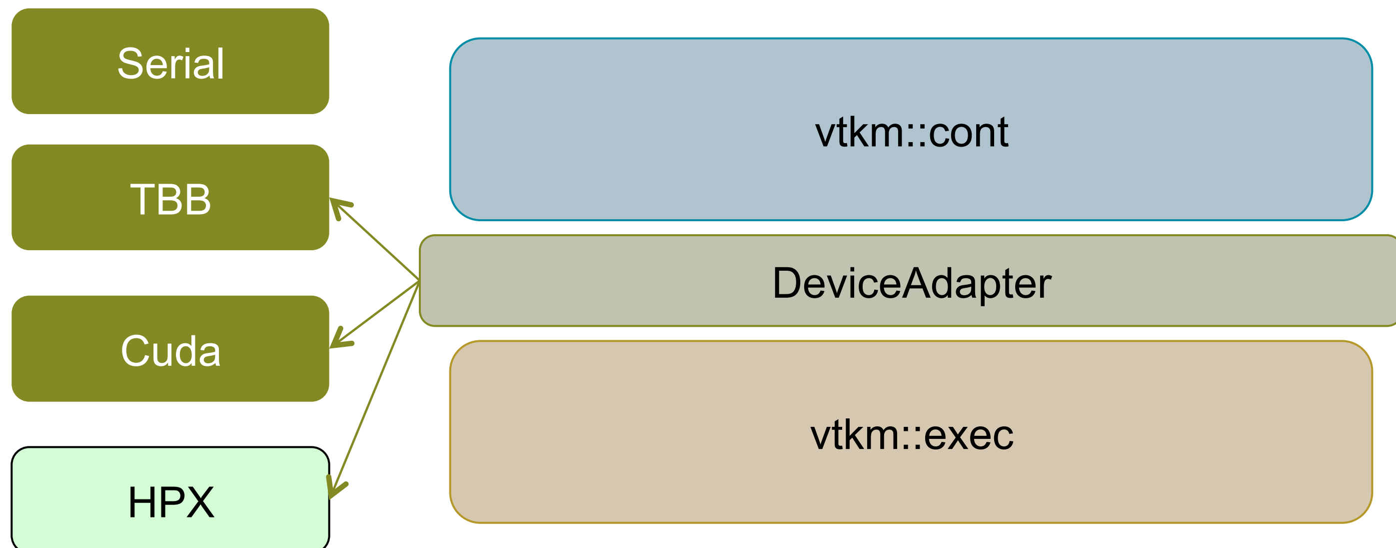
- Next generation VTK replacement
  - Some algorithms in vtk are multi-threaded
  - But not GPU friendly/ready

- VTK-m = Stream processing library for visualization

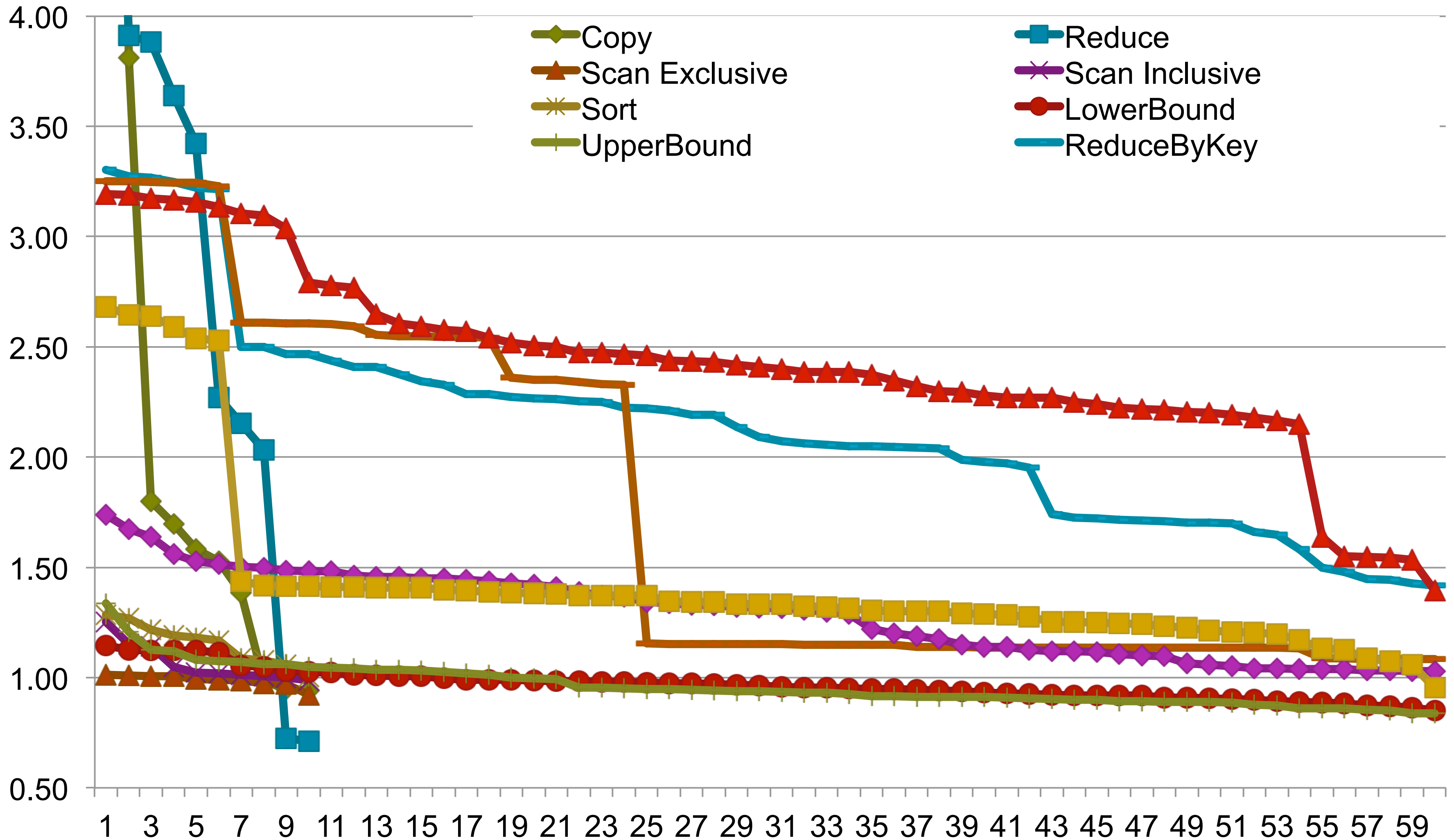
- Control and Execution environments
  - Simple API
  - GPU / many-core / serial

# VTK-m

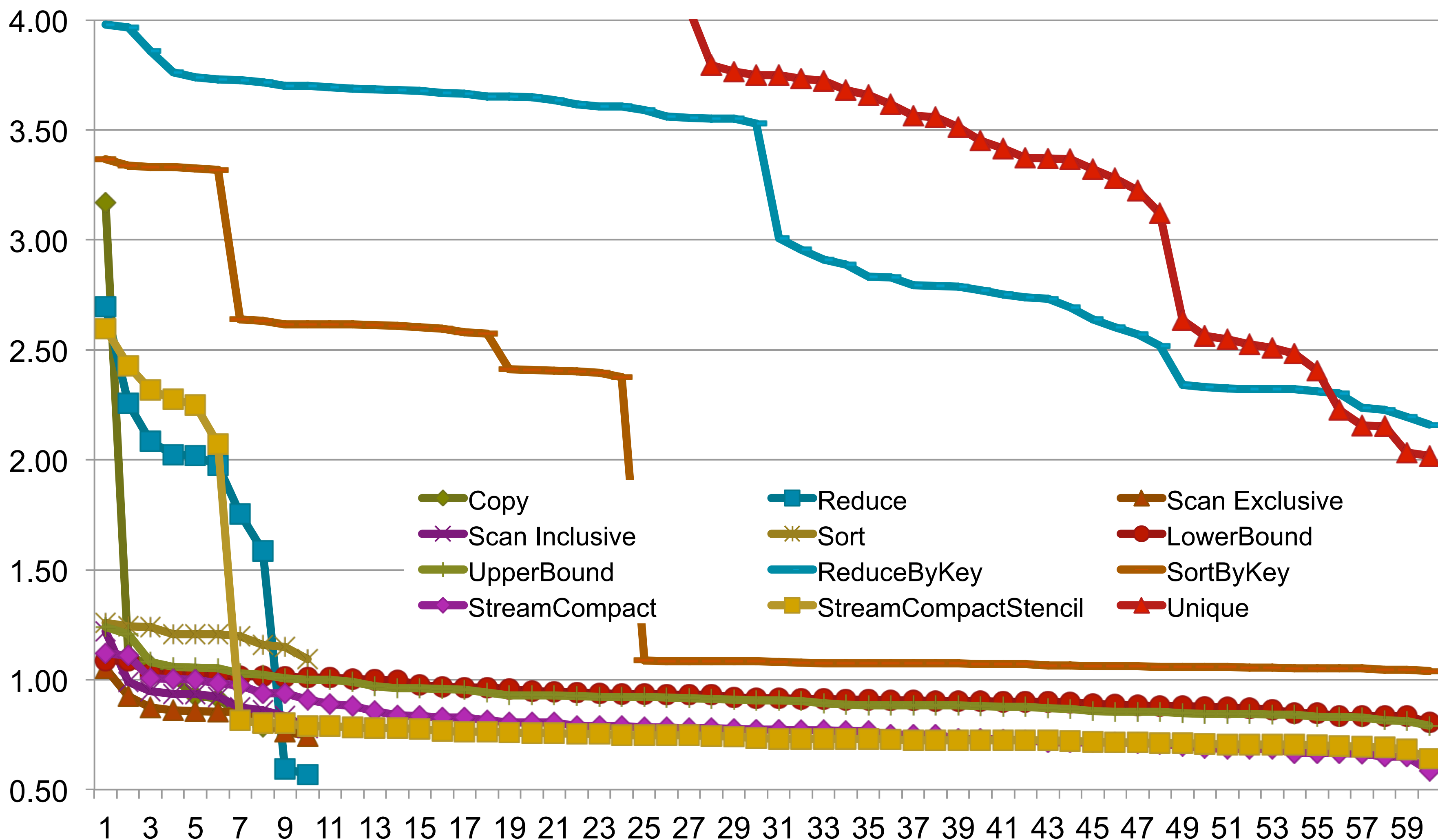
- Control environment : **vtkm::cont**
- Execution environment : **vtkm::exec**
- DeviceAdapter (array handles, memory, offloading)
- DeviceAdapterAlgorithm (API for scheduling work)
- Designed to run worklets (kernels) in the execution environment



# VTK-m benchmark timings
# Speedup of HPX compared to TBB on 10 core broadwell node

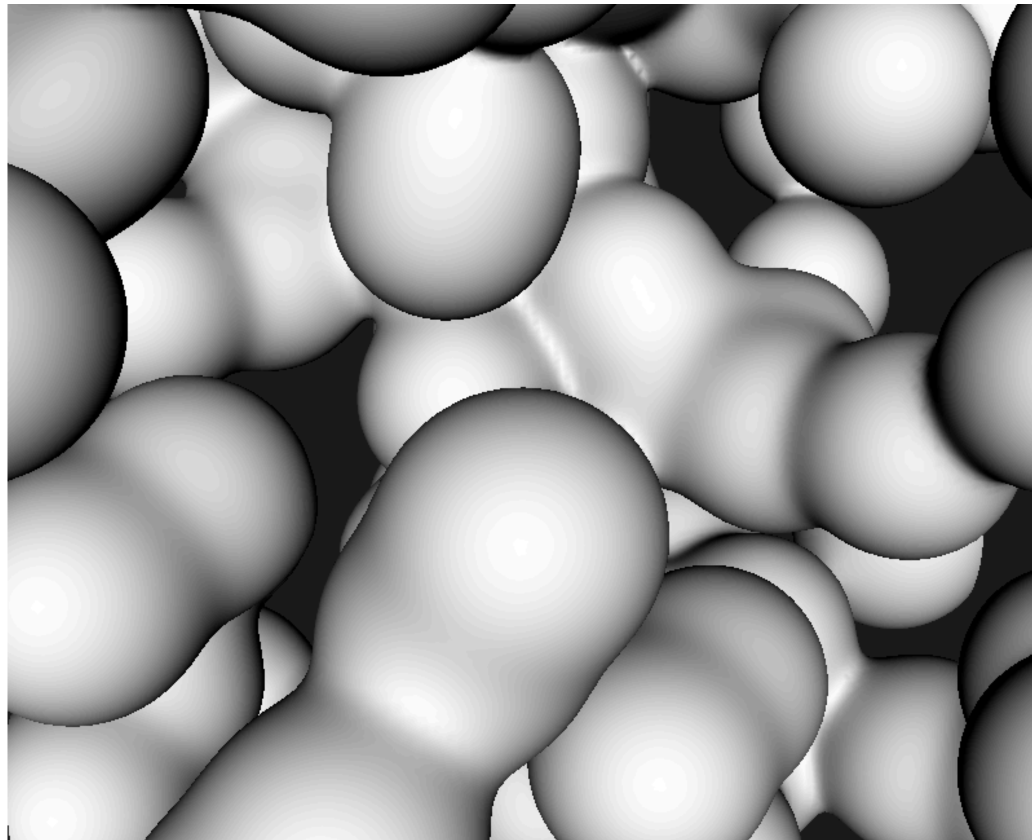


## VTK-m benchmark timings
## Speedup of HPX compared to TBB on 20 core broadwell node



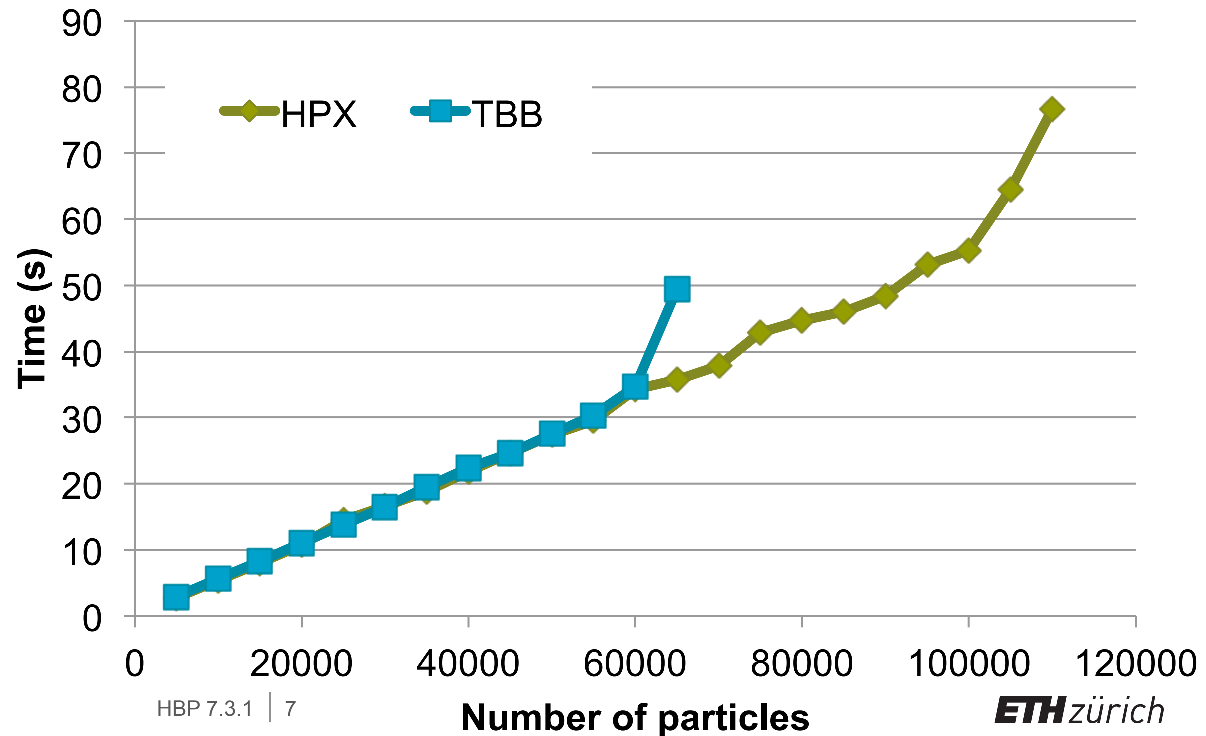
# Kernel splatter – VTK-m implementation

- Atomic version
- Race Free Parallel version



# Splat algorithm (race free) + isosurface

- Uses, Scan, SortByKey, ReduceByKey, Copy, StreamCompact, etc etc
- Scaling better than TBB
- memory



# Conclusions

- Higher-level parallelization abstractions in C++:
  - uniform, versatile, and generic
  - All of this is enabled by use of modern C++ facilities
  - Runtime system (fine-grain, task-based schedulers)
  - Performant, portable implementation
- Asynchronous task based programming to efficiently express parallelism
- Seamless extensions for distributed computing

# Parallelism is here to stay!

- Massive Parallel Hardware is already part of our daily lives!
- Parallelism is observable everywhere:
  - ⇒ IoT: Massive amount devices existing in parallel
  - ⇒ Embedded: Meet massively parallel energy-aware systems (Embedded GPUs, Epiphany, DSPs, FPGAs)
  - ⇒ Automotive: Massive amount of parallel sensor data to process
- We all need solutions on how to deal with this, efficiently and pragmatically

# More Information

- https://github.com/STEllAR-GROUP/hpx
- http://stellar-group.org
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers
- https://isocpp.org/std/the-standard
- hpx-users@stellar.cct.lsu.edu
- #STE||AR @ irc.freenode.org

Collaborations:

- FET-HPC (H2020): AllScale (https://allscale.eu)
- NSF: STORM (http://storm.stellar-group.org)
- DOE: Part of X-Stack