Distributed Parallelism with HPX (1)

Lecture 19

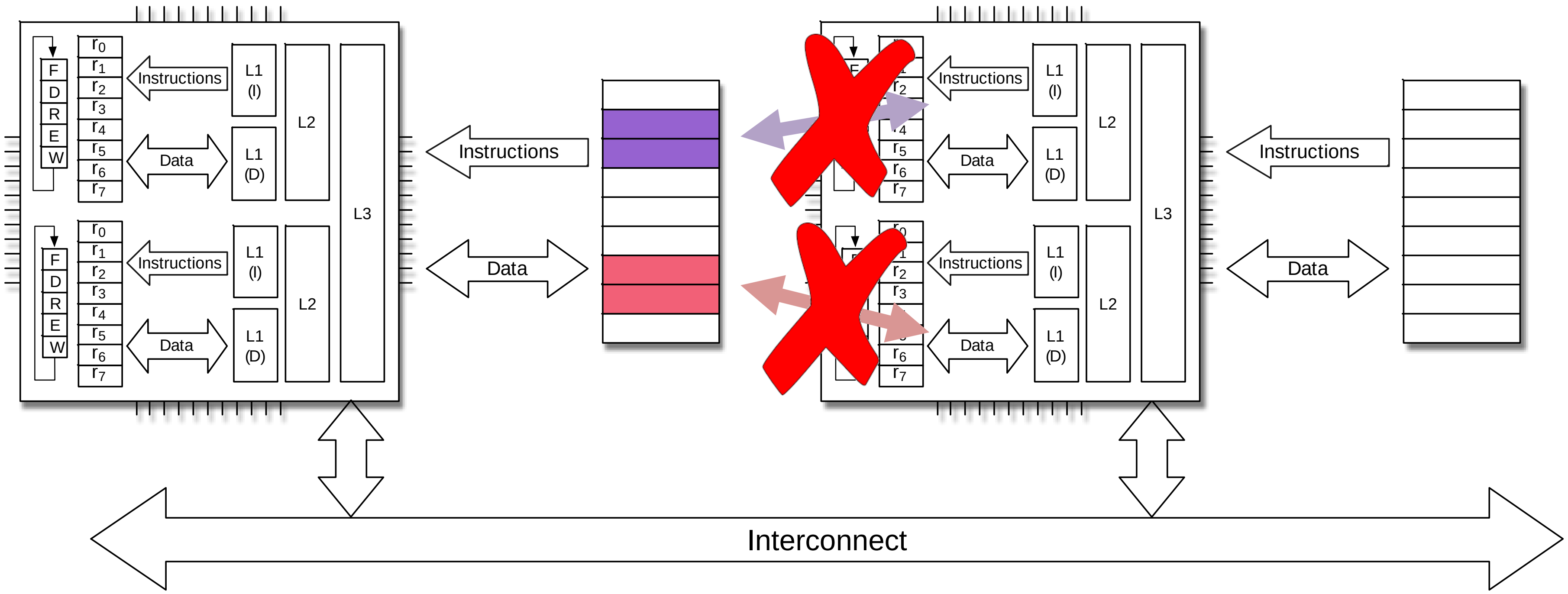
Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4400/

Overview

- SPMD / CSP recap
- A simple mental model
- Basic HPX
- Four Function HPX Point to Point Version
- Four Function HPX Collective Version
- Laplace’s equation on a regular grid

Distributed Memory



Distributed Memory

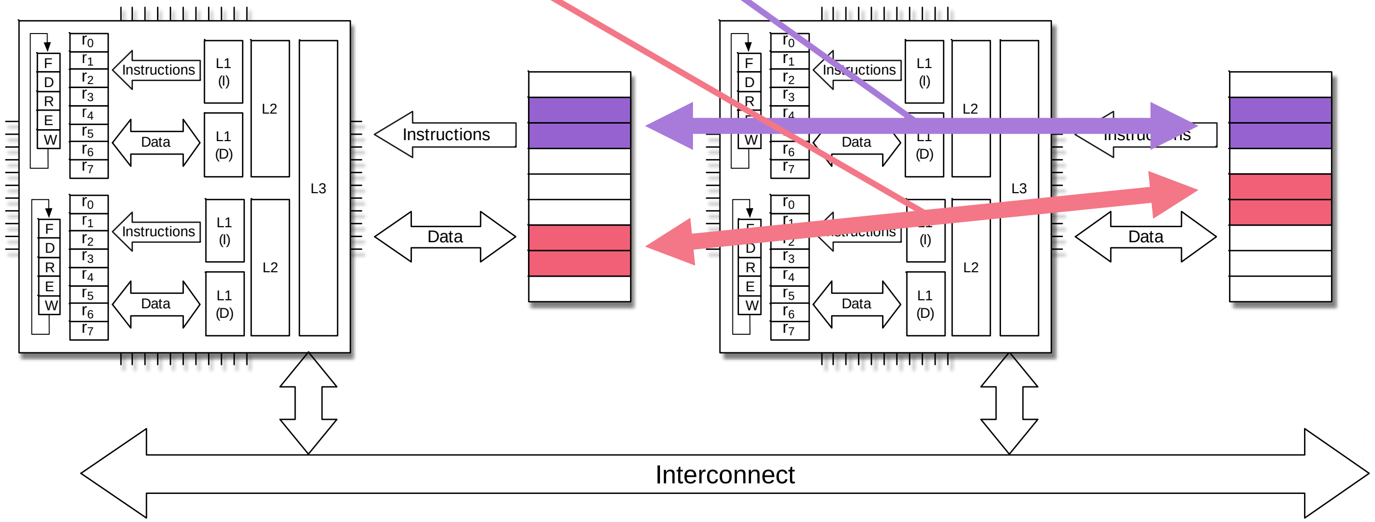
- This begs new questions:
  - Should all nodes do exactly the same thing?
  - Will there be speedup if we do?
- As we add more CPUs, we make the problem bigger
- Can we keep all the data on every node if we keep making the problem bigger?
  - Hint: No
- But. Do we need all the data on every node?
  - Hint: No
- What do we keep? What do we not keep?
  - Every node has some of data, however, the union of all should be the whole problem
  - “Collectively exhaustive”

Distributed Memory

- What about the program?
  - Does it grow with problem size?
  - Hint: No
- We probably need all of it everywhere anyways

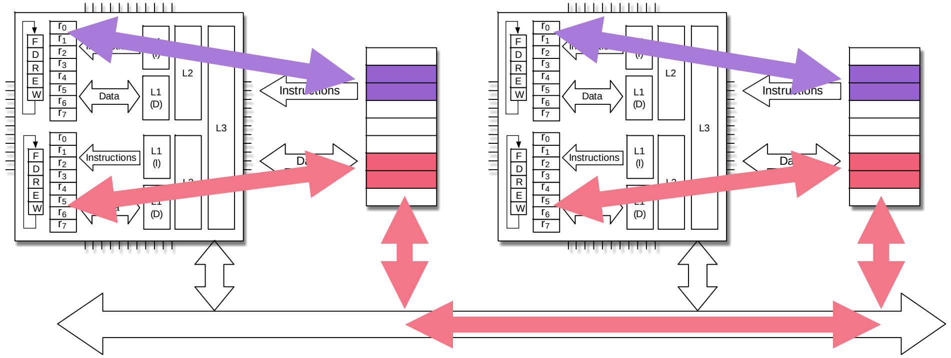
SPMD ('SpimDee')

- Single program
  - This is the same on all nodes
- Multiple Data
  - This is not the same on all nodes, but collectively exhaustive



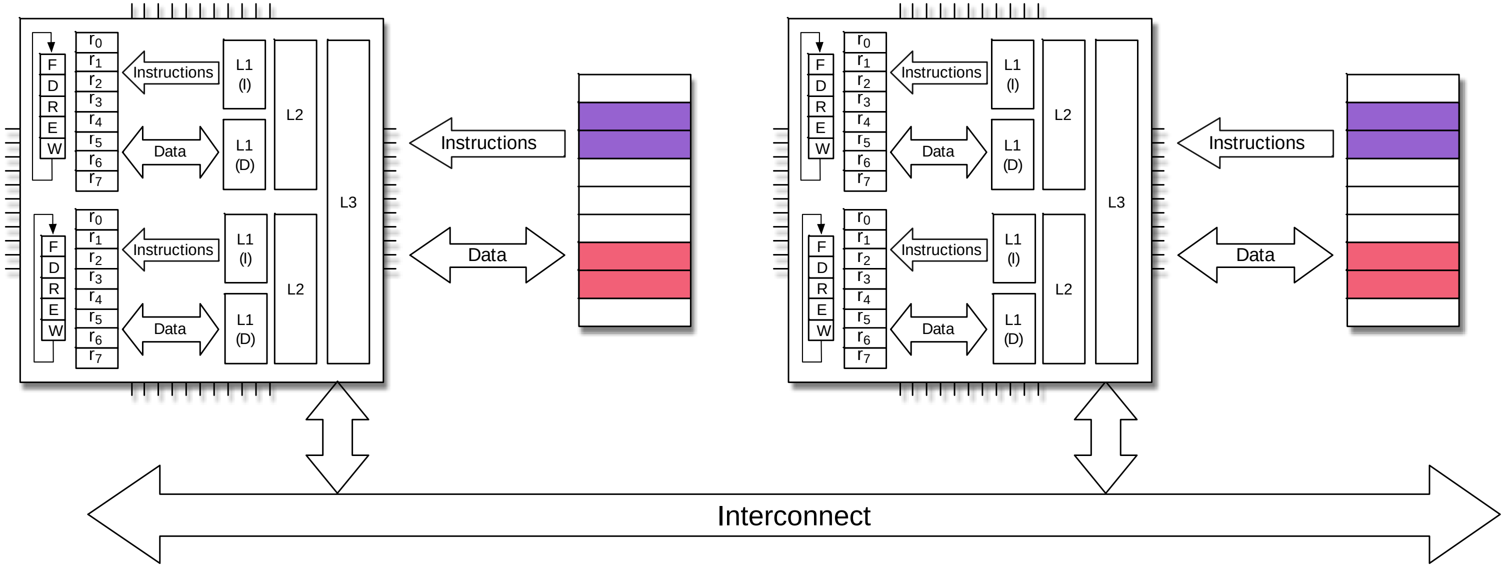
Communicating Sequential Processes (CSP)

- Every node runs a sequential process (nowadays locally parallelized)
  - All of the code is replicated
- Data is distributed using resource allocation mechanisms
  - However, data dependencies are probably not disjoint
  - Data has cross-node dependencies
  - Data may be needed by another node, but can't be accessed directly
  - Data are partitioned
  - The union of the partitions should be the whole problem



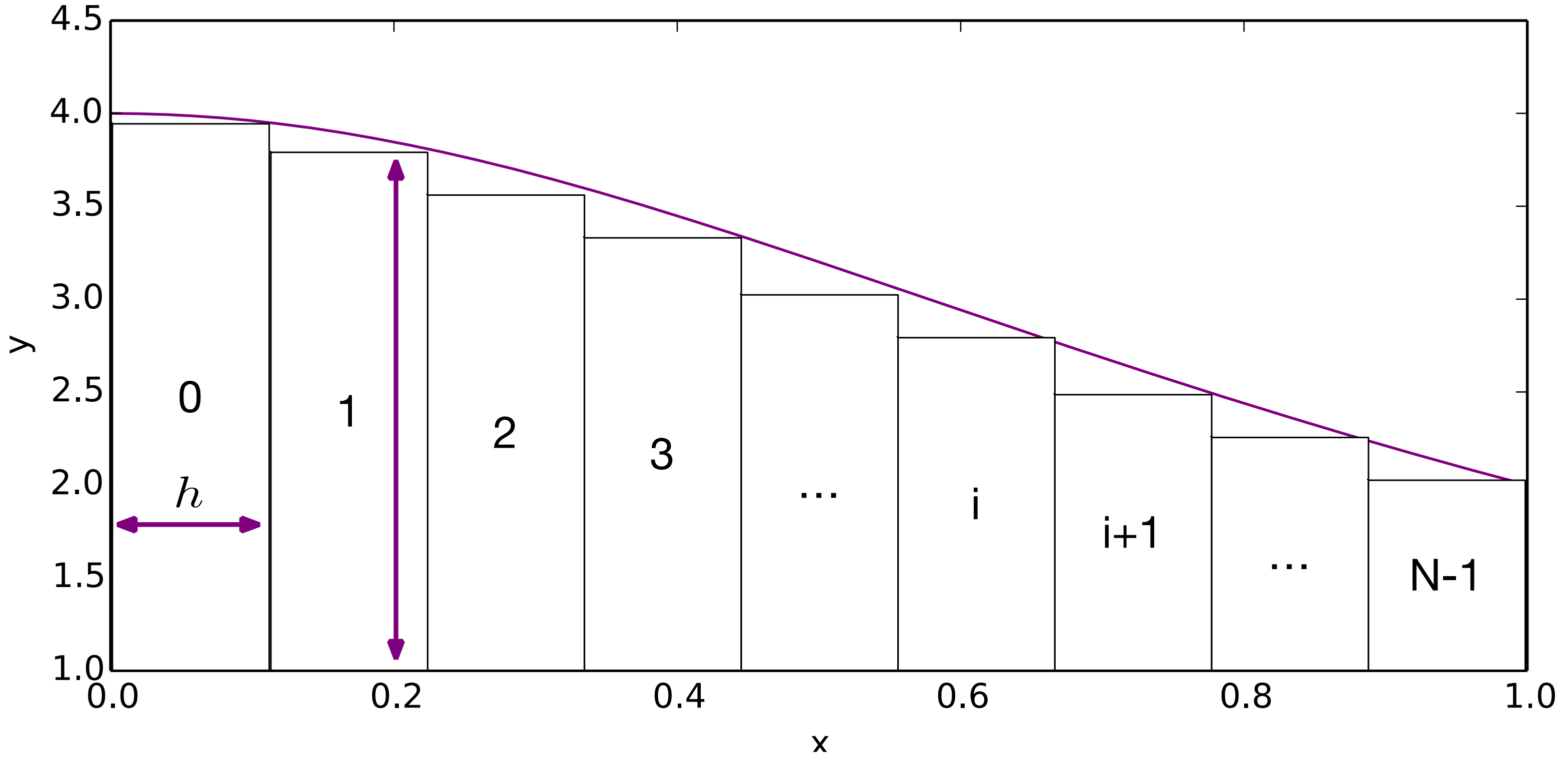
Distributed Memory

- Every process:
  - Independent memory space
  - Code is replicated
- Data are partitioned
  - The union of the partitions should be the whole problem



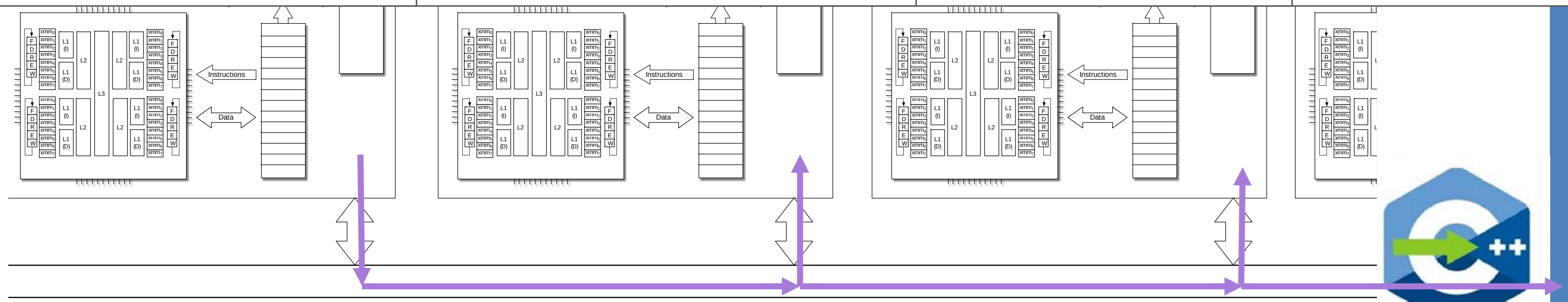
Numerical Integration (Sequential)

```
double h = 1.0 / N;
double pi = 0;
for (int i = 0; i != N; ++i)
    pi += h * 4.0 / (1 + sqr(i * h));
```



Distinguished Replicated Processes

```
int main(int argc, char* argv[])
{
    long N = 1'000'000'000;
    std::uint32_t num_localities =
        hpx::get_num_localities(hpx::launch::sync);
    std::uint32_t locality = hpx::get_locality_id();

    if (locality == 0 && argc > 1) N = stol(argv[1]);

    std::size_t blocksize = N / num_localities;
    std::size_t begin = blocksize * locality_id;
    std::size_t end = blocksize * (locality_id + 1);

    double h = 1.0 / N;
    double pi = 0.0;
    for (long i = begin; i != end; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    if (locality == 0) println("pi: {}", pi);
}
```



HPX Collective Operations

- We assume that process with `locality_id == 0` receives `N` from the command line
- This process sends `N` to all others
  - This is an operation that is called `broadcast`

```
hpx::collectives::broadcast(
    hpx::collectives::get_world_communicator(), N);
```

- Every process calculates its own `begin` and `end` based on its `locality_id`
- Every process now computes part of the overall solution
- So every process needs to provide its partial result, all of which need to be consolidated
  - This is an operation that is called `reduce`

```
hpx::collectives::reduce(
    hpx::collectives::get_world_communicator(),
    pi, std::plus{});
```

‘DropDee’ using HPX

```
int main(int argc, char* argv[]) {
    long N = 1'000'000'000;
    std::uint32_t num_localities =
        hpx::get_num_localities(hpx::launch::sync);
    std::uint32_t locality = hpx::get_locality_id();

    if (locality == 0 && argc > 1) N = stol(argv[1]);

    hpx::collectives::broadcast(
        hpx::collectives::get_world_communicator(), N)

    std::size_t blocksize = N / num_localities;
    std::size_t begin = blocksize * locality_id,
    std::size_t begin = end = blocksize * (locality_id + 1);
    double h = 1.0 / N, pi = 0.0;
    for (long i = begin; i != end; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    hpx::collectives::reduce(
        hpx::collectives::get_world_communicator(), pi);

    if (locality == 0) println("pi: {}", pi);
}
```

- Get our locality id and number of other nodes
- Locality 0 gets N, shares N
- Everyone computes their own partial result
- Locality 0 collects all partial results, adds them, and prints

This pattern is ubiquitous

Communicating Sequential Processes

- Every process can only read/write its own memory
  - One process sends data
  - Other processes receive data
- Communicating Sequential Processes operate purely local
  - Every process talks to its own memory and to its own networking interface
  - It is as if we operated in shared memory, but it is purely local
- When we run a "parallel program" we aren't running a parallel program (no 'distributed parallelism')
  - We are running multiple copies of a sequential program (nowadays possibly locally parallelized)
  - All copies execute exactly the same code (not in lock step)

Hello HPX World

```cpp
#include <cstdint>
#include <print>

#include <hpx/hpx.hpp>          // make all of HPX available
#include <hpx/hpx_main.hpp>     // initialize HPX before main

int main()
{
    std::uint32_t num_localities = hpx::get_num_localities(hpx::launch::sync);
    std::uint32_t locality_id = hpx::get_locality_id();

    std::println("Hello world!");
    std::println("I am {} of {}", locality_id, num_localities);

    return 0;
}
```

Minimal HPX CMake Support

```
project(spring2025-csc4700-distributed)

set(CMAKE_CXX_STANDARD 23)      # possible values: 17, 20, 23

find_package(HPX REQUIRED)      # make HPX available

# create an executable hello_world from hello_world.cpp and link it to HPX
add_executable(hello_world code/hello_world.cpp)
target_link_libraries(hello_world PRIVATE HPX::hpx HPX::wrap_main)

# make sure main() is run on all localities
target_compile_definitions(hello_world PRIVATE HPX_HAVE_RUN_MAIN_EVERYWHERE)
```

Important!

Running an HPX Application

- Launch four copies of hello_world:

```
> hpxrun.py --localities=4 ./hello_world
```

- Output (printed from all processes since this was local on my laptop):

```
Hello world! I am 0 of 4
Hello world! I am 1 of 4
Hello world! I am 2 of 4
Hello world! I am 3 of 4
```

- `hpxrun.py` has more options, use `--help` for a list

Sending and Receiving

- First question: What is it we are we sending?
  - "N" only has meaning in source code
  - We are sending the value of "N", i.e. a stream of bytes
  - The bits in the memory location
  - And we need to be able to id other processes
- Second question: Where are we sending the bits?
  - How do we say "N@other_process"?
  - "N" only has meaning in source code
  - And its location only makes sense locally
  - But sender and receiver can agree on an alias

Aside: Serialization

- HPX supports serialization of variables of arbitrary C++ types
  - Serialization: convert C++ object into stream of bytes
  - Deserialization: convert a stream of bytes back to a C++ object of a known type
  - HPX sends the byte-stream generated from the arguments of the collective operations over the network (return values also)
- All built-in types are directly supported
  - `int`, `short`, `long`, `double`, `float`, etc.
- All C++ standard containers and utility types are directly supported
  - E.g., `std::vector<T>` as long as `T` is serializable
  - `std::variant`, `std::tuple`, `std::optional`, etc.
- Many of HPX' own types are directly supported
  - E.g., `hpx::future<T>` (as long as `T` is serializable)

Aside: Serialization

- Serializing (and de-serializing) custom types

```
struct point {

    double coord_x, coord_y;
    std::vector<double> data;

    template <typename Archive>
    void serialize(Archive& ar, unsigned version)
    {
        ar & coord_x & coord_y & data;
    }
};
```

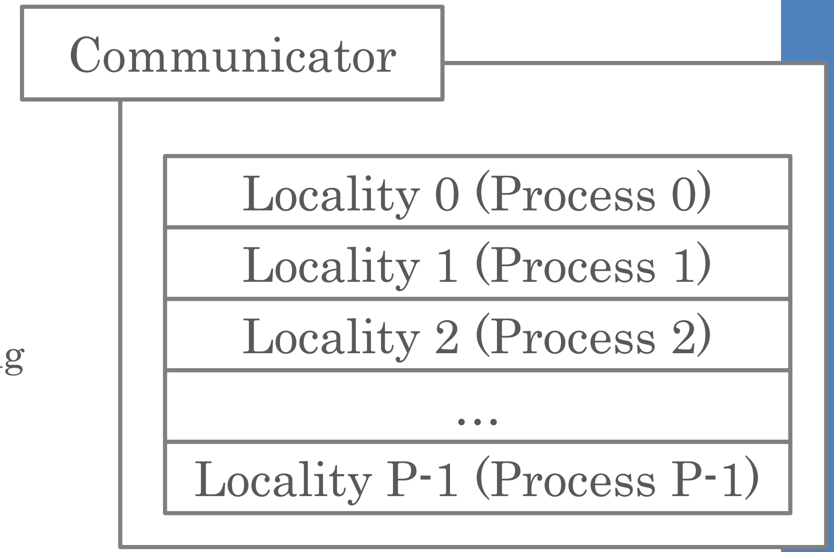
- This will enable sending i.e., a `std::vector<point>` to other localities

A Simple Mental Model

- All HPX communication takes place in the context of a source locality and a target (destination) locality
- Each locality is exposed by a sequential process
- HPX exposes two types of communicators
  - Channel-based communicators (for peer-to-peer send/receives)
  - Communicators for collective operations
- HPX exposes predefined communicators that refer to all localities
  - `hpx::collectives::get_world_communicator()`: communicator for collective operations across all localities
  - `hpx::collectives::get_world_channel_communicator()`: channel communicator for peer-to-peer operations between all localities

A Simple Mental Model

- A channel-based communicator enables the peer-to-peer communication between all localities that are part of it
- A communicator for collective operations (in general) groups several localities
- Only localities in the communicator can use it
- All processes in the communicator see an identical list of localities
  - Behavior is as if this list was global and shared
- We use the index of a locality in the communicator to identify other localities
  - You can think of the index as the sequence number of participating endpoints
- The size of a communicator is defined as the number of endpoints it represents
- Localities/endpoints can query for the size and for their own index in the communicator



A Simple Mental Model

- Query properties of a communicator:

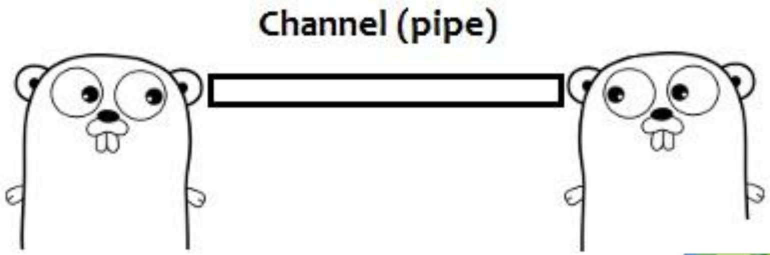
```
void communicator_properties()
{
    auto comm = hpx::collectives::get_world_communicator();
    auto [num_localities, this_locality] = comm.get_info();

    std::println("The global communicator has: ");
    std::println("  Number of connected localities: {}", num_localities);
    std::println("  Sequence number of this locality: {}", this_locality);
}

void channel_communicator_properties()
{
    auto comm = hpx::collectives::get_world_channel_communicator();
    auto [num_localities, this_locality] = comm.get_info();

    std::println("The global channel communicator has: ");
    std::println("  Number of connected localities: {}", num_localities);
    std::println("  Sequence number of this locality: {}", this_locality);
}
```

A Simple Mental Model

```
int main(int argc, char* argv[])
{
    auto comm = hpx::collectives::get_world_communicator();
    long N = 1'000'000'000;

    auto [num_localities, locality] = comm.get_info();
    if (locality == 0 && argc > 1) N = std::stol(argv[1]);

    hpx::collectives::broadcast(comm, N);

    std::size_t blocksize = N / num_localities;
    std::size_t begin = blocksize * locality_id;
    std::size_t begin = end = blocksize * (locality_id + 1);
    double h = 1.0 / N, pi = 0.0;
    for (long i = begin; i != end; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    hpx::collectives::reduce(comm, pi);
    if (locality == 0) std::println("pi: {}", pi);
}
```

(A)synchronous Channels

- High level abstraction of communication operations
  - Perfect for asynchronous boundary exchange
  - Allows to send/receive arbitrary data types, sender and receiver have to agree on the type, though
- Modelled after Go-channels
  - Simple (uni-directional) pipeline between two endpoints
  - Channel can hold one element at any time
- Create on one locality, refer to it from another locality
  - Conceptually similar to bidirectional P2P (MPI) communicators
- A `channel_communicator` is a collection of channels, one for each combination of endpoints
- Asynchronous in nature
  - `get()` and `set()` return futures
  - But there exist synchronous variations



hpx::collectives::set

```
template <typename T>
void set(hpx::launch::sync_policy, channel_communicator comm,
    that_site_arg site, T&& value, tag_arg tag);

template <typename T>
hpx::future<void> set(channel_communicator comm,
    that_site_arg site, T&& value, tag_arg tag);
```

- Here:
  - `sync_policy`: special predefined type that instructs to execute the `set` operation synchronously (wait for operation to finish)
  - `comm`: the channel-communicator instance used for this message
  - `site`: the `locality_id` of the recipient
  - `value`: the value to send
  - `tag`: the message tag (identifies a particular communication operation, default: `tag_arg()`)

- The sender is implicit (the locality that called this function)

hpx::collectives::get

```
template <typename T>
T get(hpx::launch::sync_policy, channel_communicator comm,
    that_site_arg site, tag_arg tag);

template <typename T>
hpx::future<T> get(channel_communicator comm,
    that_site_arg site, tag_arg tag);
```

- Here:
  - `sync_policy`: special predefined type that instructs to execute the `get` operation synchronously (wait for operation to finish)
  - `comm`: the channel-communicator instance used for this message
  - `site`: the `locality_id` of the sender
  - `tag`: the message tag (identifies a particular communication operation, default: `tag_arg()`)
- Function returns the received data
- The receiver is implicit (the locality that called this function)

set and get

- In order for a data item to be delivered
  - Both, the sender must call `set` and the receiver must call `get`
  - With a matching `tag` and matching sender and receiver `locality_id`'s
    - (i.e. the receiver must specify the sender, while the sender specifies the receiver)
- The data type passed to `set` must match the data type explicitly specified for `get<T>`
  - This is because the message that is being transferred is essentially a bit-stream
  - This bit-stream represents the value of the variable that is being sent/received
  - In order to properly interpret this bit-stream the receiver must 'know' what type of the variable the sender used to generate the bit-stream from

set and get

- Both functions are asynchronous by default
  - `set` returns a future that becomes ready once the message was accepted by the channel
  - `get` returns a future that holds the received value that becomes ready once the data has been received

- If invoked with `hpx::launch::sync` as the first argument, both functions will be synchronous
  - `hpx::launch::sync` is a predefined instance of a `hpx::launch::sync_policy`
  - Return only after the requested operation has finished

- BTW: Using `hpx::launch::sync` as the first argument to a asynchronous function is generally possible for many of HPX' APIs
  - Will turn this function into the synchronous equivalent

Ping Pong

```
int main(int argc, char* argv[])
{
    using namespace hpx::collectives;

    auto comm = get_world_channel_communicator();
    int sent = 42, received = 0;

    set(hpx::launch::sync, comm, that_site_arg(1), sent, tag_arg(123));
    received = get<int>(hpx::launch::sync, comm, that_site_arg(0), tag_arg(123));

    set(hpx::launch::sync, comm, that_site_arg(0), received, tag_arg(123));
    sent = get<int>(hpx::launch::sync, comm, that_site_arg(1), tag_arg(123));

    std::println("Received: {}", sent);
}
```

Ping Pong

- What happened?

```
$ hpxrun.py --localities=2 ./pingpong
Received: 42
... ^C ... Process terminated
```

Ping Pong

```
int main(int argc, char* argv[])
{
    using namespace hpx::collectives;

    auto comm = get_world_channel_communicator();
    int sent = 42, received = 0;

    set(hpx::launch::sync,
        comm, that_site_arg(1), sent, tag_arg(123));
    received = get<int>(hpx::launch::sync,
        comm, that_site_arg(0), tag_arg(123));

    set(hpx::launch::sync,
        comm, that_site_arg(0), received, tag_arg(123));
    sent = get<int>(hpx::launch::sync,
        comm, that_site_arg(1), tag_arg(123));

    std::println("Received: {}", sent);
}
```

- All processes run this same program
- Both processes send this
- And try to receive
- Process with `locality_id == 0` will never receive anything

Ping Pong 2.0

```
int main(int argc, char* argv[]) {
    std::uint32_t locality_id = hpx::get_locality_id();

    using namespace hpx::collectives;

    auto comm = get_world_channel_communicator();
    int sent = 42, received = 0;

    if (locality_id == 0) {
        set(hpx::launch::sync, comm, that_site_arg(1), sent, tag_arg(123));
        received = get<int>(hpx::launch::sync, comm, that_site_arg(1), tag_arg(123));
        std::println("Received: {}", received);
    }
    if (locality_id == 1) {
        received = get<int>(hpx::launch::sync, comm, that_site_arg(0), tag_arg(123));
        set(hpx::launch::sync, comm, that_site_arg(0), received, tag_arg(123));
    }
}
```

Ping Pong 2.0

```
int main(int argc, char* argv[]) {
    std::uint32_t locality_id = hpx::get_locality_id();

    using namespace hpx::collectives;

    auto comm = get_world_channel_communicator();
    int sent = 42, received = 0;

    if (locality_id == 0) {
        set(hpx::launch::sync,
            comm, that_site_arg(1), sent, tag_arg(123));
        received = get<int>(hpx::launch::sync,
             comm, that_site_arg(1), tag_arg(123));
        std::println("Received: {}", received);
    }
    if (locality_id == 1) {
        received = get<int>(hpx::launch::sync,
            comm, that_site_arg(0), tag_arg(123));
        set(hpx::launch::sync,
            comm, that_site_arg(0), received, tag_arg(123));
    }
}
```

- Only process 0 sends this
- Only process 0 receives this
- Only process 1 receives this
- Only process 1 sends this

Ping Pong 2.0

```
$ hpxrun.py --localities=2 ./pingpong2
Received: 42
$
```

```
$ hpxrun.py --localities=8 ./pingpong2
Received: 42
$
```

Four Function HPX (Point to Point)

```
int main()
{
    using namespace hpx::collectives;

    auto comm = get_world_channel_communicator();
    auto [num_localities, locality_id] = comm.get_info();

    if (locality_id == 0) {
        int received = get<int>(hpx::launch::sync, comm, that_site_arg(1));
        std::println("locality_id(0): received: {}", received);
    }
    else {
        set(comm, that_site_arg(0), 42);
    }
}
```

The Other Four HPX Functions

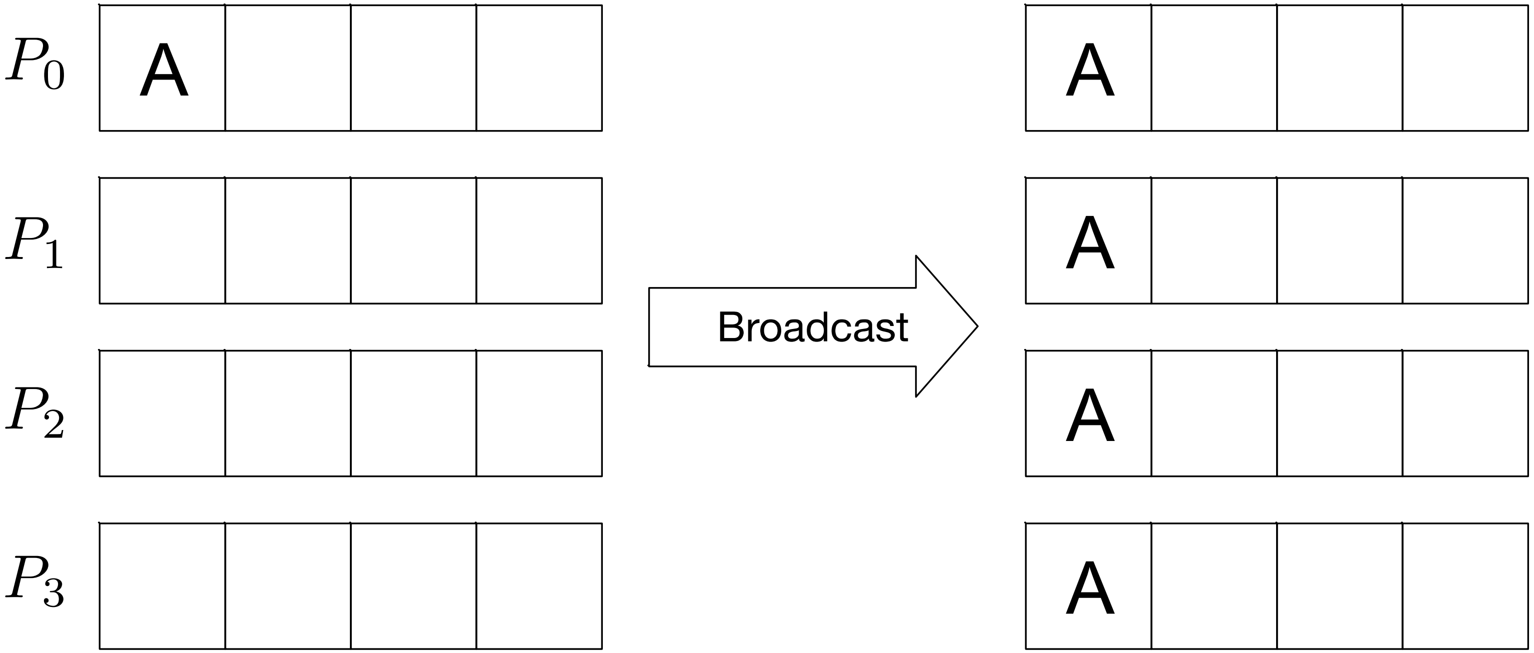
```
int main(int argc, char* argv[]) {
    auto comm = hpx::collectives::get_world_communicator();
    long N = 1'000'000'000;

    auto [num_localities, locality] = comm.get_info();

    if (locality == 0 && argc > 1) N = std::stol(argv[1]);
    hpx::collectives::broadcast(
        hpx::collectives::get_world_communicator(), N)

    std::size_t blocksize = N / num_localities;
    std::size_t begin = blocksize * locality_id,
    std::size_t begin = end = blocksize * (locality_id + 1);
    double h = 1.0 / N, pi = 0.0;
    for (long i = begin; i != end; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    hpx::collectives::reduce(
        hpx::collectives::get_world_communicator(), pi);

    if (locality == 0) println("pi: {}", pi);
}
```

- Get our locality id and number of other nodes
- locality 0 gets N, shares N
- Everyone computes their own partial
- locality 0 collects all partials, adds them, and prints

This pattern is ubiquitous

hpx::collectives::broadcast

```
template <typename T>
void broadcast(communicator comm, T& value,
    this_site_arg this_site = this_site_arg(),
    generation_arg generation = generation_arg());
```



hpx::collectives::broadcast

- The value is the 'send buffer' for root (locality 0), but is the 'receive buffer' for all others
- After broadcast returns, all will all have a copy of what root had
- Note (true for all collective operations):
  - All localities connected to the given communicator must call the function
  - Otherwise none of the localities will finish the operation

hpx::collectives::broadcast

```
template <typename T>
void broadcast(communicator comm, T& value,
    this_site_arg this_site = this_site_arg(),
    generation_arg generation = generation_arg());
```

- Here:
  - `comm`: the communicator instance used for this message
  - `value`:
    - on locality 0 [in]: local data value to use for broadcast,
    - on all localities: [out]: result
  - `this_site`: the local ‘endpoint’ (defaults to this locality)
  - `generation`: a sequence number of the operation (defaults to invocation counter)
- Only `this_site == 0` provides the value to broadcast to all participating localities
- Note: this function is synchronous by default (use `broadcast_to / broadcast_from` for asynchronous operation)

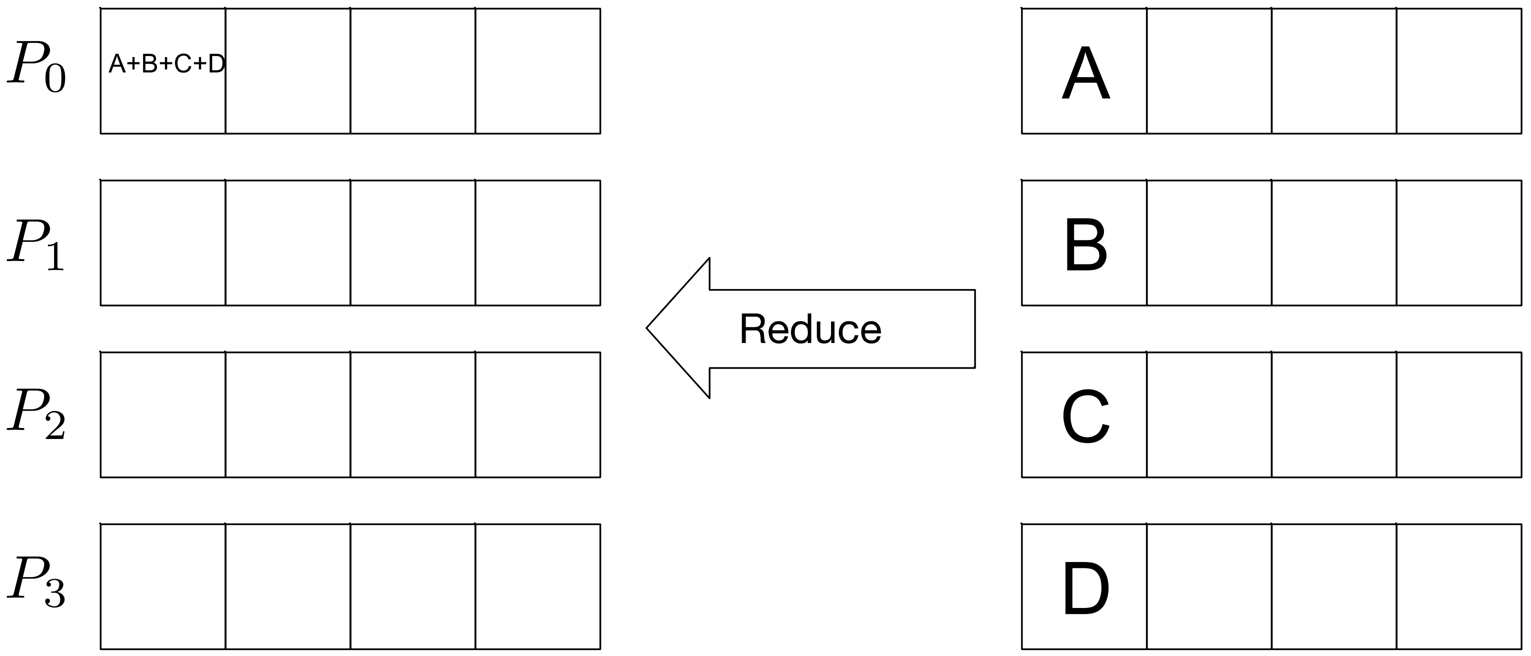
hpx::collectives::broadcast

```
template <typename T>
hpx::future<T> broadcast_to(communicator comm,
    T local_result, this_site_arg this_site, generation_arg generation);

template <typename T>
hpx::future<T> broadcast_from(communicator fid,
    this_site_arg this_site, generation_arg generation);
```

- Here:
  - comm: the communicator instance used for this broadcast operation
  - value:
    - broadcast_to [in]: local data value to broadcast to all other participating sites,
    - broadcast_from: on all localities: function returns received result
  - this_site: the local 'endpoint' (defaults to this locality)
  - generation: a sequence number of the operation (defaults to invocation counter)

hpx::collectives::reduce

```
template <typename T, typename F>
void reduce(communicator comm, T& result, F&& op,
    this_site_arg this_site = this_site_arg(),
    generation_arg generation = generation_arg());
```



hpx::collectives::reduce

- The value is the 'send buffer' for all localities, but is the 'send/receive buffer' for the root (locality 0)
- After reduce returns, locality 0 will have the reduction result of the values supplied by all localities
- Note (true for all collective operations):
  - All localities connected to the given communicator must call the function
  - Otherwise none of the localities will finish the operation

hpx::collectives::reduce

```
template <typename T, typename F>
void reduce(communicator comm, T& result, F&& op,
    this_site_arg this_site = this_site_arg(),
    generation_arg generation = generation_arg());
```

- Here:
  - `comm`: the communicator instance used for this message
  - `result`:
    - [in] local data value to use for reduction
    - on locality 0: [in/out]: result
  - `op`: reduction operator (defaults to `std::plus`)
  - `this_site`: the local ‘endpoint’ (defaults to this locality)
  - `generation`: a sequence number of the operation (defaults to invocation counter)
- Only `this_site == 0` receives the reduced value from all participating localities
- Note: this function is synchronous by default (use `reduce_to` / `reduce_from` for asynchronous operation)

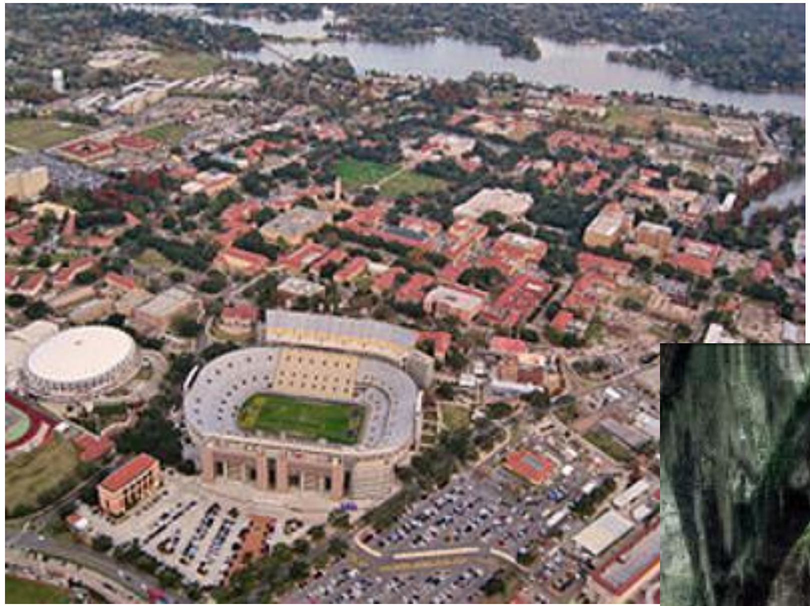
hpx::collectives::reduce

```
template <typename T, typename F>
hpx::future<T> reduce_here(communicator fid, T&& result,
    F&& op, this_site_arg this_site, generation_arg generation);

template <typename T>
hpx::future<void> reduce_there(communicator fid, T&& result,
    this_site_arg this_site, generation_arg generation);
```

- Here:
  - `comm`: the communicator instance used for this message
  - `result`:
    - `reduce_there`, `reduce_here`: [in] local data value to use for reduction
    - `reduce_here`: also returns reduction result
  - `op`: reduction operator (defaults to `std::plus`)
  - `this_site`: the local 'endpoint' (defaults to this locality)
  - `generation`: a sequence number of the operation (defaults to invocation counter)

Conclusions

- DropDee (Distibuished Replicated Processes with Distributed Data) requires data decomposition and communication
  - To make data small enough to fit into one node's memory
- DropDee model can be implemented using four functions of HPX
  - Use peer-to-peer communication (channel-based set/get)
  - Use collective operations (reduce, broadcast, etc.)
- DropDee Processes operate purely local
  - Every process talks to its own memory and to its own networking interface
  - It is as if we operated in shared memory, but it is purely local
- When we run a "parallel program" we aren't running a parallel program
  - We are running multiple copies of a sequential program (possibly locally parallelized)
  - All copies execute exactly the same code (not in lock step)



LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP