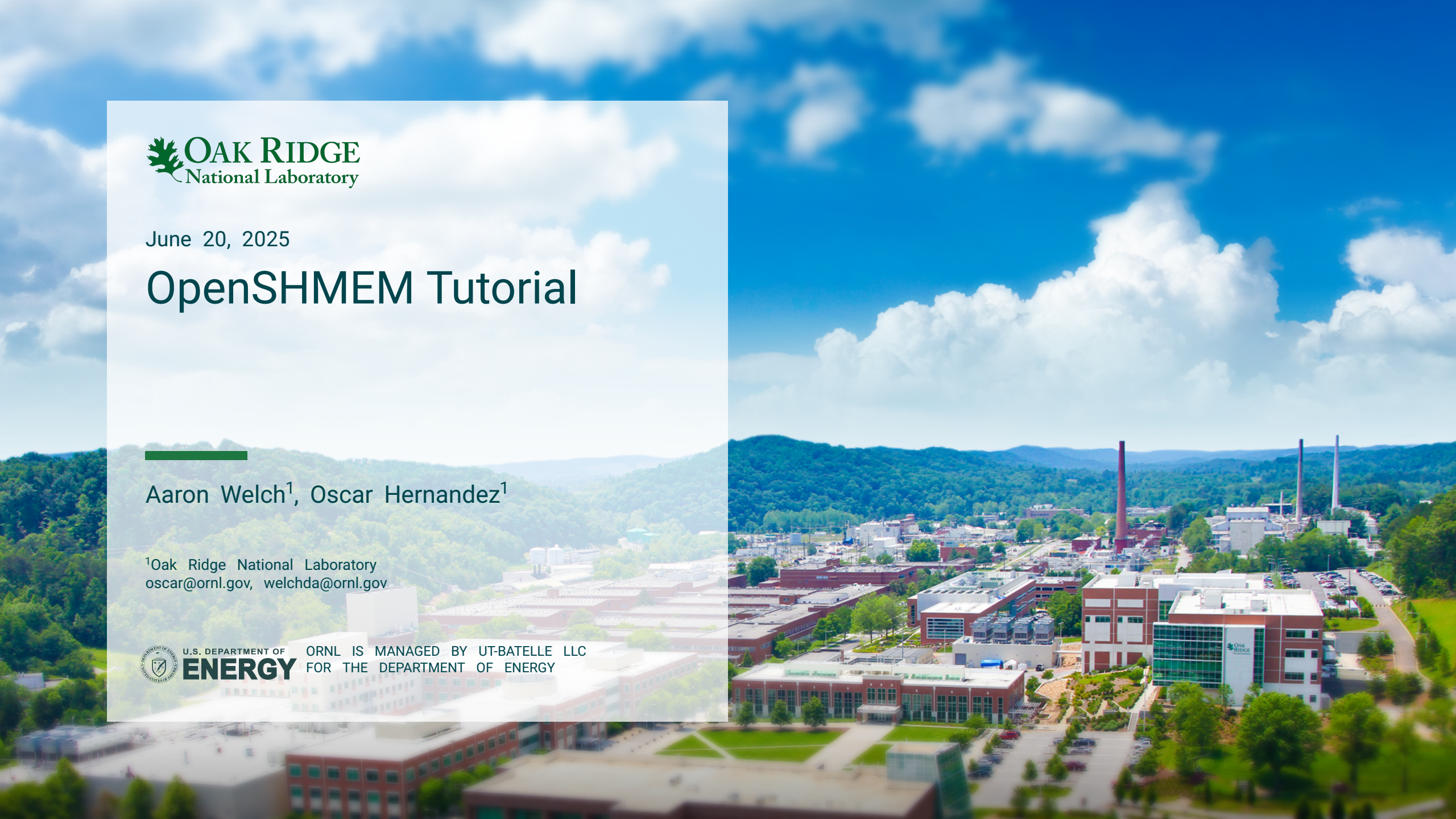


OAK RIDGE National Laboratory

June 20, 2025

OpenSHMEM Tutorial

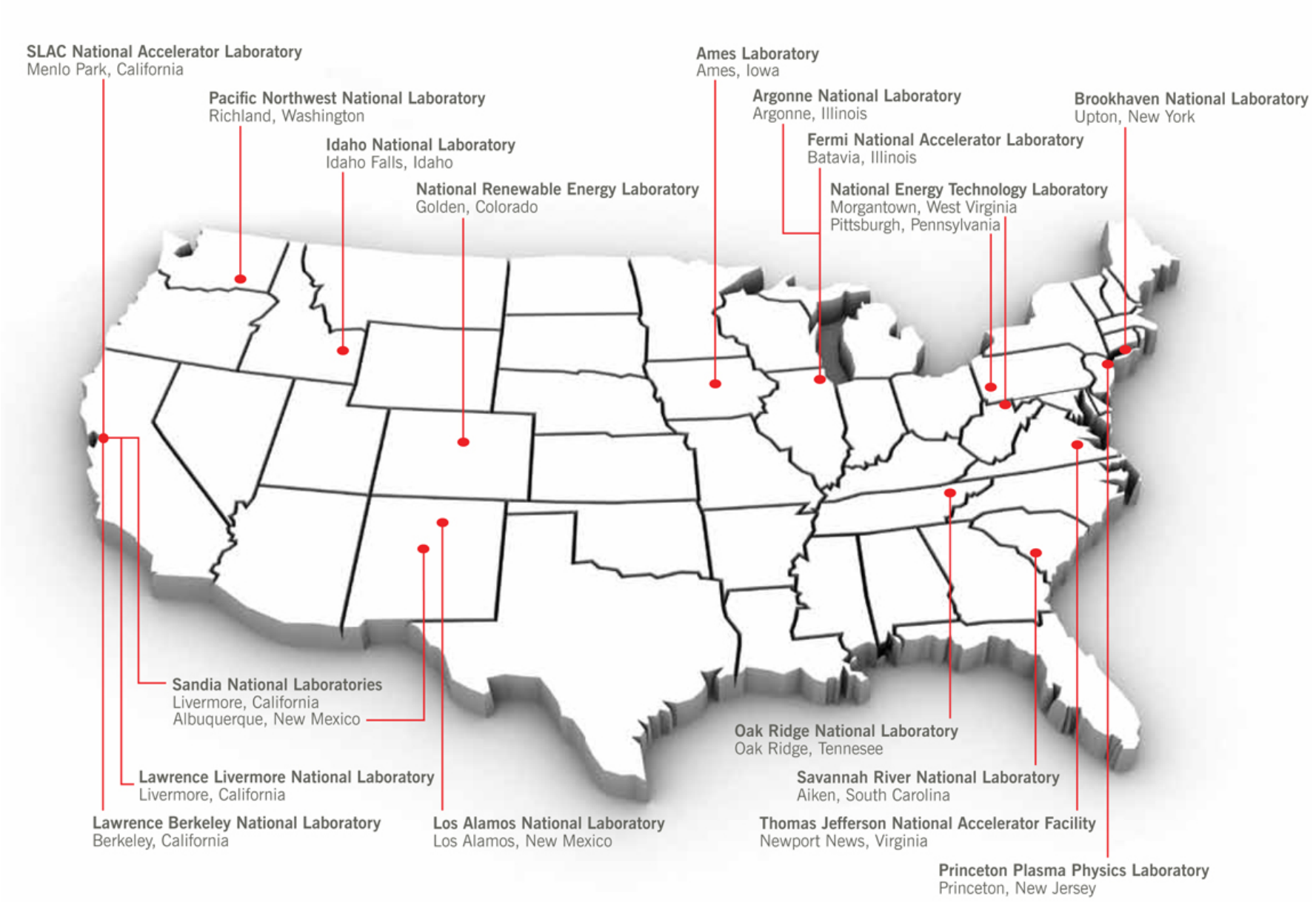
Aaron Welch[1], Oscar Hernandez[1]

[1]Oak Ridge National Laboratory
oscar@ornl.gov, welchda@ornl.gov

U.S. DEPARTMENT OF ENERGY

ORNL IS MANAGED BY UT-BATELLE LLC FOR THE DEPARTMENT OF ENERGY

A Little About ORNL. . .



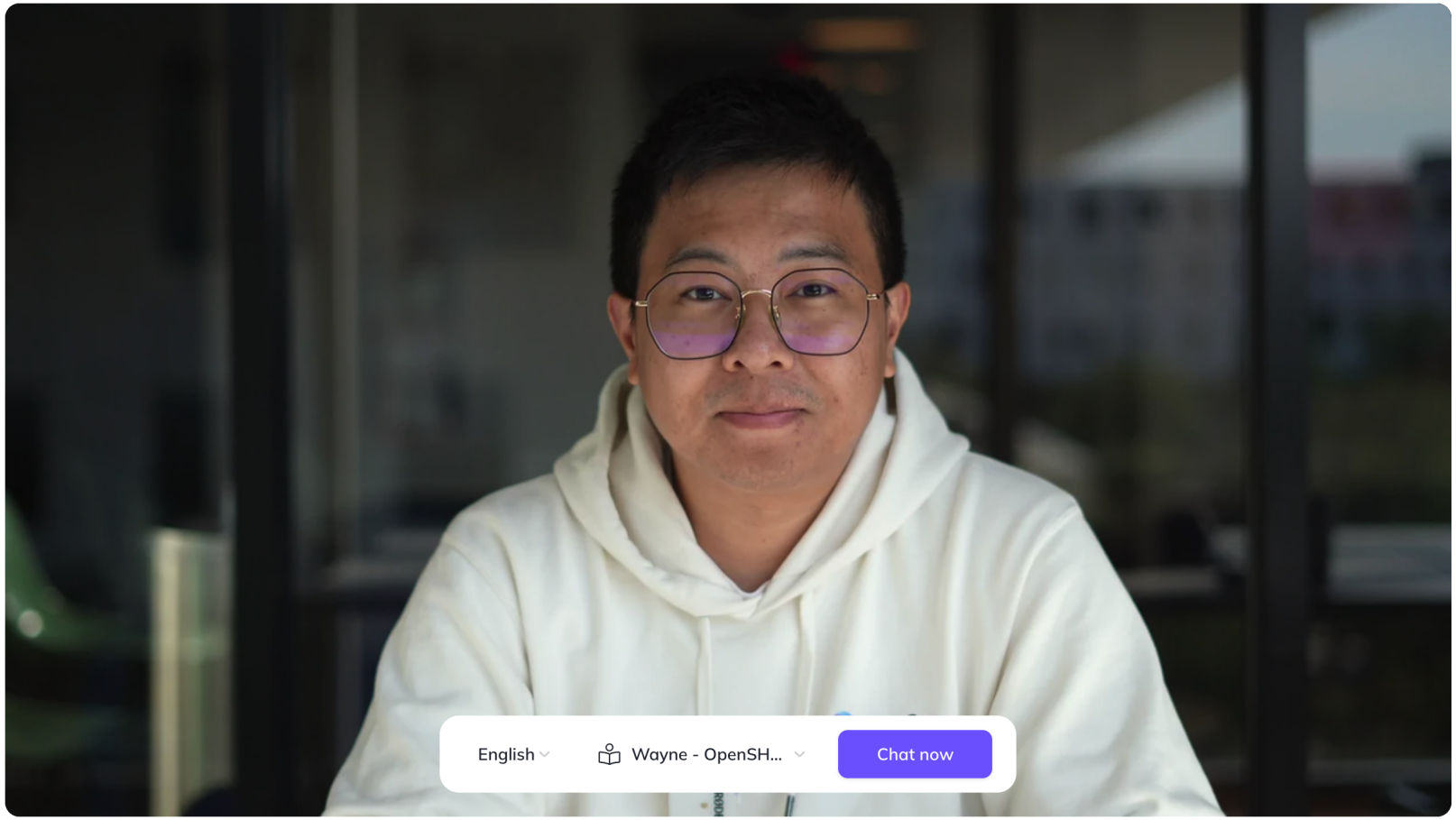
Logistics

- Tutorial material will be placed here: https://github.com/ORNL/OpenSHMEM_Tutorial/
  - Other examples: https://github.com/jdevinney/bale
- We will cover the material and provide demonstrations on how to run the examples
- You will be able to try them out on your own systems during or after the tutorial

Logistics

- Throughout the tutorial, various topics or components of OpenSHMEM will be coded to indicate their importance and guide your attention
  - Essential items will be in green
  - Items in blue may not be required but may be useful and shouldn't prove too difficult to master
  - Anything in red is considered either more advanced or special-use
- Feel free to ask questions at any time — please raise your hand in Teams so we are aware of your question
- This tutorial is designed to be interactive and to encourage conversations with you!

Meet Wayne, our AI Avatar OpenSHMEM Tutorial Assistant!



https://tinyurl.com/363haxdt[1]

[1]May need firewall exceptions to work

OAK RIDGE National Laboratory

June 20, 2025

Module 1: Introduction to PGAS

Aaron Welch[1], Oscar Hernandez[1]

[1]Oak Ridge National Laboratory
oscar@ornl.gov, welchda@ornl.gov

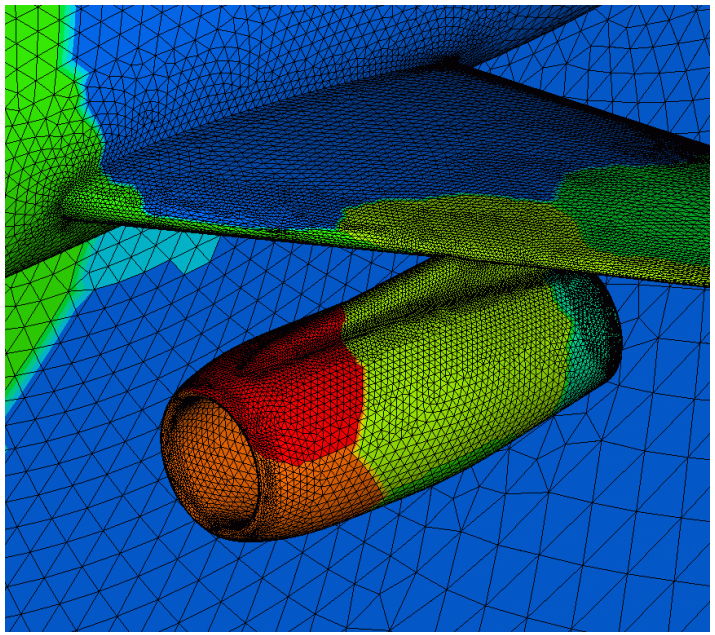
U.S. DEPARTMENT OF ENERGY

ORNL IS MANAGED BY UT-BATELLE LLC FOR THE DEPARTMENT OF ENERGY

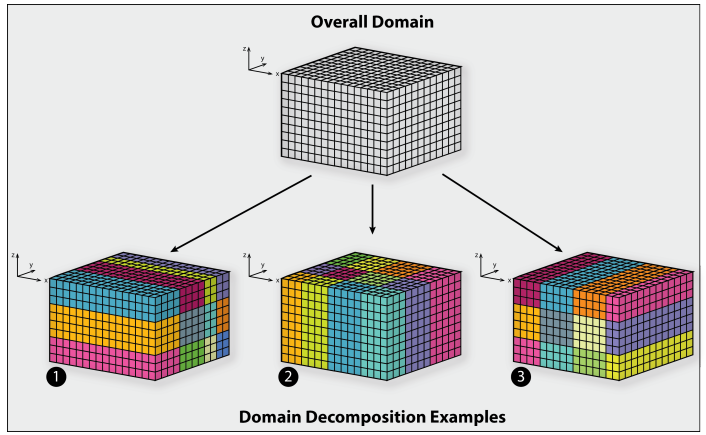
What is Parallel Computing?

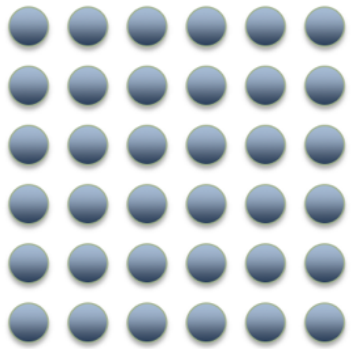
- Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem
- Essential for modern high performance computing (HPC)
- Challenges: thinking in parallel!
  - Data Locality
  - Load imbalance
  - Synchronization
  - Scalability

Serial Parallel



PROBLEM



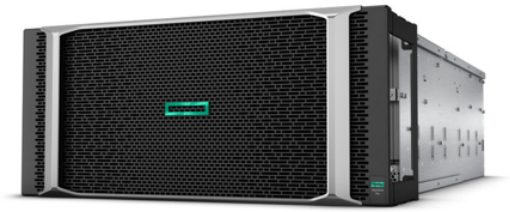


processes

HPC Landscape

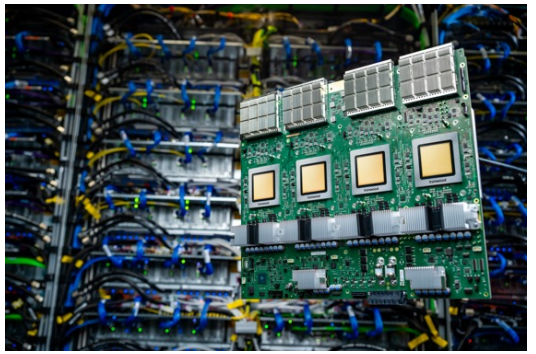
- Symmetric Multi-Processing (SMP)
- Massively Parallel Processing (MPP)
- Accelerator-based Heterogenous Supercomputers
- AI specialized systems

Accelerator-based (AMD GPU MI300A) Supercomputer



Shared Memory System (Xeon CPU) HPE Superdome Flex

MPP (Arm CPU A64fx, 158,976 nodes) Fugaku, Japan



AI specialized — Google TPUv5/Ironwood

Programming Models

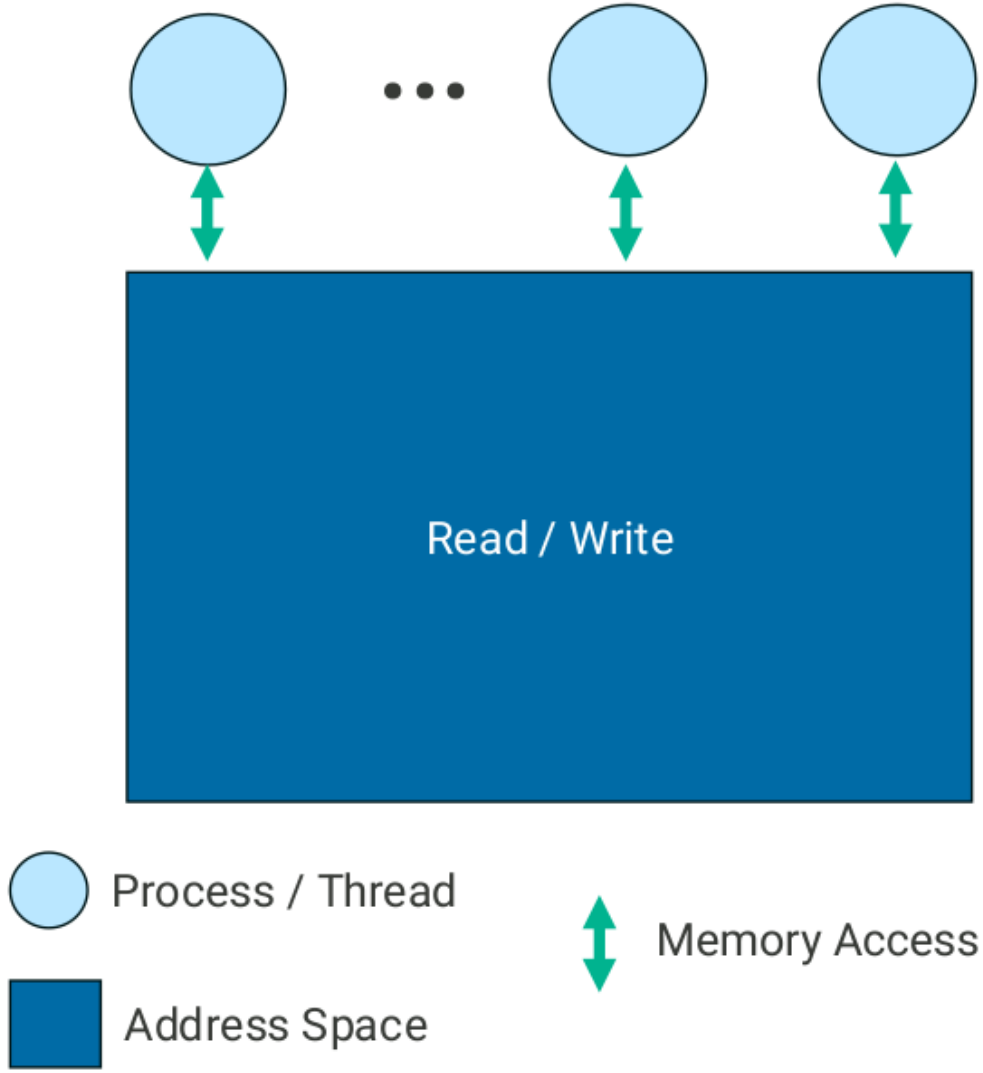
- What is a programming model?
  - The logical interface between architecture and applications
- Why use programming models?
  - Decouple applications and architecture
- Write applications that run effectively across architectures
  - Design new architectures that can effectively support legacy applications
- Programming model design considerations
  - Expose modern architectural features to exploit machine power and improve performance
  - Maintain ease of use

Examples of Parallel Programming Models

- Shared memory
- Message passing
- Partitioned global address space (PGAS)
- Accelerator-based offload models

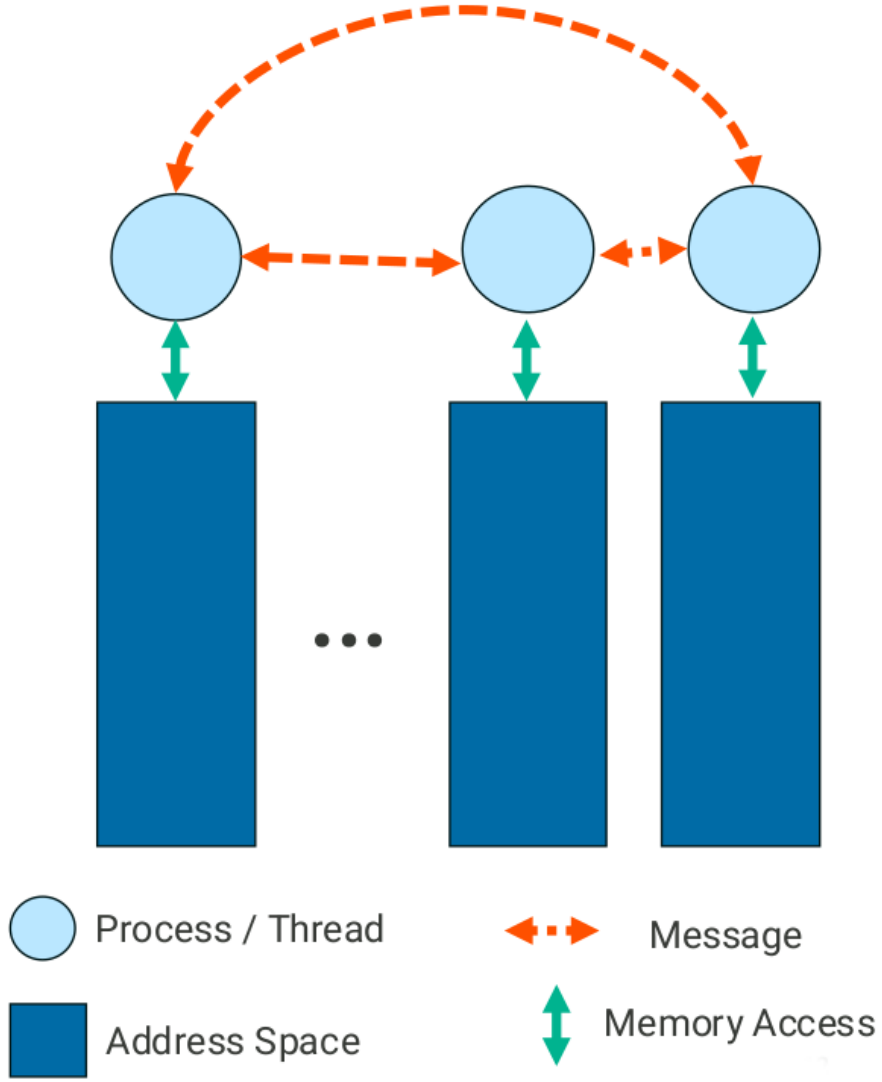
The Shared Memory Model

- Concurrent threads with shared space
- Positives:
  - Simple statements
  - Read memory via an expression
  - Write memory through assignment
- Negatives:
  - Updating shared data requires synchronization
  - Can lead to contention
  - Locality can be explicit or implicit
- Examples: OpenMP, Pthreads



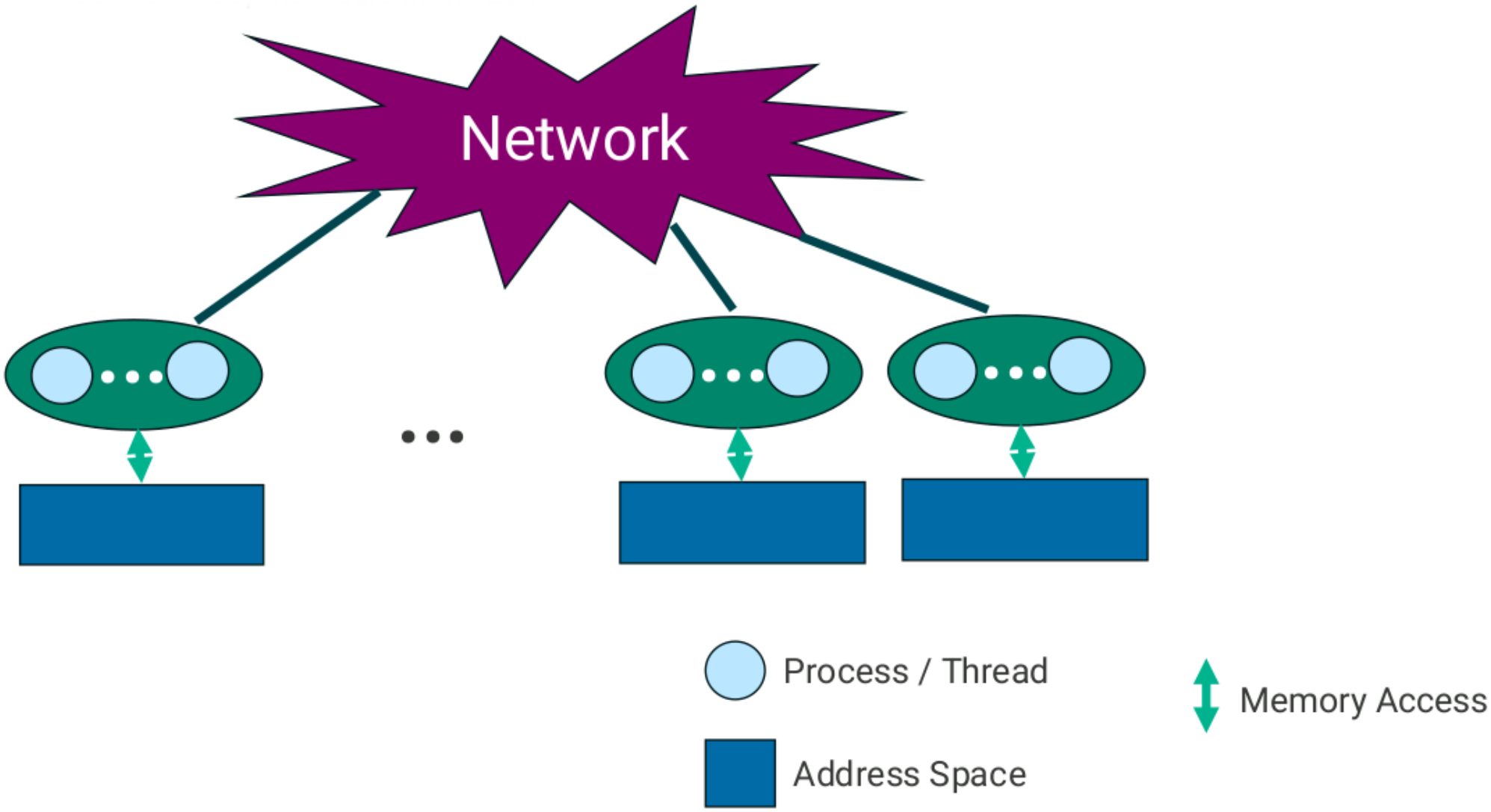
The Message Passing Model

- Concurrent sequential processes
- Explicit communication, two-sided
- Library-based
- Positives:
  - Programmers control data and work distribution
- Negatives:
  - Significatn communication overhead for small transactions
  - Excessive buffering
  - Hard to program in
- Examples: MPI



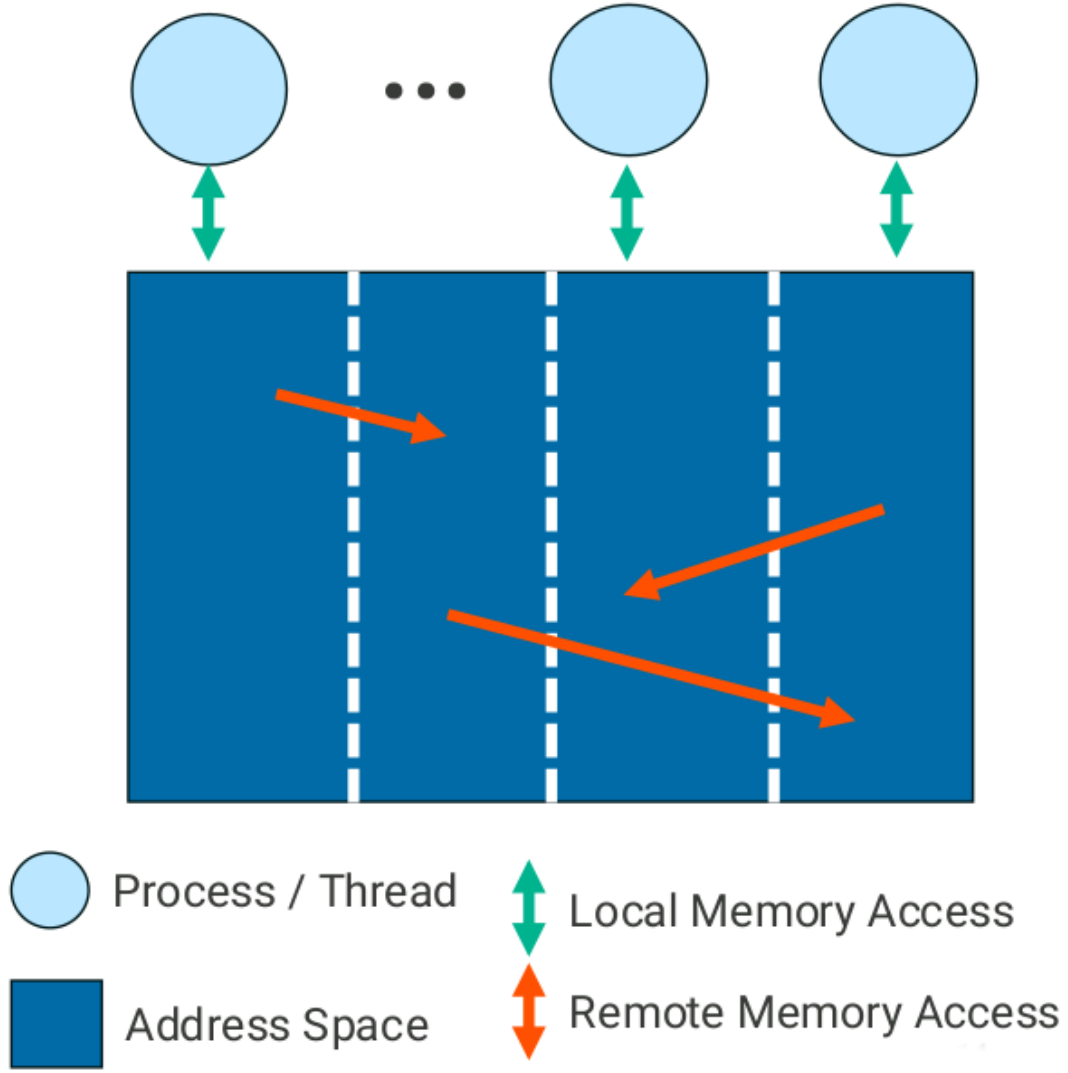
Hybrid Model(s): Shared + Message Passing

- Example: OpenMP at the node, MPI across nodes



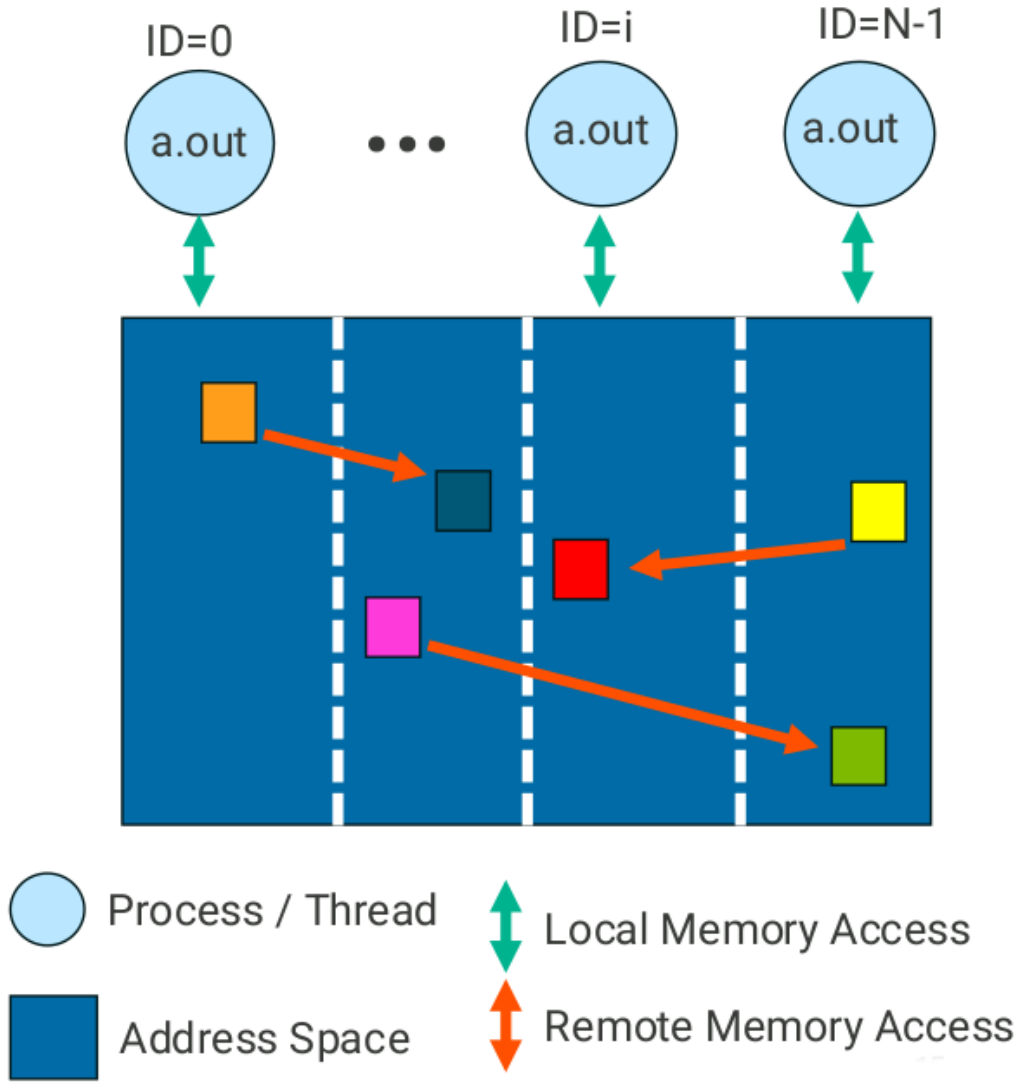
The PGAS Model

- Concurrent threads/processes with a partitioned shared space
  - A datum may reference data in other partitions
  - Global arrays have fragments in multiple partitions
- Positives:
  - Explicit locality
  - Simple statements like shared memory
- Negatives:
  - Synchronization overheads can be very expensive
  - Race conditions
- Examples: Chapel, UPC, OpenSHMEM



What is SPMD?

- Single program, multiple data
- Core concept:
  - All processing elements (PEs) execute the same program
  - Each element operates on a different portion of the data
- Key characteristics:
  - A single executable is launched across all PEs
  - Each PE has a unique identifier to distinguish its role
  - PEs may follow different control paths based on their identifier
  - Designed for scalability on distributed or partitioned systems

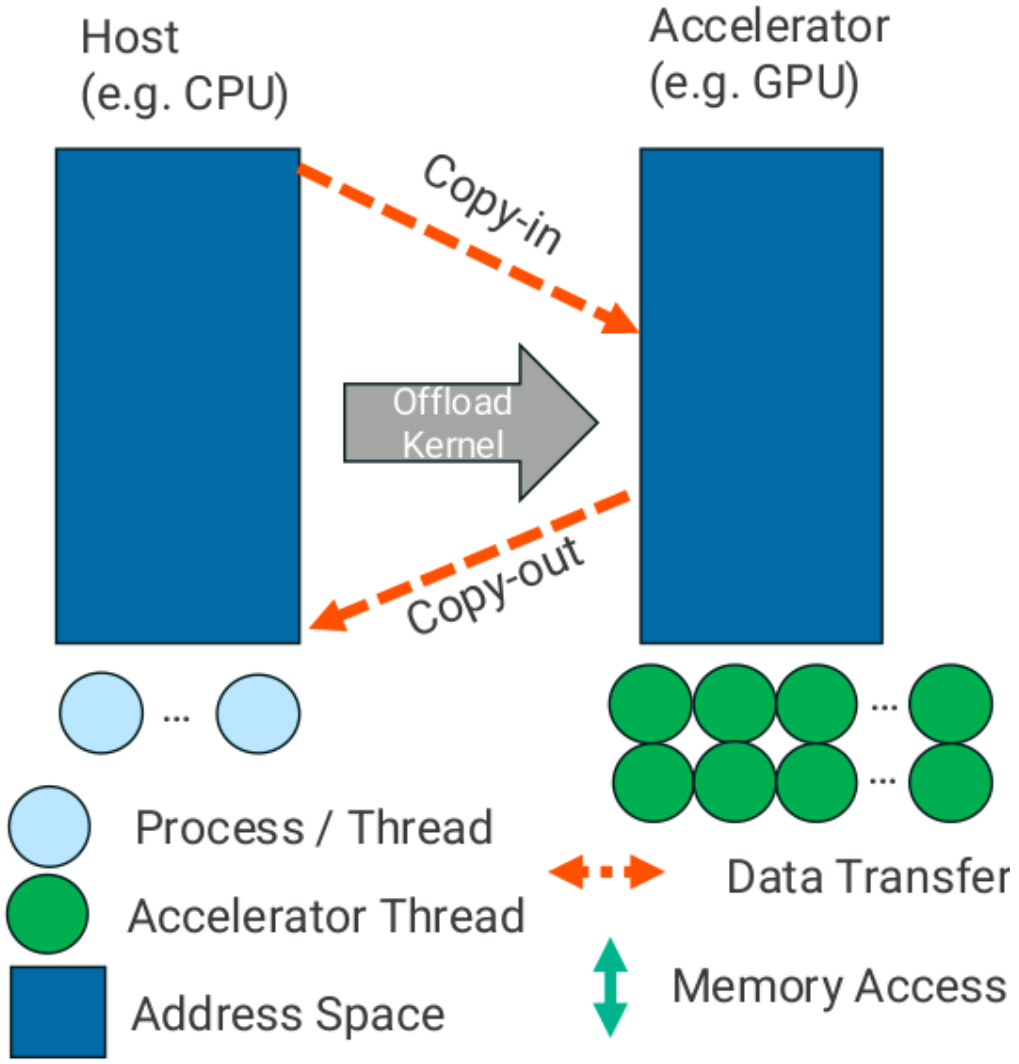


PGAS vs Other Programming Models/Languages

	Chapel, UPC, Fortran Co-arrays	OpenSHMEM	MPI	OpenMP
Memory model	PGAS	PGAS	Distributed memory	Shared memory
Notation	Language	Library	Library	Directive APIs
Global arrays	Yes	Yes	No	No
Global pointers	Yes	Yes	No	No
Locality exploitation	Yes	Yes	Yes	Implicit/explicit (caches, etc.)

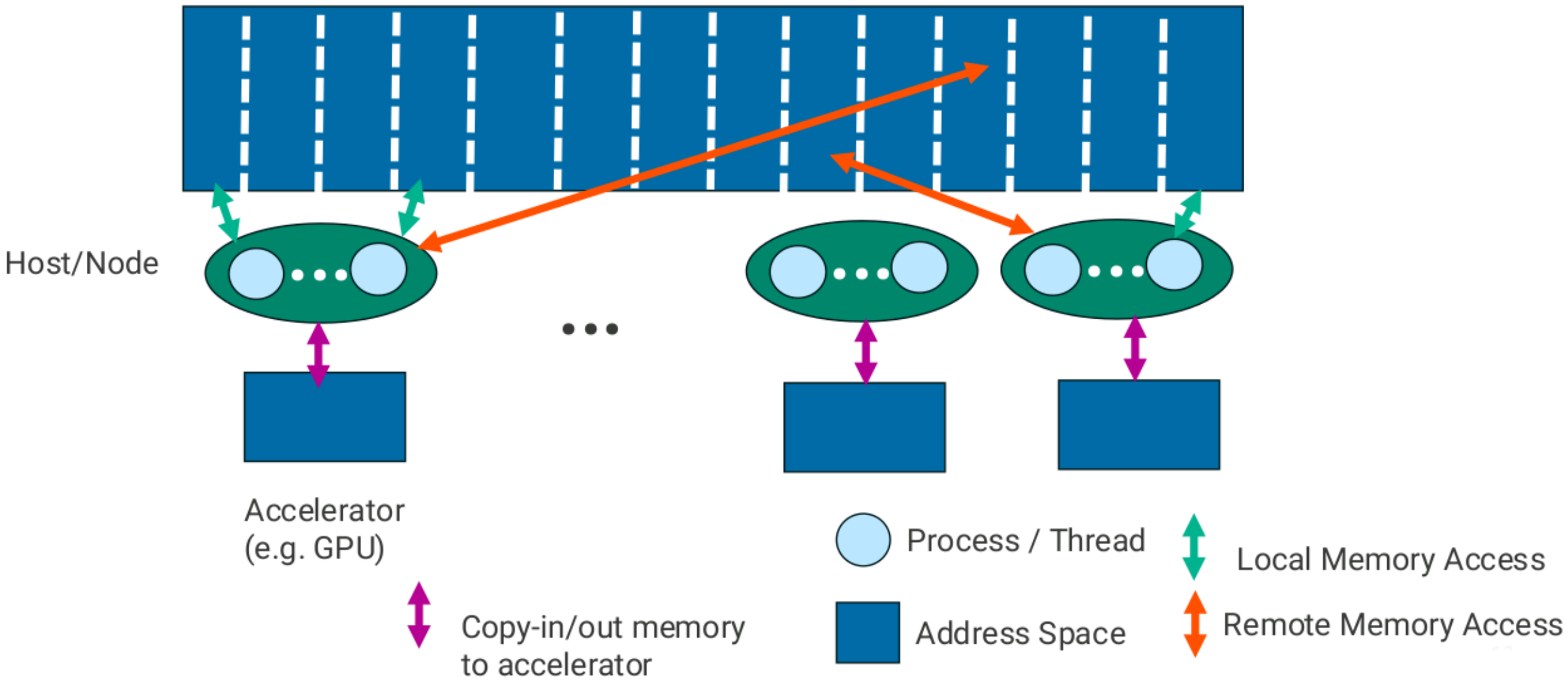
Accelerator-based Model

- Host and device abstractions
- Discrete memories or unified-shared memory
- Copy-in and copy-out data
- Offloading computations to device (e.g., kernels)
- Positives:
  - Programmers can control data movement
- Negatives:
  - Complex to program
  - Missing data transfers can lead to bugs
- Examples: CUDA, HIP, OpenMP target, OpenACC



Hybrid Model: PGAS + Accelerator

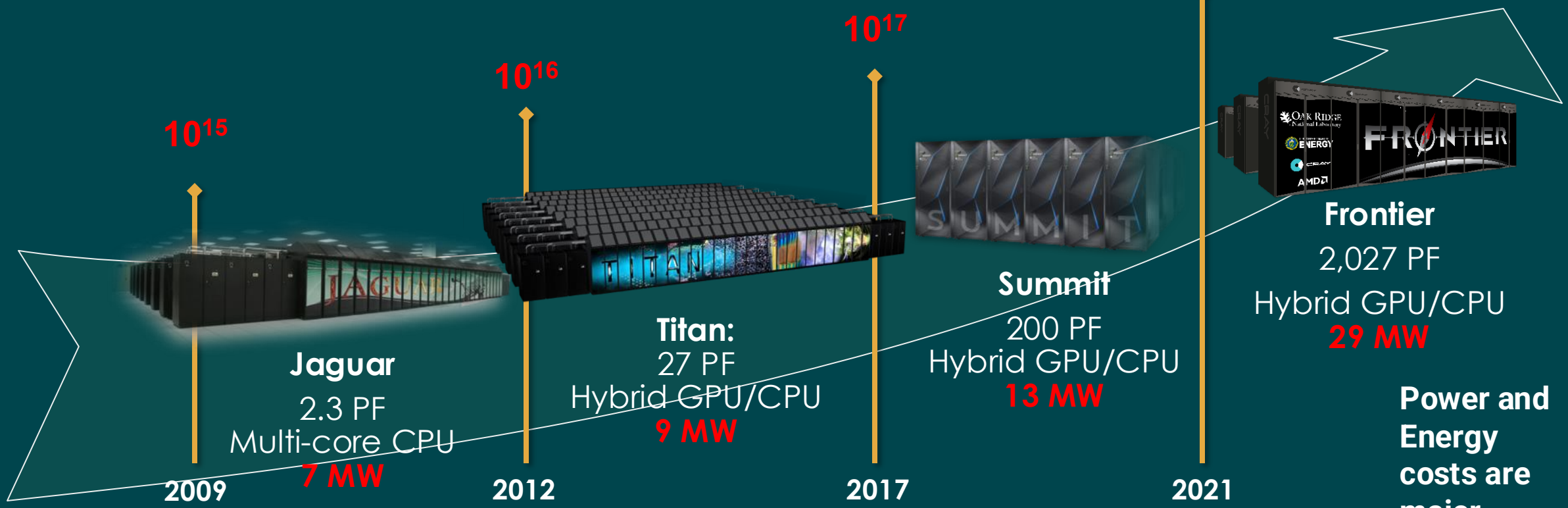
- Example: PGAS across nodes, accelerator within node



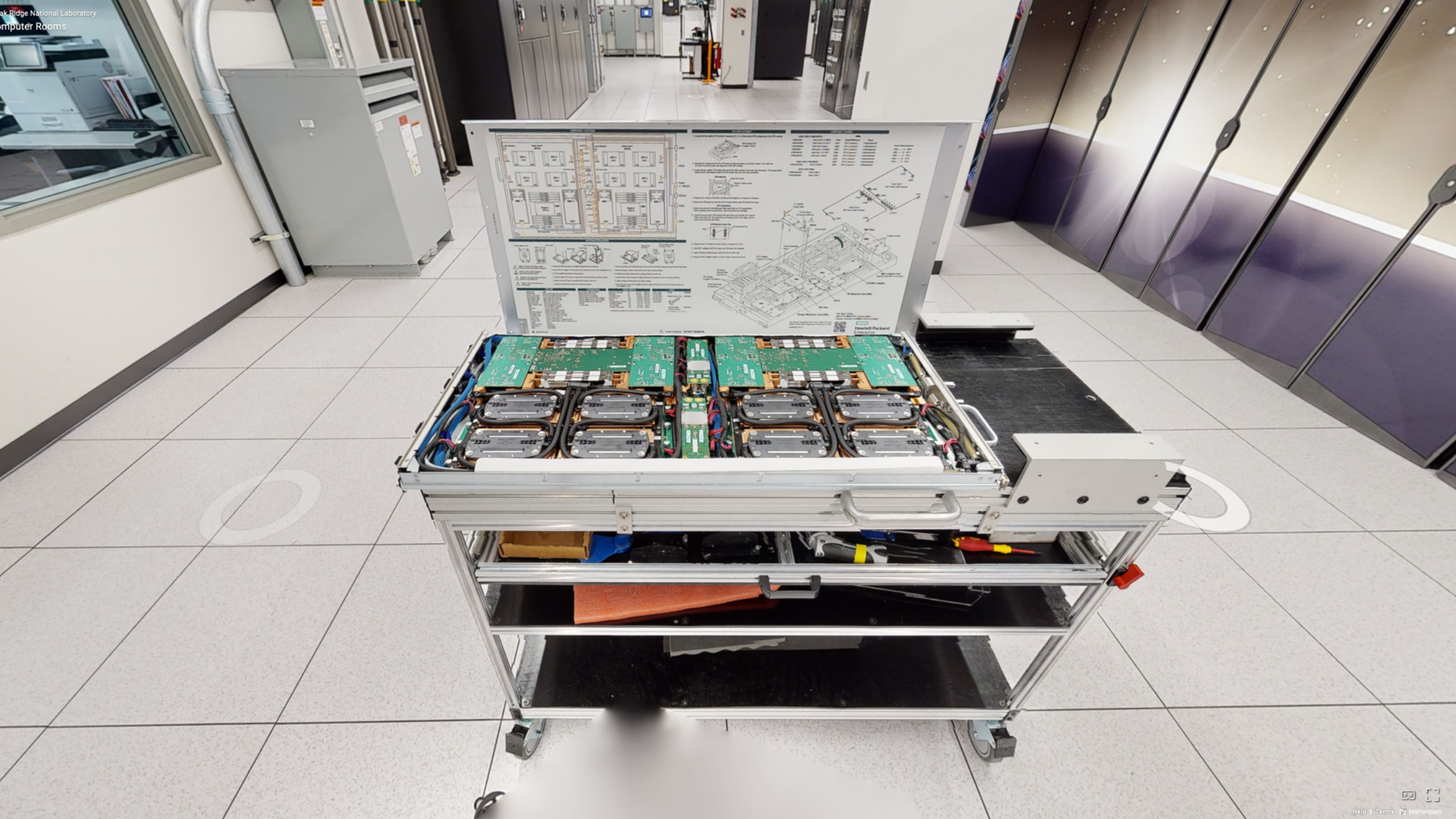
Examples of HPC Architectures

Accelerator-based Heterogenous Supercomputers

- Evolution of Supercomputers
- Deployed exascale computing on GPU systems. Frontier first exascale system. Purpose: **science**
- Accelerator-based architectures for power/energy efficiency




Oak Ridge National Laboratory
Computer Rooms
AU
18
FRONTIER
Help | Terms
Matterport


ak Ridge National Laboratory
mputer Rooms
Hewlett Packard Enterprise
Help | Terms
Matterport

The Frontier Supercomputer

System Size 9,408 nodes 74 compute racks (8,000 lbs each) 128 AMD nodes per rack

Compute Node 1 64-core AMD "Optimized 3rd Gen EPYC" CPU 4 AMD Instinct MI250X GPUs

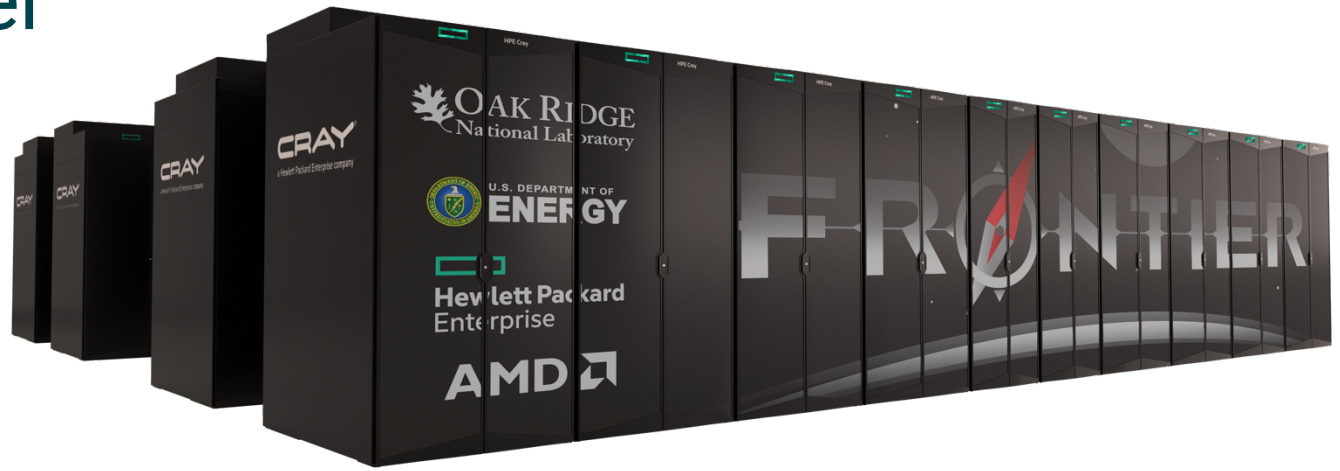
GPU Architecture 37,632 GPUs, each featuring 2 **GCD!**s (**GCD!**s) for a total of 8 **GCD!**s per node

System Interconnect 4-port HPE Slingshot 200 Gbit/s (25 GB/s) NICs providing a node-injection bandwidth of 800 Gbit/s (100 GB/s)

High-Performance Storage 700 PB **HDD!** + 11 PB Flash Performance Tier, 9.4 TB/s and 10 PB Metadata Flash Lustre

Programming Models MPI, OpenMP, OpenACC, CUDA, OpenCL, DPC++, HIP, RAJA, Kokkos, and others

Node Performance 214 TF double precision

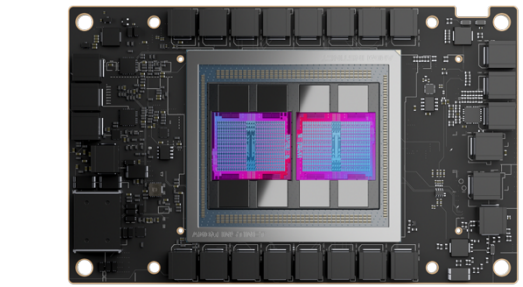


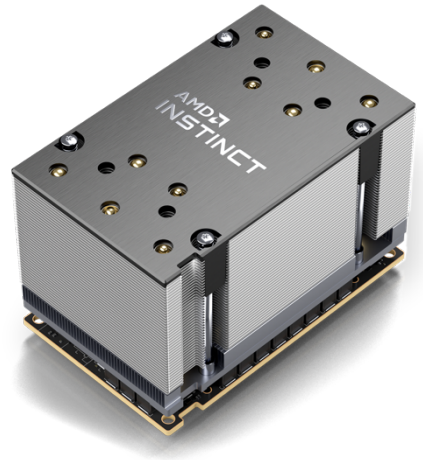
FRONTIER
HPE Cray EX

PEAK PERFORMANCE
1.7 EF double precision

FRONTIER COMPUTE NODES
1 64-core AMD CPU
4 AMD Instinct MI250X GPUs

Debuting in 2022 at 1.102 EF, Oak Ridge National Laboratory's Frontier supercomputer is the world's first exascale system. ORNL's long history of supercomputing excellence enables scientists to expand the scale and scope of their research, solve complex problems in less time, and fill critical gaps in knowledge. The OLCF team tuned the system and improved the Linpack score in May 2023, adding 0.92 EF for a score of 1.194 EF.





HPE Slingshot Interconnect

- High-speed, low-latency network architecture
- Switches provide 64 ports with 25 GB/s bi-directional bandwidth each
- NICs capable of 25 GB/s bi-directional bandwidth per link
- Slingshot is a superset of Ethernet with optimized HPC functionality
- Frontier uses a dragonfly topology

Two cabinets, 32 switches each. Each switch in a cabinet is cabled to every other switch. Black is electrical and teal is optical.

Frontier Network Topology

Dragonfly groups

- A group of endpoints connected to switches that are connected all-to-all

Dragonfly topology

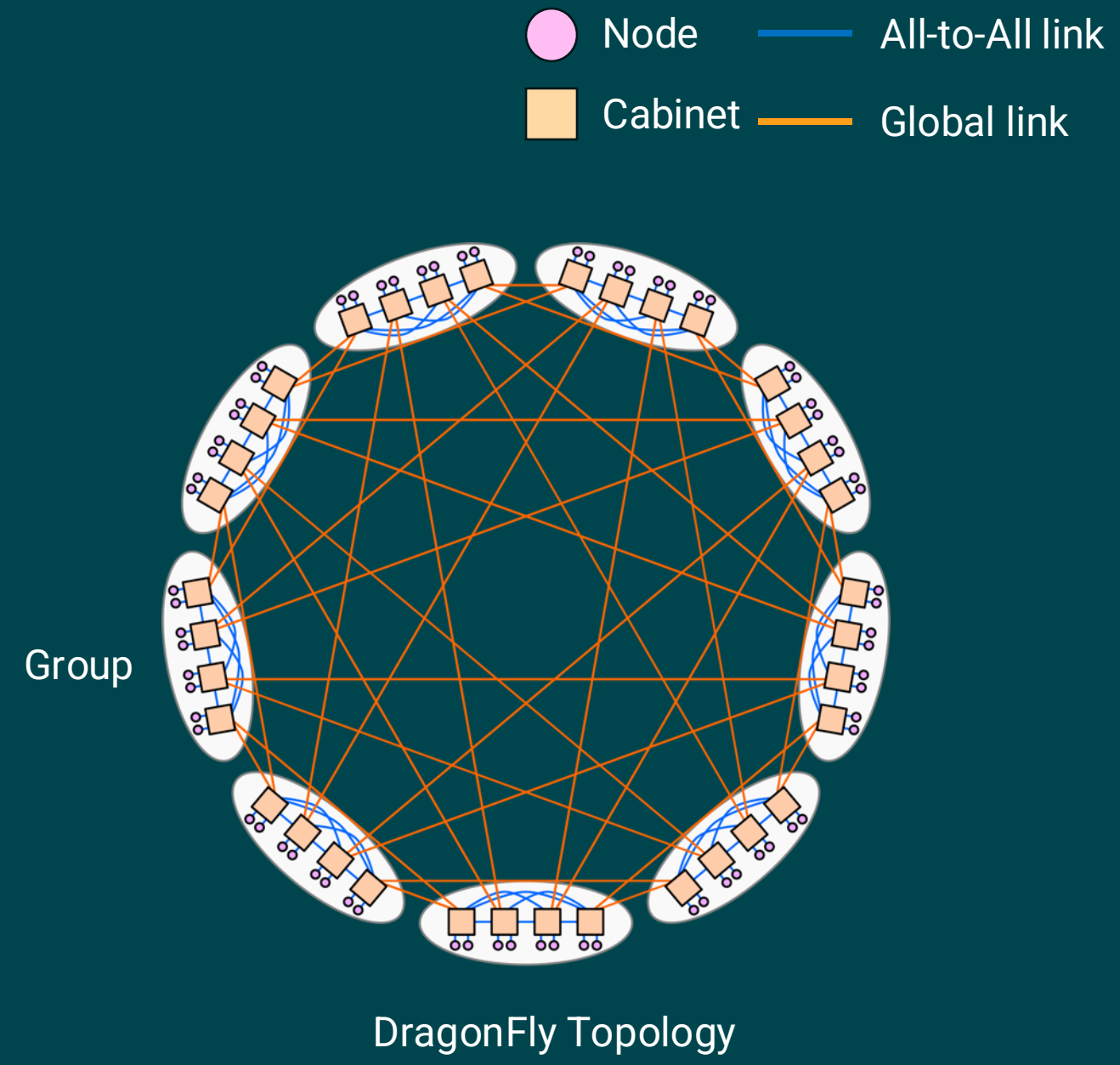
- A set of groups connected all-to-all

Each group has >=1 link to every other group

Frontier has 74 compute groups

- 128 nodes per compute group
- 32 switches per computer group
- 4 NICs per node

Fun Fact: Frontier has >90 miles of cables



DragonFly Topology

Summary

- Programming models provide abstractions for parallel computing systems
- PGAS combines the ease of shared memory programming with the scalability of distributed systems
- OpenSHMEM is a PGAS programming model that provides a fast and portable way to access local and remote memory

Acknowledgements

- For this module, we used and updated materials from these tutorials:

  - Kathy Yelick, Partitioned Global Address Space Programming with Unified Parallel C (UPC) and UPC++

  - David Henty (EPCC), Alan Simpson (EPCC), Harvey Richardson, Bill Long (Cray), Single-sided PGAS Communication Libraries

OAK RIDGE National Laboratory

June 20, 2025

Module 2: OpenSHMEM Basics

Aaron Welch[1], Oscar Hernandez[1]

[1]Oak Ridge National Laboratory
oscar@ornl.gov, welchda@ornl.gov

U.S. DEPARTMENT OF ENERGY

ORNL IS MANAGED BY UT-BATELLE LLC FOR THE DEPARTMENT OF ENERGY

Have You Set Yourself Up for the Demos?

- https://github.com/ORNL/OpenSHMEM_Tutorial

- https://github.com/jdevinney/bale

A Brief History of OpenSHMEM

- SHMEM first developed independently by Cray for the T3D and popularised by its inclusion in SGI's Message Passing Toolkit (MPT)
- As its popularity increased, an increasing number of versions of it started to proliferate to support a widening array of systems and environments
  - These competing versions suffered from numerous incompatibilities and non-standard extensions
- The OpenSHMEM effort unified the different vendors and versions under a single standard API and surrounding community
- Since its inception, it has seen a fairly rapid growth in the development of its feature set to better support modern and future HPC requirements
  - Its latest version is 1.6, released in late 2024, though most implementations are not yet updated to reflect this
  - We will focus this tutorial on the state of OpenSHMEM as of its 1.5 specification, though we will still make some references to features from 1.6 and beyond

OpenSHMEM Overview

- OpenSHMEM is a PGAS library specification aimed at providing a low overhead interface over remote direct memory access (RDMA) hardware
- Designed exclusively around one-sided communication, using a single program, multiple data (SPMD) model
- Communication occurs strictly on “symmetric” data objects[1] from explicitly managed memory regions
- Many implementations including:
  - OSSS
  - OpenMPI
  - Cray OpenSHMEM-X
  - Sandia OpenSHMEM
  - HPCX
  - Intel SHMEM
  - NVSHMEM
  - ROCM SHMEM
- http://openshmem.org/site/
- https://github.com/openshmem-org/specification

[1]The symmetric quality may be loosened in future specification versions

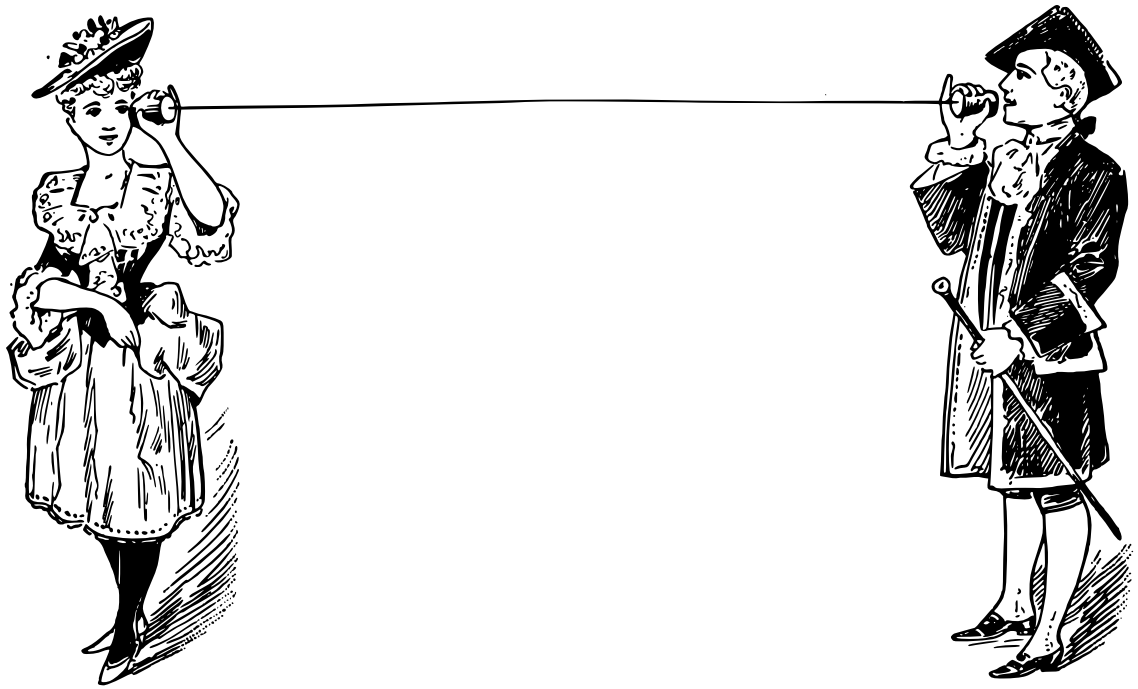
OpenSHMEM Overview

- Execution consists of multiple PEs numbered from 0–($n - 1$) (akin to MPI *ranks*)
- No dynamic process creation (must start with and maintain set PE count)
- Most communication operations are strongly typed according to the data they work on (e.g., `int`, `long`)
- Communication progress managed asynchronously by the implementation (fire and forget!)
- Language bindings provided for C and C++
- Applications typically compiled with `oshcc` and launched with `oshrun`[1]

[1] Best practice may vary by implementation

Two-Sided Communication

- Requires active participation on both sides for the communication to succeed
- Could result in stalling until the other side is ready
- The primary method of transfer for MPI (and most of the Internet!)

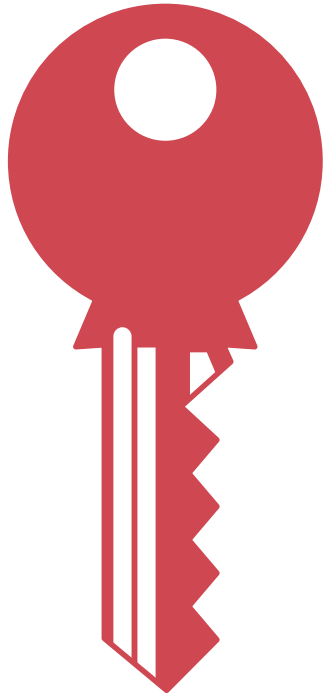


One-Sided Communication

- The receiver does not need to be present to send messages

One-Sided Communication

- . . . or even aware!
- The sender can directly write into the memory at the destination
- Can be much more efficient, but. . .
- Be careful not to shoot your foot!



The OpenSHMEM (PGAS) Memory Model

Can you distinguish. . . 1) local memory, 2) globally accessible memory, and 3) the “partitions” within it?

OpenSHMEM API

- The OpenSHMEM API broadly contains interfaces within the following categories. We will provide a conceptual overview for all of them now, but focus more deeply on the highlighted ones:
  - Setup/Query
  - Memory Management
  - Team management
  - Thread/context management
  - Point to point memory access
  - Synchronisation/ordering
  - Collective operations

Getting Started

- Before we can use any OpenSHMEM functions, the library must first be initialised through a call to `shmem_init()` (or `shmem_init_thread()` if thread support is required)
- Once we are done using it, we should uninitialise it through a call to `shmem_finalize()` to free resoures
  - This will not terminate the application
  - Be sure! You may not be able to reinitialise the library after calling this function
- If anything goes wrong and you want to forcibly terminate the entire application from any PE, you can use `shmem_global_exit()`
- Once the library environment is initialised, we can query the number of PEs with `shmem_n_pes()` and the ID of the calling PE with `shmem_my_pe()`
- You can also query the implementation's vendor with `shmem_info_get_name()` and specification version with `shmemm_info_get_version()`
  - This is also available through the constants `SHMEM_VENDOR_STRING`, `SHMEM_MAJOR_VERSION`, and `SHMEM_MINOR_VERSION`
- Let's try it out!

Hello, World!

Let's Add Some Memory

- As previously established, in order to perform any meaningful communication within OpenSHMEM, we must do so using symmetric data objects
- What qualifies as a symmetric data object?
  - Global variables
  - Static variables
  - Memory explicitly allocated through the OpenSHMEM API
- How do you allocate memory in OpenSHMEM?
  - `shmem_malloc()`, `shmem_calloc()`, `shmem_realloc()`, and `shmem_align()` correlate to the standard memory allocation functions
  - `shmem_free()` is used to release symmetric memory just like the standard `free()`
- We can verify whether an address is remotely accessible from a given PE by querying `shmem_addr_accessible()`
- We may even be able to leverage `shmem_ptr()` to get an address with which we can bypass OpenSHMEM to access directly

Like This?

- No!
- Unlike `malloc()`, OpenSHMEM memory management functions are *collective* across all PEs, so all PEs must participate and manage an equal share of the resulting memory
  - That is what makes them “symmetric”!
- How might we fix this code if PE 0 still wants the memory for some reason?

```
int main(int argc, char **argv) {
    shmem_init();
    if (shmem_my_pe() == 0) {
        int *data = shmem_malloc(sizeof(data[0]) * 32);
        ...
    }
    ...
    shmem_finalize();
}
```

Like This?

- No!
- Unlike `malloc()`, OpenSHMEM memory management functions are *collective* across all PEs, so all PEs must participate and manage an equal share of the resulting memory
  - That is what makes them “symmetric”!
- How might we fix this code if PE 0 still wants the memory for some reason?
  - Make `data` a static variable

```
int main(int argc, char **argv) {
    shmem_init();
    if (shmem_my_pe() == 0) {
        int *data = shmem_malloc(sizeof(data[0]) * 32
        static int data[32];
        ...
    }
    ...
    shmem_finalize();
}
```

Like This?

- No!
- Unlike `malloc()`, OpenSHMEM memory management functions are *collective* across all PEs, so all PEs must participate and manage an equal share of the resulting memory
  - That is what makes them “symmetric”!
- How might we fix this code if PE 0 still wants the memory for some reason?
  - Make `data` a static variable
  - Allocate `data` across all PEs before branching control flow

```
int main(int argc, char **argv) {
    shmem_init();
    int *data = shmem_malloc(sizeof(data[0]) * 32);
    if (shmem_my_pe() == 0) {
        int *data = shmem_malloc(sizeof(data[0]) * 32
        ...
    }
    ...
    shmem_finalize();
}
```

Memory Management Demo

Teams

- Teams provide a method for logically grouping subsets of PEs for a task
- Applications start execution with all PEs being members of the global `SHMEM_TEAM_WORLD`
- Additional teams can be created by splitting them off from another
- PE IDs automatically reindexed with respect to a team
  - Comparable ID query interfaces for teams with `shmem_team_my_pe()` and `shmem_team_n_pes()`
    - What's the difference between the interfaces? Can you rewrite one with the other?
  - Can translate IDs between teams with `shmem_team_translate_pe()`

Teamwork!

Teams

- Teams are represented by opaque “handles” used to reference them in various communication operations
- There are currently two ways of defining new teams. . . if either fails, the result is `SHMEM_TEAM_INVALID`
  - `shmem_team_split_strided()` creates a subteam of a given size representing PEs starting at a given ID and continuing at a chosen interval stride (e.g., PEs 4, 7, 10, 13, . . .)
  - `shmem_team_split_2d()` provides a convenient way of representing a two-dimensional Cartesian grid
- Teams can be created with configurable options to tune their behaviour by providing `shmem_team_config_t` objects during their creation
- There is currently no way to support allocation of remotely accessible memory exclusive to a team
- Teams must eventually be destroyed by passing them to `shmem_team_destroy()`

Maybe Some Examples Might Help. . .

What Are These “Teams” You Speak Of?

- Can’t find them? Check your implementation’s specification compliancy for 1.5+
- Past versions relied exclusively on implicitly defined “active sets”
  - Only useful for collective operations, not point to point communication
  - Requires manual management of internal buffers and synchronisation semantics
- More on this when we look at collectives…

Contexts

- Point to point operations are always performed with respect to a given “communication context”
  - By default, this is `SHMEM_CTX_DEFAULT`
- Contexts allow for a finer granularity of control over completion semantics
- Particularly useful for thread support

Contexts

- Contexts are created via `shmem_ctx_create()` and can be supplied combinations of options to restrict their usage:
  - `SHMEM_CTX_PRIVATE` — context should only be usable by the thread that created it
  - `SHMEM_CTX_SERIALIZED` — context is shareable but should not be concurrently accessed
  - `SHMEM_CTX_NOSTORE` — context will not need to enforce completion/ordering operations
- Failure to create a context results in `SHMEM_CTX_INVALID`
- Contexts can also be made specific to a given team via `shmem_team_create_ctx()`
  - Which team a context corresponds to can be queried with `shmem_ctx_get_team()`
- Contexts must eventually be destroyed by passing them to `shmem_ctx_destroy()`

Point to Point Communiction

- OpenSHMEM provides a variety of remote memory access (RMA) functions for “putting” and “getting” data to/from PEs
  - These can be thought of as roughly the equivalent for load/store operations on remote memory over the network
  - RMA is the primary vehicle for data transfer
- A selection of atomic memory operations (AMOs) are also available to ensure completion of semantic primitives without interference from other AMOs/PEs
- Many operations are available in both blocking and non-blocking variants

Synchronisation/Ordering

- Methods for management the status of PEs and operations issued on them come in two main forms — testing for completion of individual operations, and wholesale management of completion and ordering constraints
  - The former operates irrespective of context, while the latter exclusively so
- *Fences* are useful for ensuring that operations prior to the fence complete before any operations following the fence
- *Quiet* operations ensure that all prior operations are completed
- These are all still one-sided, only concerned with locally issued operations!

Collective Operations

- Collectives operate on entire teams at once, so as to communicate or synchronise in a coordinated fashion
- Currently no non-blocking variants — all PEs in the corresponding team must participate before any can complete the operation
  - No two teams sharing any PEs may have ongoing collectives (deadlock!)
- Relevant memory accesses must not overlap with any other ongoing communications
- Available operations include:
  - Barrier synchronisation
  - Broadcast
  - Collection
  - Reductions

Time for a Break!

We will resume in 15 minutes!

OAK RIDGE National Laboratory

June 20, 2025

Module 3: Point to Point Communication

Aaron Welch[1], Oscar Hernandez[1]

[1]Oak Ridge National Laboratory
oscar@ornl.gov, welchda@ornl.gov

U.S. DEPARTMENT OF ENERGY

ORNL IS MANAGED BY UT-BATELLE LLC FOR THE DEPARTMENT OF ENERGY

Let’s Send Some Data

- RMA functions are at the heart of data transfer in OpenSHMEM
  - “Put” contents of local buffer to remotely accessible memory
  - “Get” contents of remotely accessible memory and store to local variable
- RMA API provides many composable forms based on combinations of parametres such as:
  - Data type/size
  - Synchronous/asynchronous execution
  - Access Pattern
  - Communication context
- All variations are simply methods for transferring data from a single source PE to a single destination PE

Forms of RMA Function Signatures

- The general form for an RMA operation is `shmem_[ctx_]<TYPENAME_><OPERATION>[_nbi]()`
- What?! Let's break it down...

Forms of RMA Function Signatures

- The general form for an RMA operation is `shmem_[ctx_]<TYPENAME_>put[_nbi]()`
- What?! Let's break it down...
  - First determine the operation you want (e.g., "put")

Forms of RMA Function Signatures

- The general form for an RMA operation is `shmem_[ctx_]int_put[_nbi]()`
- What?! Let's break it down...
  - First determine the operation you want (e.g., "put")
  - Then determine the data type of the variable you want to transfer

Forms of RMA Function Signatures

- The general form for an RMA operation is `shmem_[ctx_]int_put~~_nbi~~()`
- What?! Let's break it down…
  - First determine the operation you want (e.g., "put")
  - Then determine the data type of the variable you want to transfer
  - Determine whether you want a blocking or non-blocking ("nbi") interface (more on the coming slides!)

Forms of RMA Function Signatures

- The general form for an RMA operation is `shmem_ctx_int_put()`
- What?! Let's break it down…
  - First determine the operation you want (e.g., "put")
  - Then determine the data type of the variable you want to transfer
  - Determine whether you want a blocking or non-blocking ("nbi") interface (more on the coming slides!)
  - Determine whether you want to communicate on the default/global context or choose a different one

Forms of RMA Function Signatures

- The general form for an RMA operation is `shmem_ctx_int_put()`
- What?! Let's break it down…
  - First determine the operation you want (e.g., "put")
  - Then determine the data type of the variable you want to transfer
  - Determine whether you want a blocking or non-blocking ("nbi") interface (more on the coming slides!)
  - Determine whether you want to communicate on the default/global context or choose a different one
- Not all possible combinations may be possible — consult the documentation!

C11 is Magic!

- If you're using the C API, the data type and context selections can be generically inferred based on the arguments provided
- Now the previous call could simply be reduced to: `shmem_put()` (or `shmem_get()`, etc.)
- The explicit form is still supported, and remains the only option portable across both C and C++

Blocking Operations

- Blocking operations won't return until local completion is achieved
  - This is usually what you would expect by default
- Cool! . . . but what do we mean by "local completion"?
  - Any data to be copied from/to local buffers has been, and may be accessed or reused immediately
  - There is no guarantee that anything has been received or operated on at the destination PE
- In practice, this means that gets have fully completed, but data from puts may not be received/visible until some indeterminate time in the future

Non-Blocking Operations

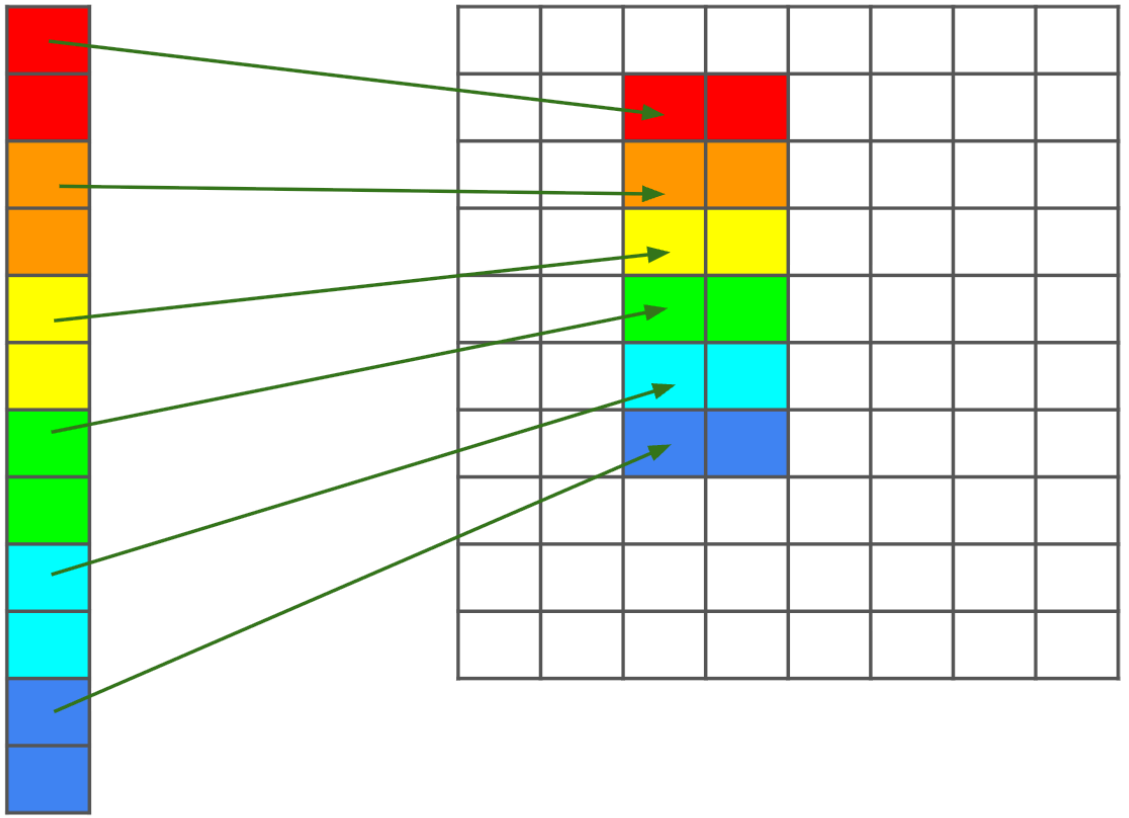
- In contrast, non-blocking operations wait for nobody!
- There is no guarantee that local buffers are reusable upon return
  - If you try to access them prematurely, you may either alter the result of the prior operation, or receive stale results!
- This may seem undesirable, but in many cases you may not immediately need to access the data in question
- The reward is greater performance potential!

What RMA Operations are Available?

- `shmem_p` — put a single value to destination PE
- `shmem_put` — put *n* values to destination PE
- `shmem_iput` — copy *strided* data to destination PE
- `shmem_ibput` — copy strided data *blocks* to destination PE (OpenSHMEM 1.6+)
- `shmem_g` — get a single value from destination PE
- `shmem_get` — get *n* values from destination PE
- `shmem_iget` — copy *strided* data from destination PE
- `shmem_ibget` — copy strided data *blocks* from destination PE (OpenSHMEM 1.6+)

“Stride”? Now You’re Messing With Me!

- It’s simpler than it may seem
- Strided operations copy *n* elements each spaced *s* elements apart
  - Compare this to the contiguous put/get operations copying *n* elements each spaced 0 elements apart
- The block-strided operations extend this further by copying *b* blocks of *n* contiguous elements, each spaced *s* elements apart



Enough Abstraction!

- Alright, let's look at something simple...
- Here we allocate the symmetric variable `remote_var`, and put the value of `local_var` (our own PE ID) to the copy of `remote_var` on PE 0
- Then we get the value of `remote_var` on PE 0 and store it back to `local_var`
- What is the value of `local_var` at the end?

```
int *remote_var = shmem_malloc(sizeof(*remote_var));
int local_var = shmem_my_pe();
int pe = 0;
shmem_p(remote_var, local_var, pe);
local_var = shmem_g(remote_var, pe);
```

Here Be Dragons

- Which PEs are executing your code?

```
int *remote_var = shmem_malloc(sizeof(*remote_var));
int local_var = shmem_my_pe();
int pe = 0;
shmem_p(remote_var, local_var, pe);
local_var = shmem_g(remote_var, pe);
```

Here Be Dragons

- Which PEs are executing your code?
  - All of them!
- When is the data from your puts received?

```
int *remote_var = shmem_malloc(sizeof(*remote_var));
int local_var = shmem_my_pe();
int pe = 0;
shmem_p(remote_var, local_var, pe);
local_var = shmem_g(remote_var, pe);
```

Here Be Dragons

- Which PEs are executing your code?
  - All of them!
- When is the data from your puts received?
  - Who knows?
- If you perform a put followed by a get, will the get complete after the put does?

```
int *remote_var = shmem_malloc(sizeof(*remote_var));
int local_var = shmem_my_pe();
int pe = 0;
shmem_p(remote_var, local_var, pe);
local_var = shmem_g(remote_var, pe);
```

Here Be Dragons

- Which PEs are executing your code?
  - All of them!
- When is the data from your puts received?
  - Who knows?
- If you perform a put followed by a get, will the get complete after the put does?
  - Maybe. . . or maybe not!
- What do we need?

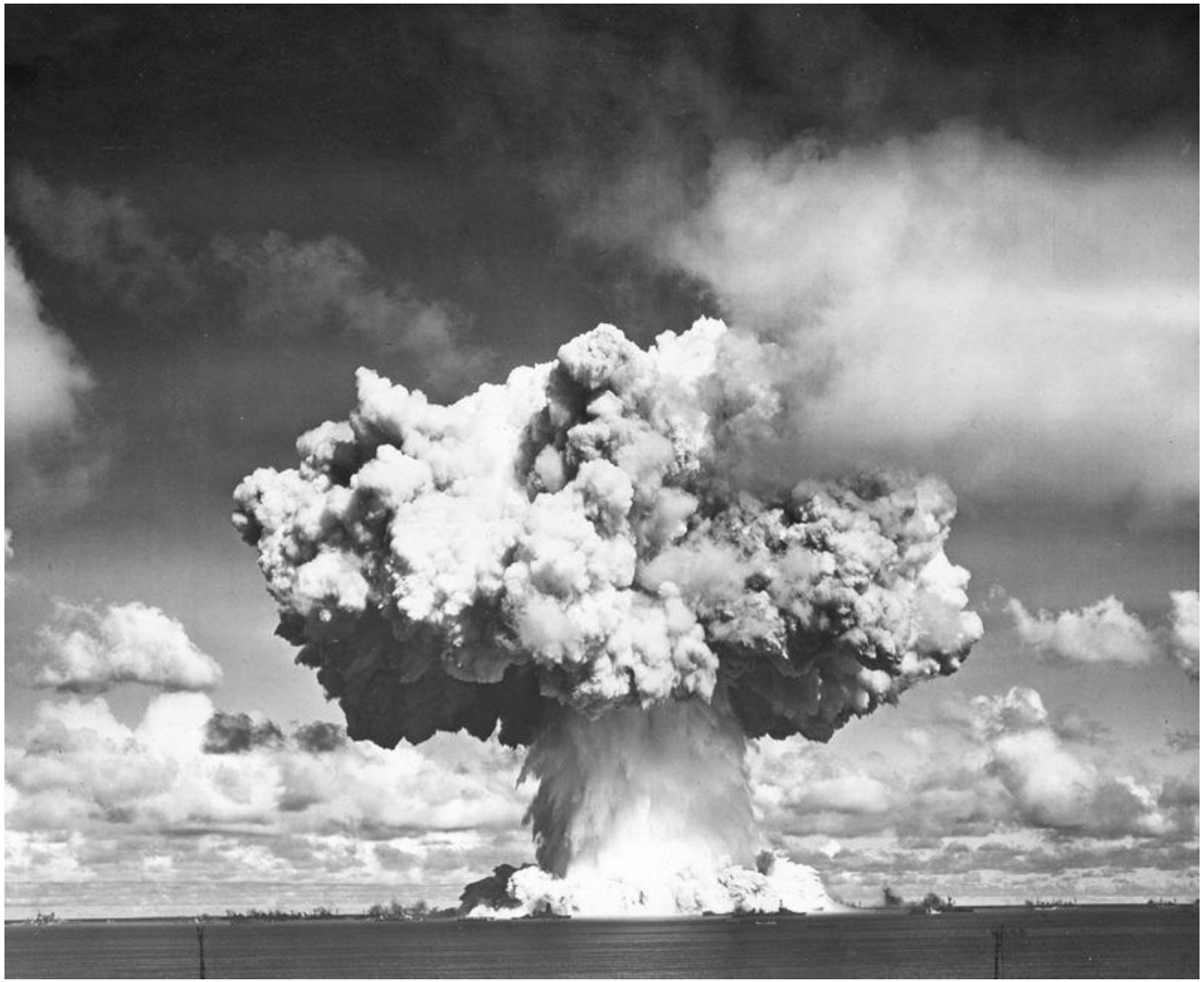
```
int *remote_var = shmem_malloc(sizeof(*remote_var));
int local_var = shmem_my_pe();
int pe = 0;
shmem_p(remote_var, local_var, pe);
local_var = shmem_g(remote_var, pe);
```

Here Be Dragons

- Which PEs are executing your code?
  - All of them!
- When is the data from your puts received?
  - Who knows?
- If you perform a put followed by a get, will the get complete after the put does?
  - Maybe. . . or maybe not!
- What do we need?
  - The next module! To be continued! (booo!)

```
int *remote_var = shmem_malloc(sizeof(*remote_var));
int local_var = shmem_my_pe();
int pe = 0;
shmem_p(remote_var, local_var, pe);
local_var = shmem_g(remote_var, pe);
```

Atomic Operations



No, not that kind!

Atomic Operations

- Atomic operations are designed such that they effectively appear to access/update memory instantly
  - This is not the same as the operation completing instantly after issuing it
- OpenSHMEM provides a limited set of AMO types, generally expected to be implemented in hardware
- RMA operations can *not* access the same memory as AMOs without breaking their atomicity guarantees
- Synchronisation must be enforced between such accesses to remain valid. . . to be continued in the next module! (booo!)
- AMOs also come in blocking and non-blocking variants

```
int value = shmem_g(remote_var, pe);
shmem_p(remote_var, value + 1, pe);
```

Dragons!

```
shmem_atomic_add(remote_var, 1, pe);
```

No dragons!

What AMOs are Available?

- `shmem_atomic_fetch[_nbi]` — atomically fetch remote value
- `shmem_atomic_set` — atomically set remote value
- `shmem_atomic_swap[_nbi]` — swap remote value with that of local variable
- `shmem_atomic_compare_swap[_nbi]` — swap, but only if remote variable has specified value
- Additionally, the following can either simply update remote memory or also fetch the previous contents:
- `shmem_atomic_[fetch_]inc[_nbi]` — add 1 to remote value
- `shmem_atomic_[fetch_]add[_nbi]` — add arbitrary amount to remote value
- `shmem_atomic_[fetch_]and[_nbi]` — atomically perform bitwise “and” on remote value
- `shmem_atomic_[fetch_]or[_nbi]` — atomically perform bitwise “or” on remote value
- `shmem_atomic_[fetch_]xor[_nbi]` — atomically perform bitwise “xor” on remote value

Demo Time!

OAK RIDGE National Laboratory

June 20, 2025

Module 4: Control Flow and Collectives

Aaron Welch[1], Oscar Hernandez[1]

[1]Oak Ridge National Laboratory
oscar@ornl.gov, welchda@ornl.gov

U.S. DEPARTMENT OF ENERGY

ORNL IS MANAGED BY UT-BATELLE LLC FOR THE DEPARTMENT OF ENERGY

What Have We Learned?

- We can send data...but we don't know when it's received and able to be read
- We want to communicate execution state as well as data...we want synchronisation!
- This can come in a few forms:
  - Point to point synchronisation
  - Enforcing ordering of operations
  - Ensuring remote completion of operations
  - Signalling oprations that combine the transmission of data with notification of its completion

Point to Point Synchronisation

- If all you need to do is ensure a particular message is received, you may either *test* or *wait* for completion
- These operations compare a symmetric variable against a known value — if the comparison passes, it succeeds
  - What previous value(s) are only possible before communication has completed?
  - Why can't we just use `if`/`while` instead of test/wait? AMOs!
- Testing allows execution to continue if the data is not ready yet, but may result in repeated polling
- Waiting avoids this, but will block indefinitely until the data is visible
  - What happens if it's never received, or the condition it's checking against is unreliable?

Comparison Conditions

- Comparisons are described by the combination of an operator and a value
  - For instance, `shmem_wait_until(svar, SHMEM_CMP_NE, 0)` will wait until `svar` is no longer 0
- The supported comparison operators can be seen below:

Constant Name	Comparison
SHMEM_CMP_EQ	Equal
SHMEM_CMP_NE	Not equal
SHMEM_CMP_GT	Greater than
SHMEM_CMP_GE	Greater than or equal to
SHMEM_CMP_LT	Less than
SHMEM_CMP_LE	Less than or equal to

The “Complete” List of Test/Wait Variants

- `shmem_<test|wait_until>` — check against a single variable
- `shmem_<test|wait_until>_all` — check against all variables in an array
- `shmem_<test|wait_until>_any` — check if at least one variable in an array satisfies the condition
- `shmem_<test|wait_until>_some` — same as above but returns status for all variables
- The below are the same as the above, but take an array of values to compare each variable against instead of reusing the same one:
- `shmem_<test|wait_until>_all_vector`
- `shmem_<test|wait_until>_any_vector`
- `shmem_<test|wait_until>_some_vector`

Ensuring Access Order and Delivery

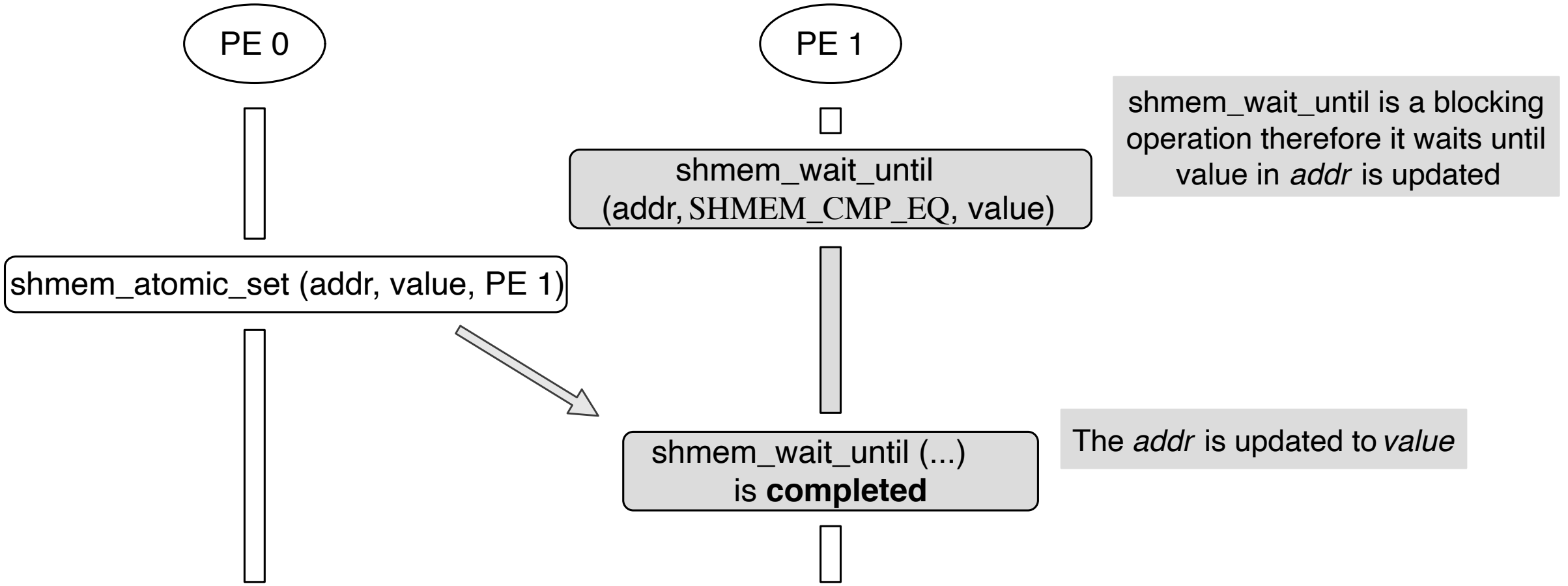
- Testing/waiting is very useful, but what if we want to synchronise over other kinds of communication or at a coarser granularity?
- Adding a “fence” ensures that all operations issued after it are completed after all operations issued before it
- Performing a “quiet” operation ensures that all previously outstanding communication is completed by the the time the quiet returns
- These are both only with respect to the calling PE, not the world!
- These are not meant for ensuring multiple PEs are at the same point in execution (we’ll get to that!)

Where Are My Pictures? ☹

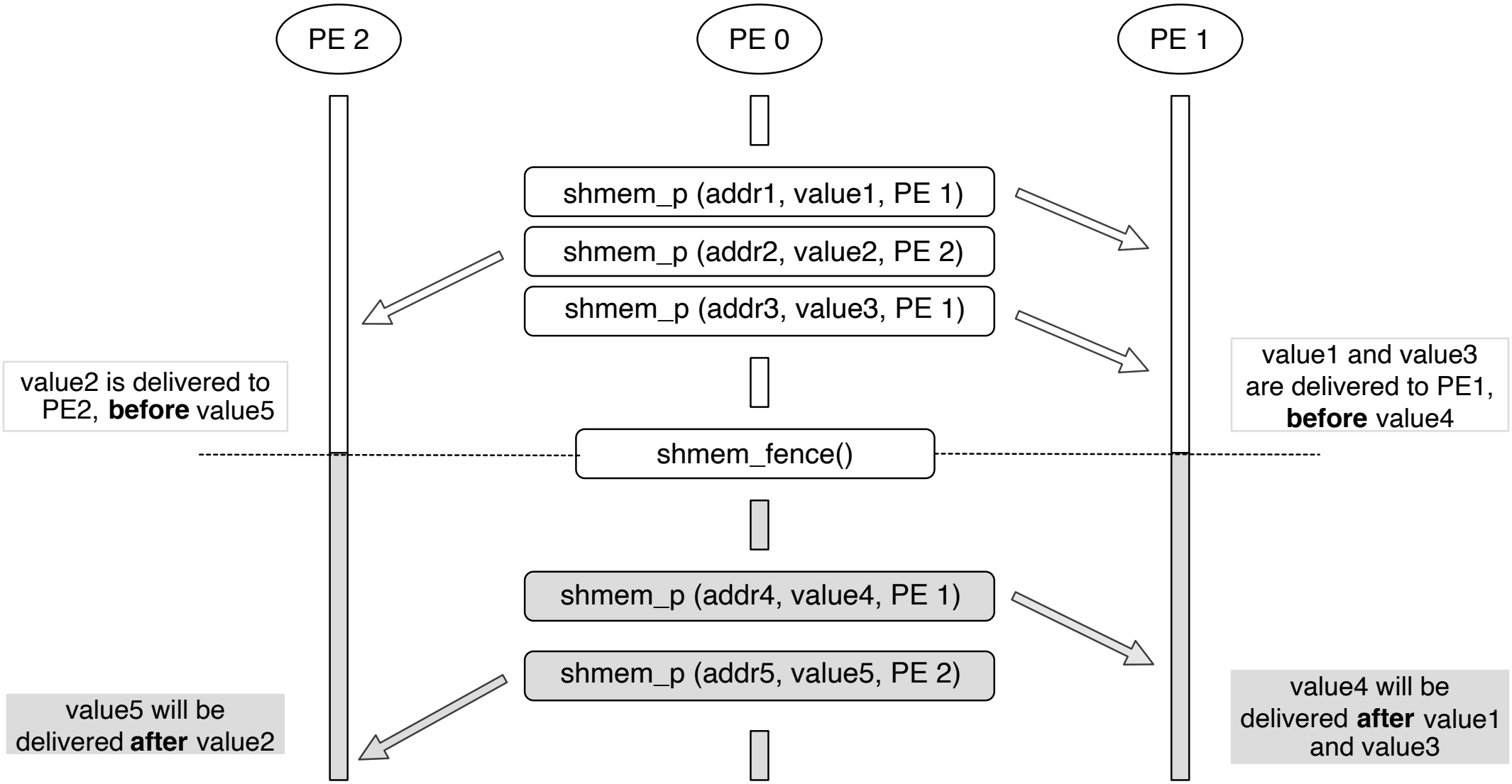
- I’ve got you covered!

- Let’s look at and compare what our point to point, fence, and quiet synchronisation methods look like. . .

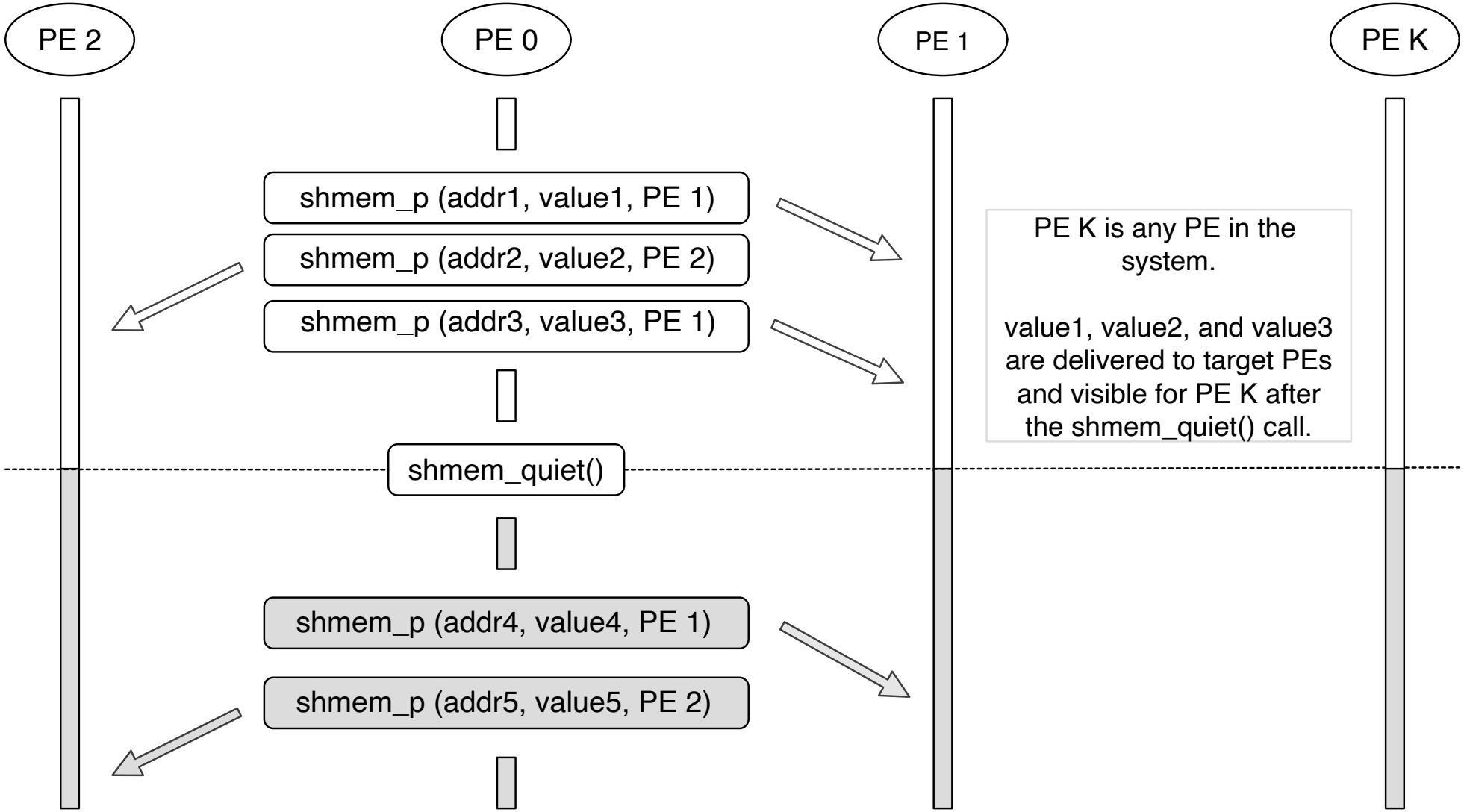
Wait Until


PE 0
PE 1
shmem_wait_until
(addr, SHMEM_CMP_EQ, value)
shmem_wait_until is a blocking operation therefore it waits until value in *addr* is updated
shmem_atomic_set (addr, value, PE 1)
shmem_wait_until (...)
is **completed**
The *addr* is updated to *value*

Fence


PE 2
PE 0
PE 1
shmem_p (addr1, value1, PE 1)
shmem_p (addr2, value2, PE 2)
shmem_p (addr3, value3, PE 1)
shmem_fence()
shmem_p (addr4, value4, PE 1)
shmem_p (addr5, value5, PE 2)
value2 is delivered to PE2, **before** value5
value1 and value3 are delivered to PE1, **before** value4
value5 will be delivered **after** value2
value4 will be delivered **after** value1 and value3

Quiet


PE 2
PE 0
PE 1
PE K
shmem_p (addr1, value1, PE 1)
shmem_p (addr2, value2, PE 2)
shmem_p (addr3, value3, PE 1)
shmem_quiet()
shmem_p (addr4, value4, PE 1)
shmem_p (addr5, value5, PE 2)
PE K is any PE in the system.
value1, value2, and value3 are delivered to target PEs and visible for PE K after the shmem_quiet() call.

Signalling Operations

- Send data *and* notify when its transmission is complete — the best of both worlds!
- The “notification” is performed by updating a remote flag
  - In effect, this is similar to initiating a transfer, performing a fence, then atomically updating a location on the receiver
  - If the receiver observes the update, it knows that the associated transfer is complete
  - . . . yet it’s all executed as a single operation, no extra synchronisation nonsense required!
- The remote flag can be updated using one of two operators:
  - `SHMEM_SIGNAL_SET` — set the flag to a specified value
  - `SHMEM_SIGNAL_ADD` — add a specified value to the flag
- Don’t mix signal operators, or signal flags with RMA or AMOs!

What Signalling Operations Are Available?

- `shmem_put_signal[_nbi]` — signalling put!
- `shmem_signal_fetch` — retrieve the value of the signal flag
- `shmem_signal_set` (1.6+) — atomically set a chosen signal flag
- `shmem_signal_add` (1.6+) — atomically add to a chosen signal flag
- Also. . . I lied earlier, there's one more wait operation:
  - `shmem_signal_wait_until` — wait until a signal flag satisfies a given comparison

We’re Ready!

- . . . almost!

- We have one more major topic to discuss. . .

Collective Operations

- Collective operations are those that operate on entire teams at once
- Can be used for either synchronisation or data transfer
- Should *not* be operated on by multiple threads simultaneously
- *Should* be called by all members of the team with the same arguments and in the same order
- Collective API is split between new interface using teams and old interface using active sets
  - “Active sets?”, you ask? In a moment. . .

Collective Operations

- `shmem_sync` establishes a synchronisation point — no PE can leave until the last PE in the team enters
- `shmem_sync_all` is equivalent to doing a sync on `SHMEM_TEAM_WORLD`
- `shmem_barrier` performs a sync on `SHMEM_CTX_DEFAULT` followed by a sync
  - This is deprecated and should not be used anymore, but you might come across it in existing code
  - . . . however, `shmem_barrier_all` does the same thing, but for `SHMEM_TEAM_WORLD` and is not deprecated

Collective Operations

- `shmem_broadcast` distributes a block of data from one PE to the rest
- `shmem_collect` and `shmem_fcollect` concatenate data from all members of the team, to all members of the team
  - While fcollect concatenates a *fixed* number of elements from each member PE, collect allows for a variable number
- Reductions perform an operation on data across member PEs to *reduce* the result down to a single value (e.g., adding all values together)
- `shmem_alltoall` and `shmem_alltoalls` swap fixed amounts of either contiguous or strided data between all members, respectively
  - This is effectively like performing a matrix transpose — if each PE's symmetric array represented a row of data prior to the call, it will instead contain a column afterward

An Illustration of All to All

- For a symmetric variable containing four elements on each of PEs a–d:

$$\begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ b_0 & b_1 & b_2 & b_3 \\ c_0 & c_1 & c_2 & c_3 \\ d_0 & d_1 & d_2 & d_3 \end{bmatrix} \rightarrow \begin{bmatrix} a_0 & b_0 & c_0 & d_0 \\ a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \end{bmatrix}$$

Reduction Operations

- `shmem_max_reduce` — find the largest value across all team members
- `shmem_min_reduce` — find the smallest value across all team members
- `shmem_sum_reduce` — find the sum of all team members' values
- `shmem_prod_reduce` — find the product of all team members' values
- `shmem_and_reduce` — perform a bitwise and across all team members' values
- `shmem_or_reduce` — perform a bitwise or across all team members' values
- `shmem_xor_reduce` — perform a bitwise xor across all team members' values

Tired Yet?

- We're "done" with collectives on teams now, but we must address the "old" collective API we mentioned
- Prior to 1.5, teams did not exist, and collectives were insted performed ad hoc on implicitly defined "active sets"
- Active sets are simply a triplet of numbers describing a starting PE, a *logarithmic* (base 2) stride, and size
  - For instance, an active set starting at PE 7, with a logarithmic stride of 2 (i.e., $2^2 = 4$), and a size of 4 would describe the following set of PEs: 7, 11, 15, and 19
- Many implementations still don't support the 1.5 specification, and even more applications exist that have yet to be updated to the new features from it, so you *will* see these come up

Preparing for Active Set Collectives

- Active set collectives additionally requires the internal memory they use to be allocated and provided by the application (so-called “pSync” and “pWrk” arrays)
- These arrays must be allocated via `shmem_malloc` according to `SHMEM_SYNC_SIZE` or `SHMEM_REDUCE_MIN_WRKDATA_SIZE`, as appropriate
- Before first use, all elements must be set to `SHMEM_SYNC_VALUE` and then *never modified again*

Demo Time Again!

Time for a Break!

We will resume in 15 minutes!

OAK RIDGE National Laboratory

June 20, 2025

Module 5: Extra Topics

Aaron Welch[1], Oscar Hernandez[1]

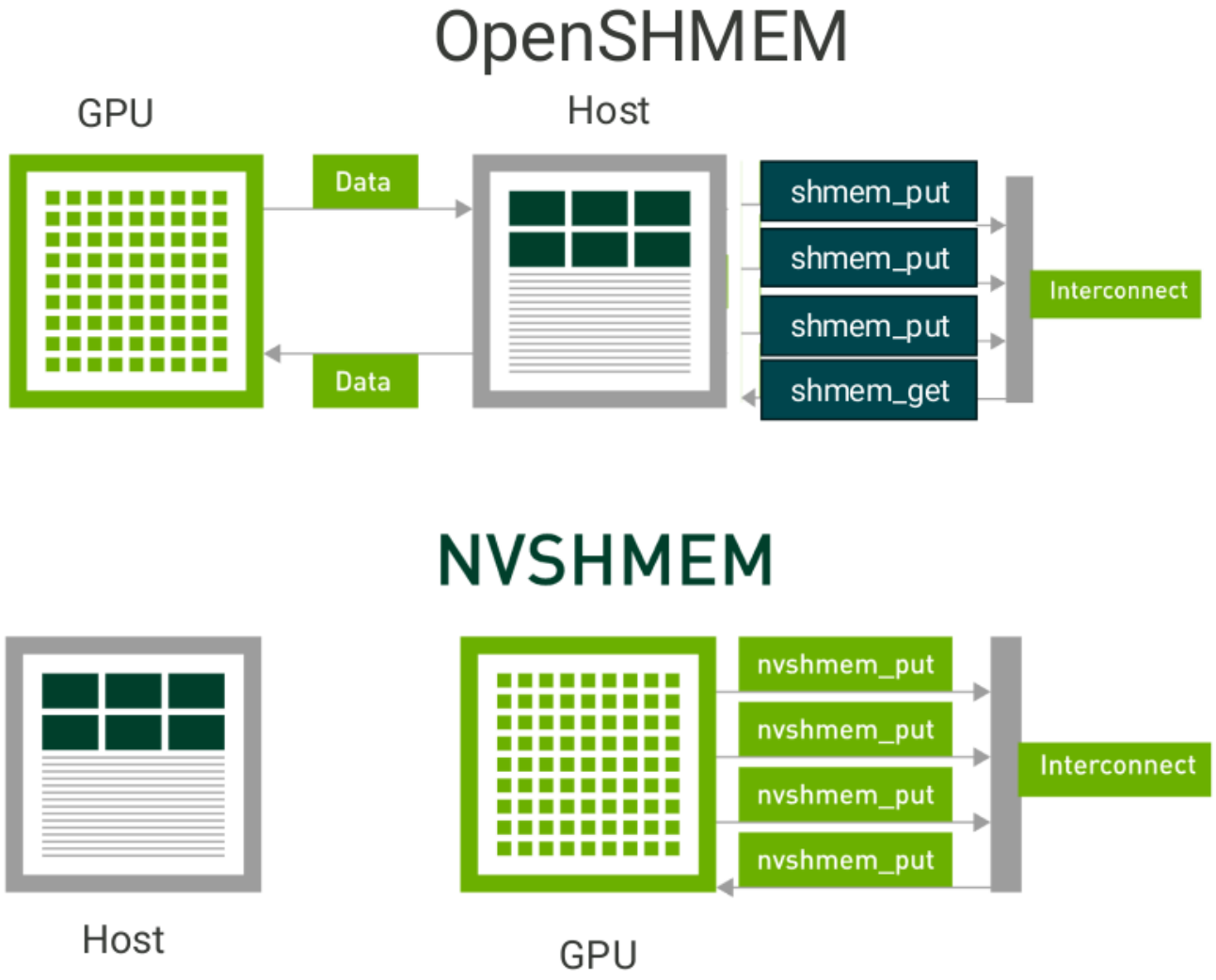
[1]Oak Ridge National Laboratory
oscar@ornl.gov, welchda@ornl.gov

U.S. DEPARTMENT OF ENERGY

ORNL IS MANAGED BY UT-BATELLE LLC FOR THE DEPARTMENT OF ENERGY

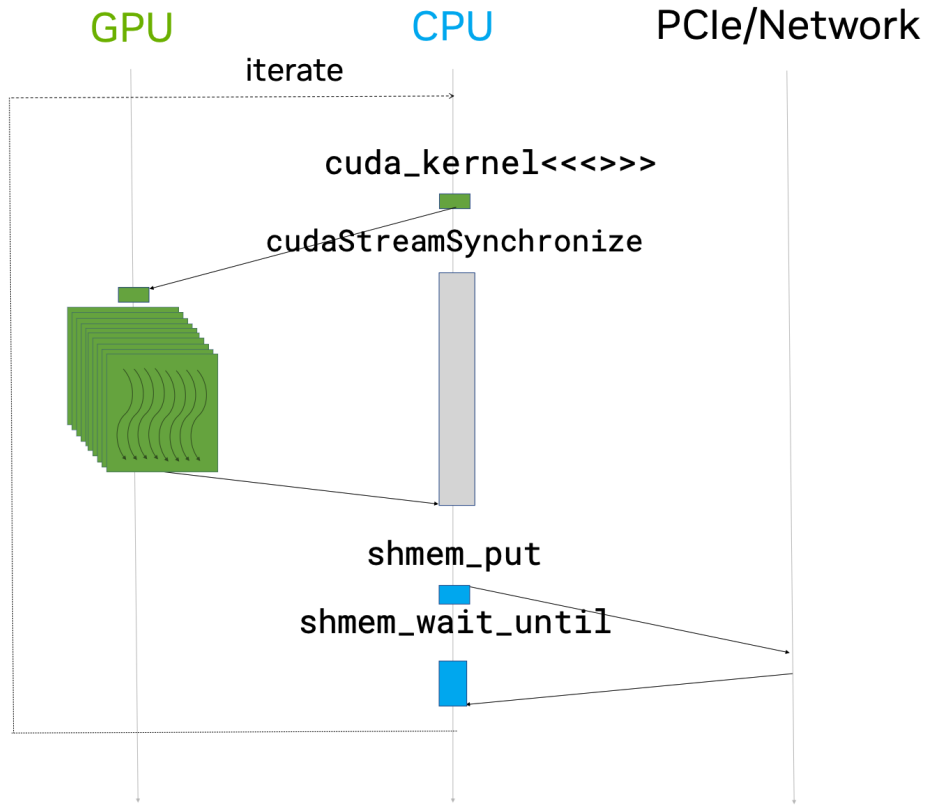
NVSHMEM – OpenSHMEM for NVIDIA GPUs

- Key features:
  - Combines the memory of multiple GPUs into a PGAS that's accessed through NVSHMEM APIs
  - Includes a low-overhead, in-kernel communication API for use by GPU threads

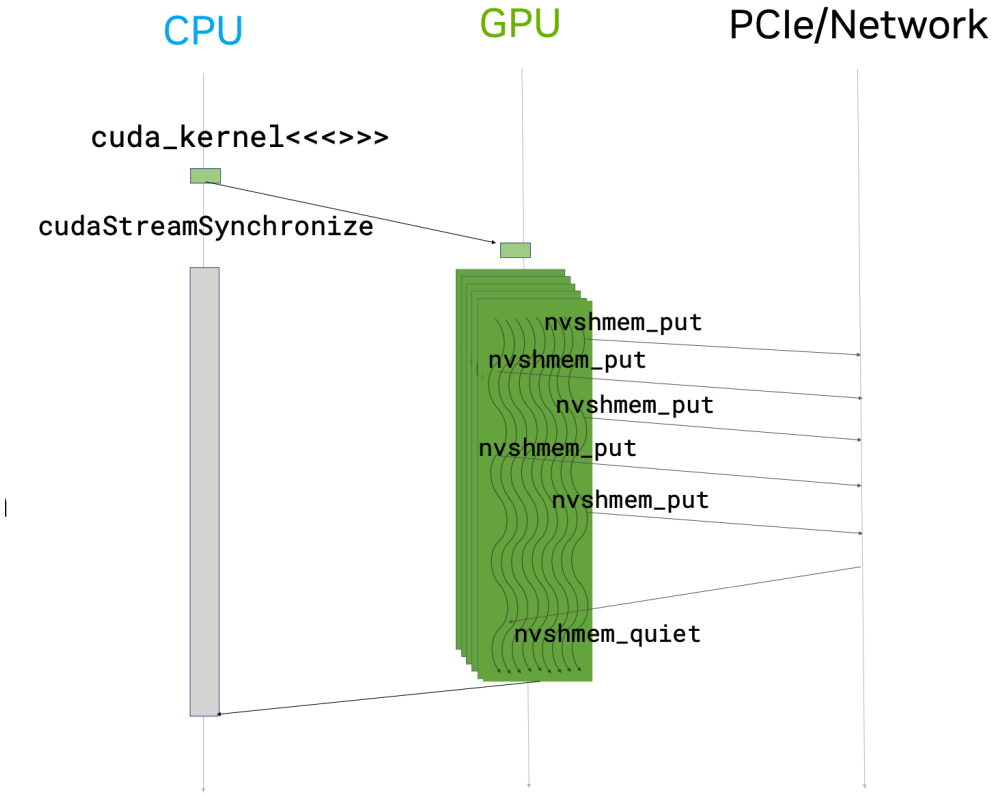


https://docs.nvidia.com/nvshmem/index.html

OpenSHMEM vs NVSHMEM



OpenSHMEM[1]



NVSHMEM[1]

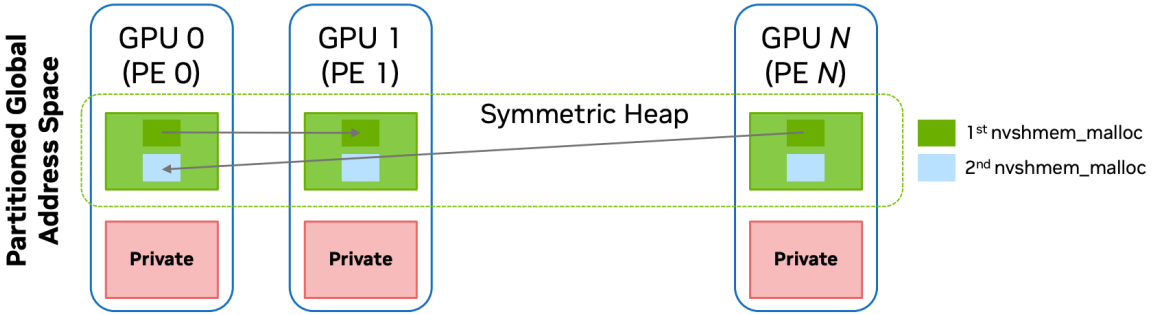
- Compute and communicate on the GPU
- Eliminates offloading latencies

- Latencies hidden by GPU threading
- Computation/communication overlap

[1]Source: https://juser.fz-juelich.de/record/1019178/files/02-NCCL_NVSHMEM.pdf

NVSHMEM Example

- One PE per GPU
- Symmetric heap allocates GPU memory
- More information:
  - https://juser.fz-juelich.de/record/1019178/files/02-NCCL_NVSHMEM.pdf
  - https://docs.nvidia.com/nvshmem/index.html




```c
int main(int argc, char **argv)  {
    int mype_node, msg;
    cudaStream_t stream;
    nvshmem_init();
    mype_node = nvshmem_team_my_pe(NVSHMEMX_TEAM_NODE);
    cudaSetDevice(mype_node);
    cudaStreamCreate(&stream);
    int *destination = (int *) nvshmem_malloc(sizeof(int));
    simple_shift<<<1, 1, 0, stream>>>(destination);
    nvshmemx_barrier_all_on_stream(stream);
    cudaMemcpyAsync(&msg, destination, sizeof(int),
       cudaMemcpyDeviceToHost, stream);
    cudaStreamSynchronize(stream);
    printf("%d: received message %d\n", nvshmem_my_pe(), msg);
    nvshmem_free(destination);
    nvshmem_finalize();
    return 0;
}
```

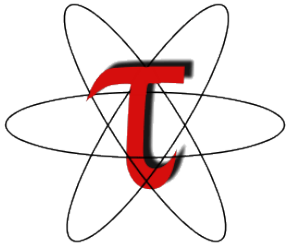
Host

```c
#include <stdio.h>
#include <cuda.h>
#include <nvshmem.h>
#include <nvshmemx.h>
__global__ void simple_shift(int *destination) {
    int mype = nvshmem_my_pe();
    int npes = nvshmem_n_pes();
    int peer = (mype + 1) % npes;
    nvshmem_int_p(destination, mype, peer);
}
```

GPU

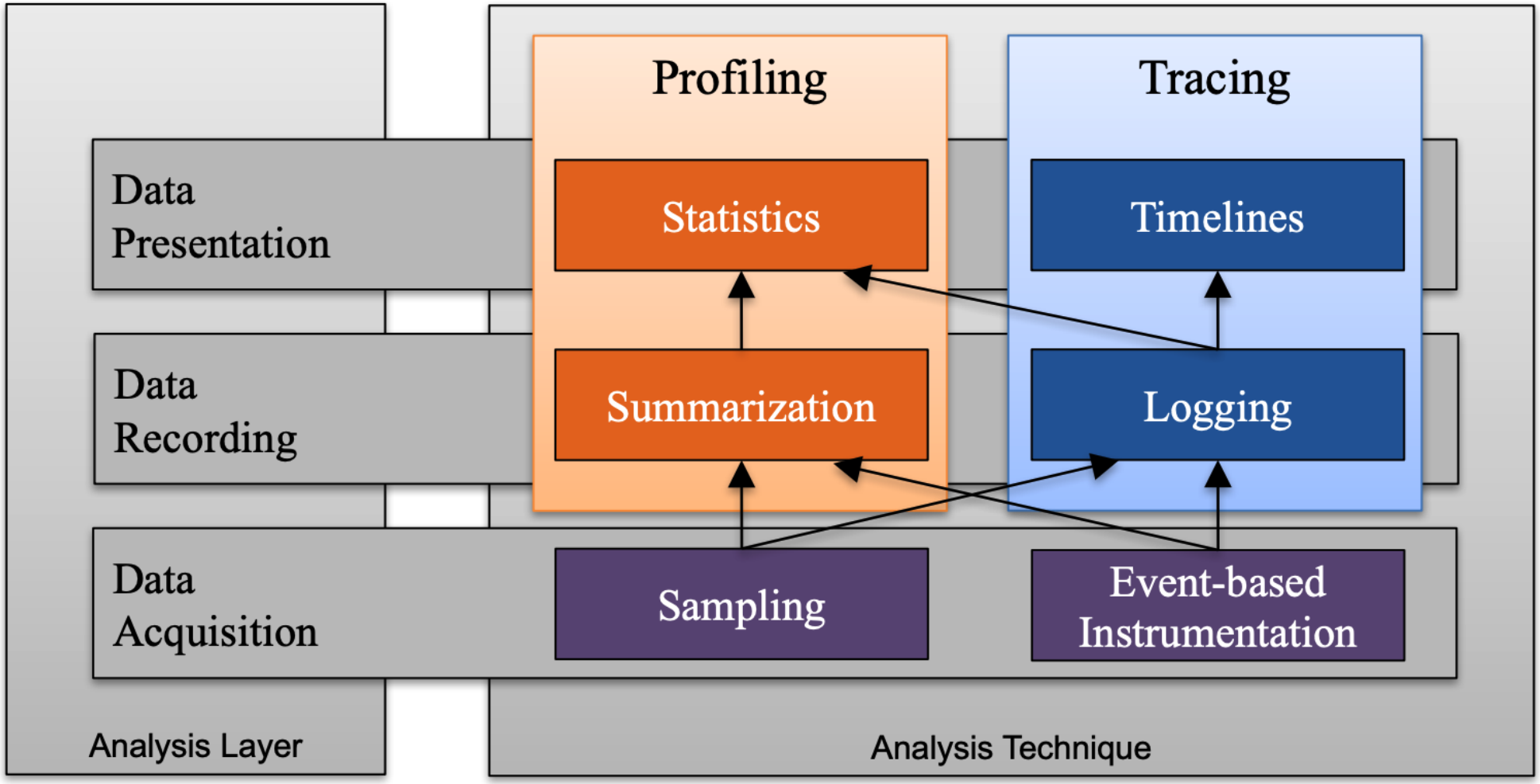
OpenSHMEM Tools

Performance Tools



Debuggers

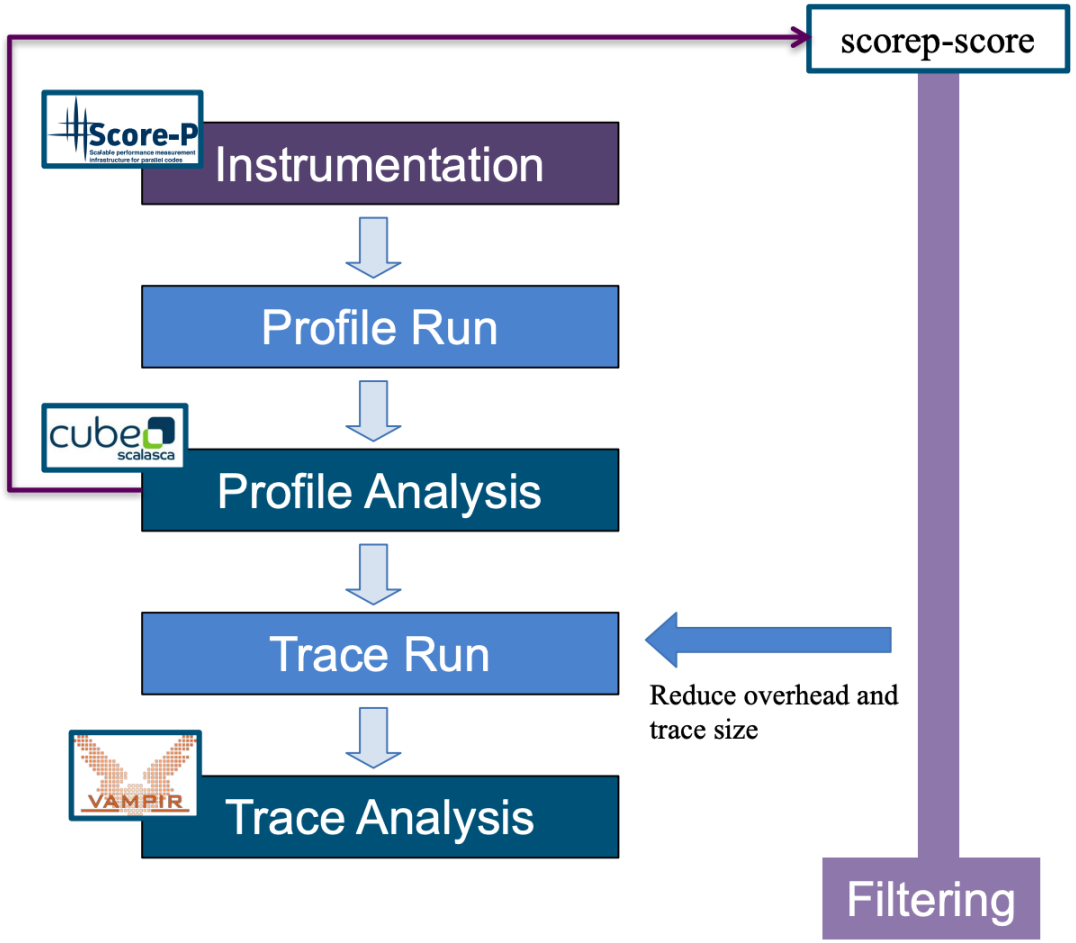
Performance Tools



Profiling vs Tracing

Performance Tools

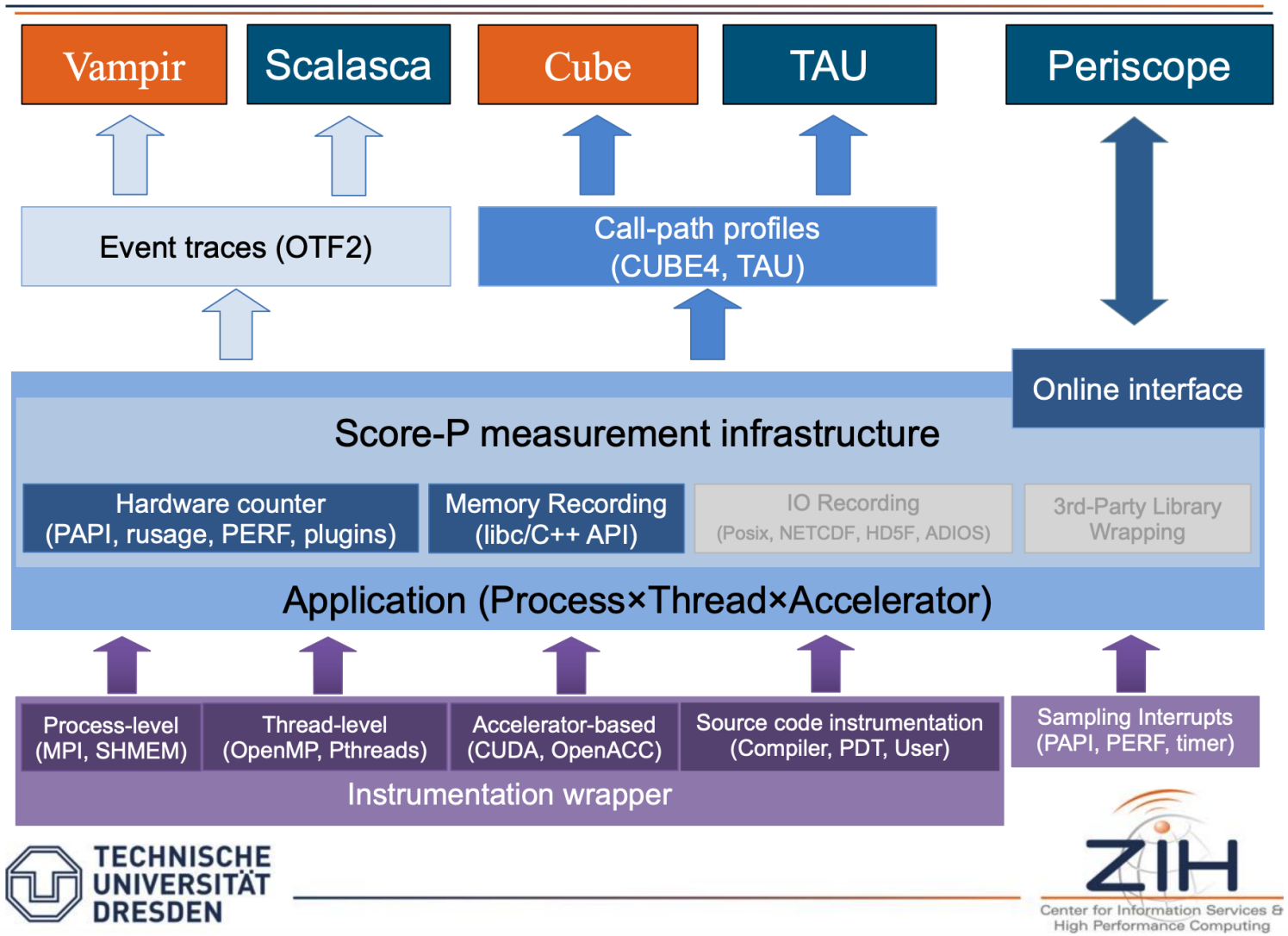
- Goal: understand where the inefficiencies are in your code with regard to runtime, hw utilization, wait times, etc.
- Different types of tools:
  - Direct measurement
  - Sampling
  - Monitoring



Score-P workflow: profiling + tracing

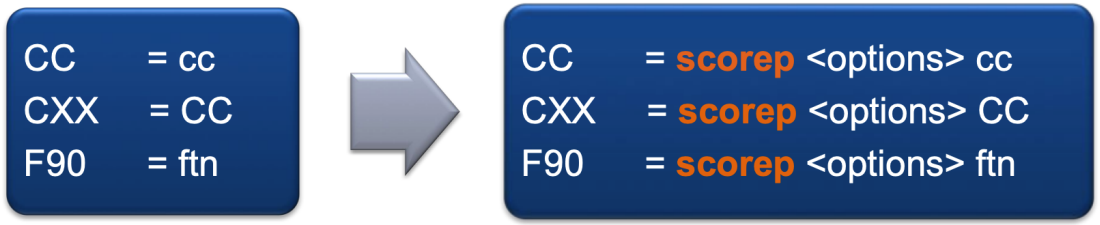
Performance Tools

- Open-source performance tools ecosystem
- Score-P measurement infrastructure



https://www.vi-hps.org/projects/score-p/overview/overview.html

Score-P Usage



Disable instrumentation

```
#CMake
SCOREP_WRAPPER=OFF cmake .. \
   -DCMAKE_C_COMPILER=scorep-icc \
   -DCMAKE_CXX_COMPILER=scorep-icpc \
   -DCMAKE_Fortran_COMPILER=scorep-ifc
```

```
#Autotools
SCOREP_WRAPPER=OFF ../configure \
     CC=scorep-icc \
     CXX=scorep-icpc \
     FC=scorep-ifc \
     --disable-dependency-tracking
```

- Pass instrumentation and compiler flags at make

```
make SCOREP_WRAPPER_INSTRUMENTER_FLAGS="--user" \
     SCOREP_WRAPPER_COMPILER_FLAGS="-g -O2"
```

scorep --user <your_compiler> –g –O2

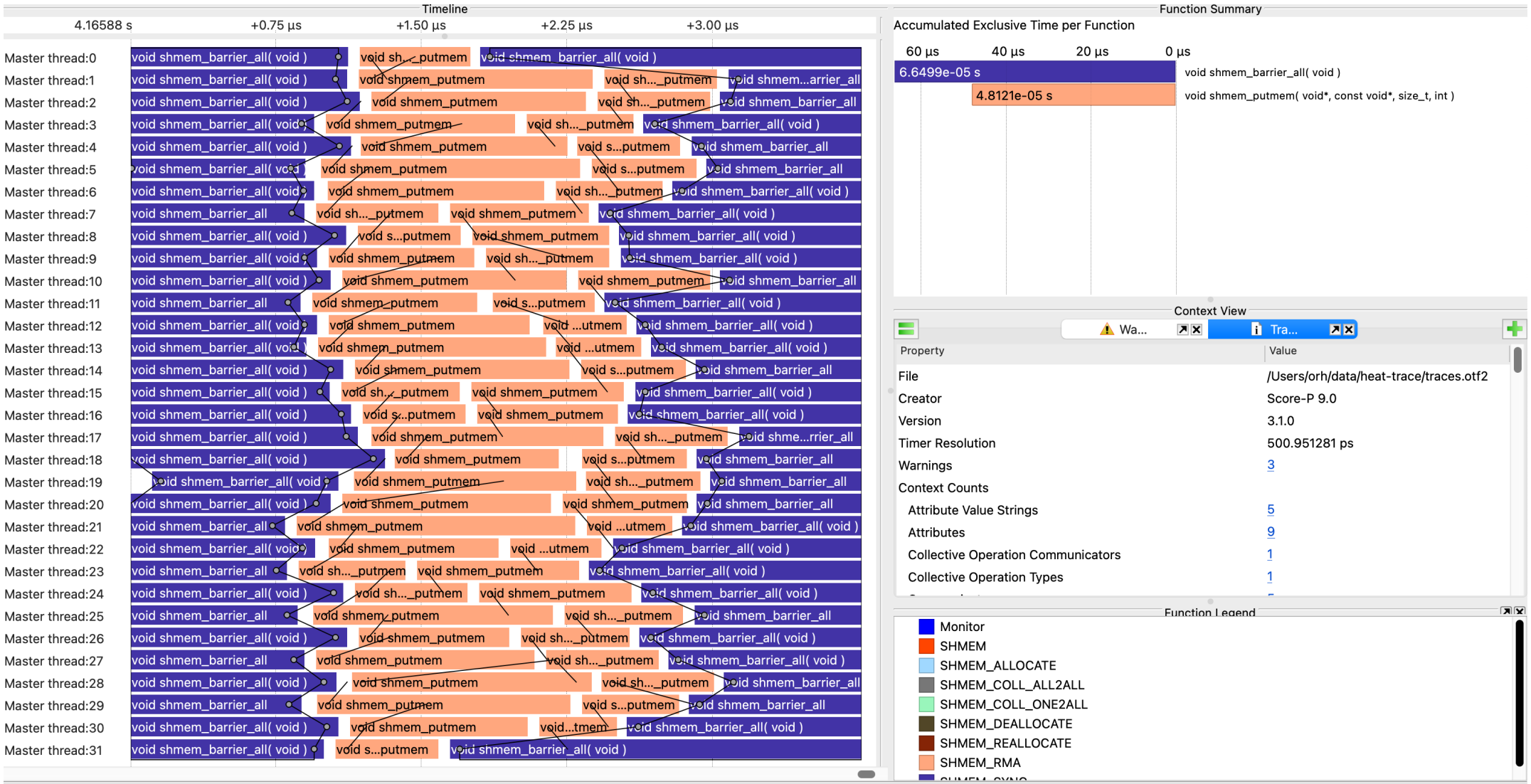
Compiling:

```
scorep cc heat.c -o heat
```

Running:

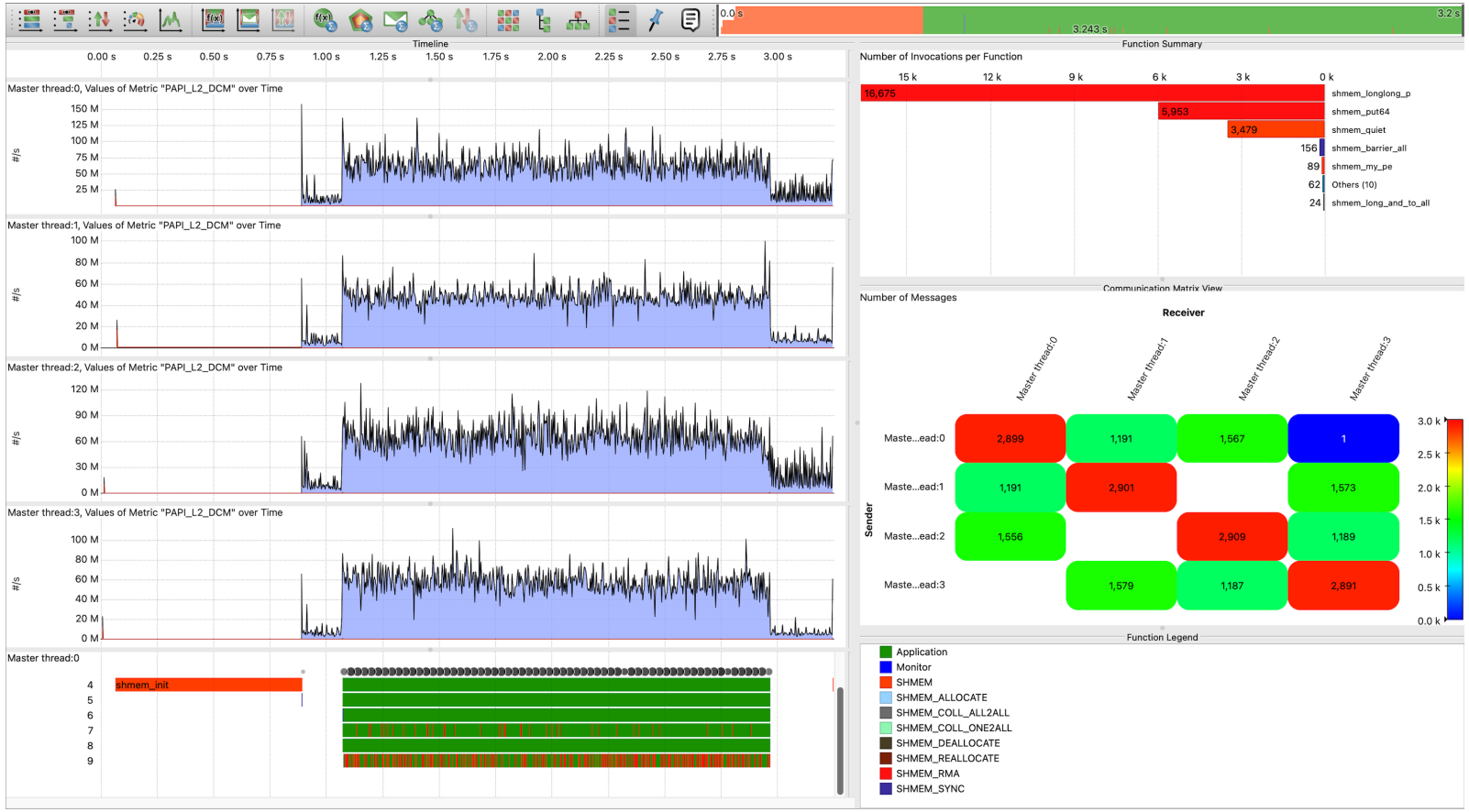
```
export SCOREP_ENABLE_PROFILING=false
export SCOREP_ENABLE_TRACING=true
export SCOREP_EXPERIMENT_DIRECTORY=heat-trace
srun -n 32 ./heat
```

Score-P Trace Visualization with Vampir



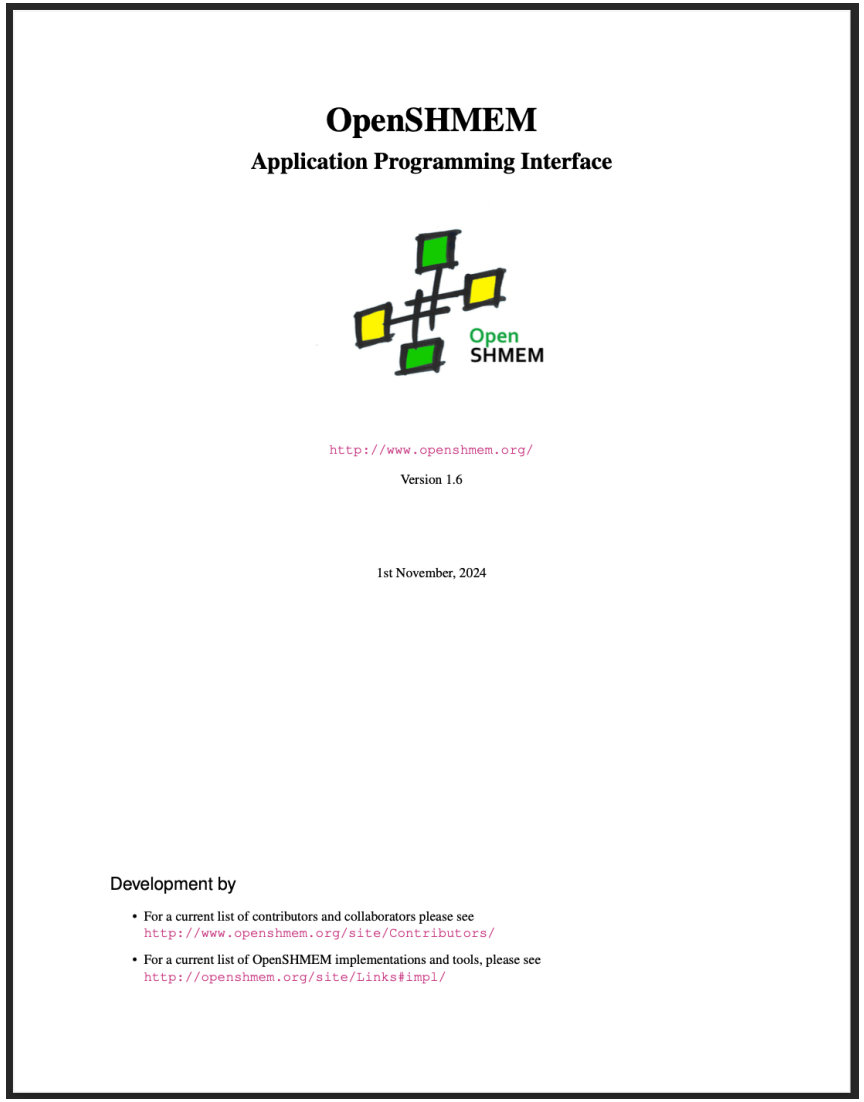
Tools — Tracing OpenSHMEM

- Timeline trace
- Per PE
- Callstack view
- Function summaries
- Communication matrix view
- Tracing with hardware counters



OpenSHMEM v1.6 New Features

- Interleaved block put/get routines
- Scan collectives
- Sessions
- Signal add and set
- PE-specific quiet
- Multiple init/finalize
- Errata section
- No new deprecations



OpenSHMEM

Application Programming Interface

Open SHMEM

http://www.openshmem.org/

Version 1.6

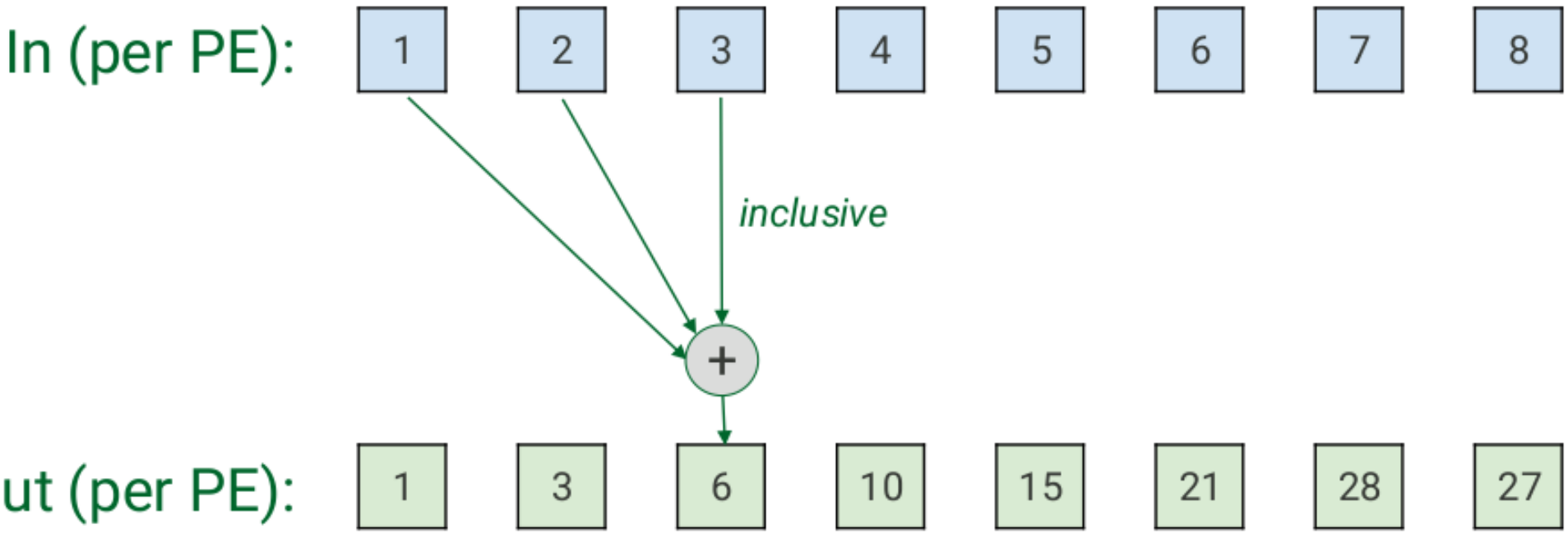
1st November, 2024

Development by

- For a current list of contributors and collaborators please see http://www.openshmem.org/site/Contributors/
- For a current list of OpenSHMEM implementations and tools, please see http://openshmem.org/site/Links#impl/

https://github.com/openshmem-org/specification/releases/tag/v1.6

Scan a.k.a. Prefix Sum



- Inclusive and exclusive prefix sum:
  - shmem_sum_inscan
  - shmem_sum_exscan
- Example use case: calculate offset into a shared buffe (in: local data size, out: offset)

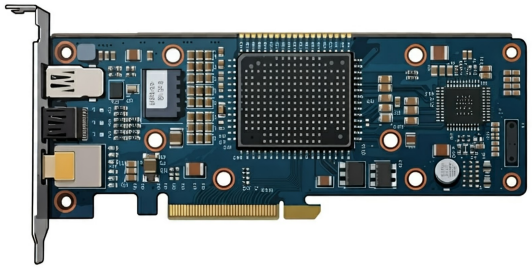
OpenSHMEM Sessions

```
shmem_ctx_session_start(ctx,
    SHMEM_CTX_SESSION_BATCH, NULL, 0);

for (i = 0; i < npes; i++)
    shmem_put(dest, src, size, i);

shmem_ctx_stop(ctx);
```

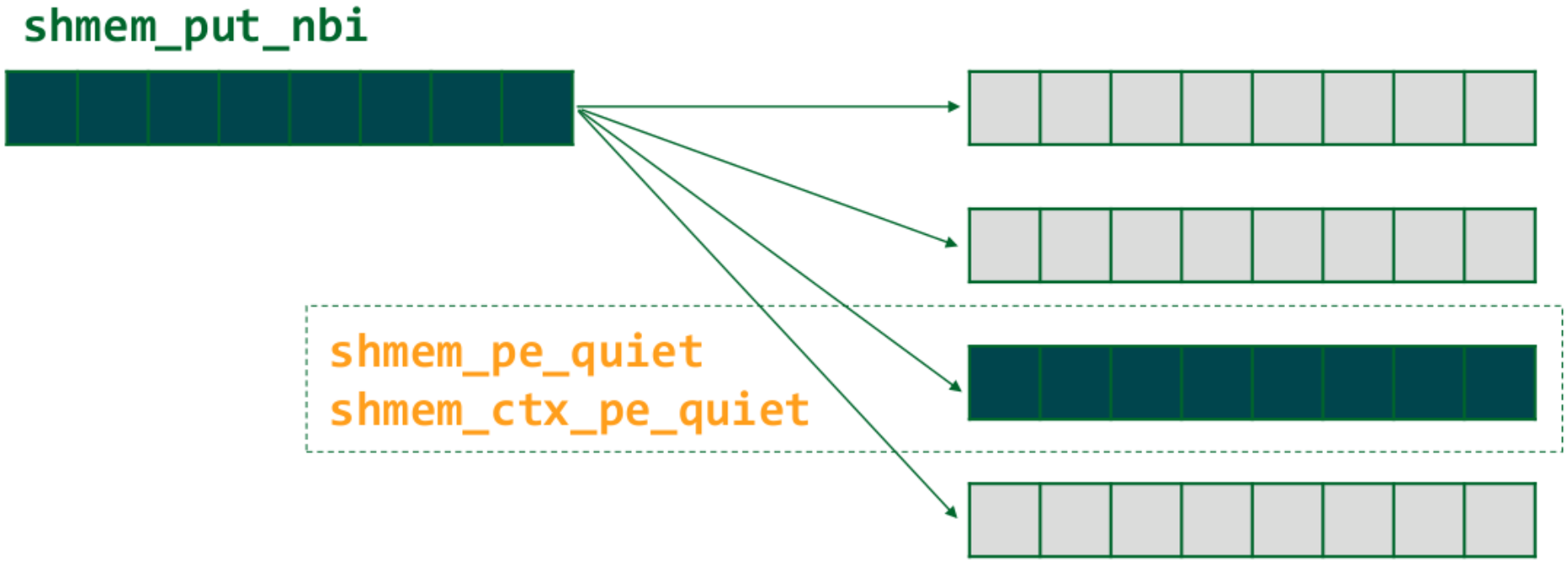
Batches

```
shmem_put(dest, src, size, 0);
shmem_put(dest, src, size, 1);
shmem_put(dest, src, size, 2);
shmem_put(dest, src, size, 3);
```

- Example use case: enable batched submission of requests to the NIC



Per-PE Quiet



- Example use case: complete non-blocking puts to a specific peer

Multiple `init` and `finalize` Calls

```
// Phase 1
shmem_init_thread(...);
library_using_shmem(...);
shmem_finalize();

// Phase 2
shmem_init_thread(...);
…
shmem_finalize(...);
```

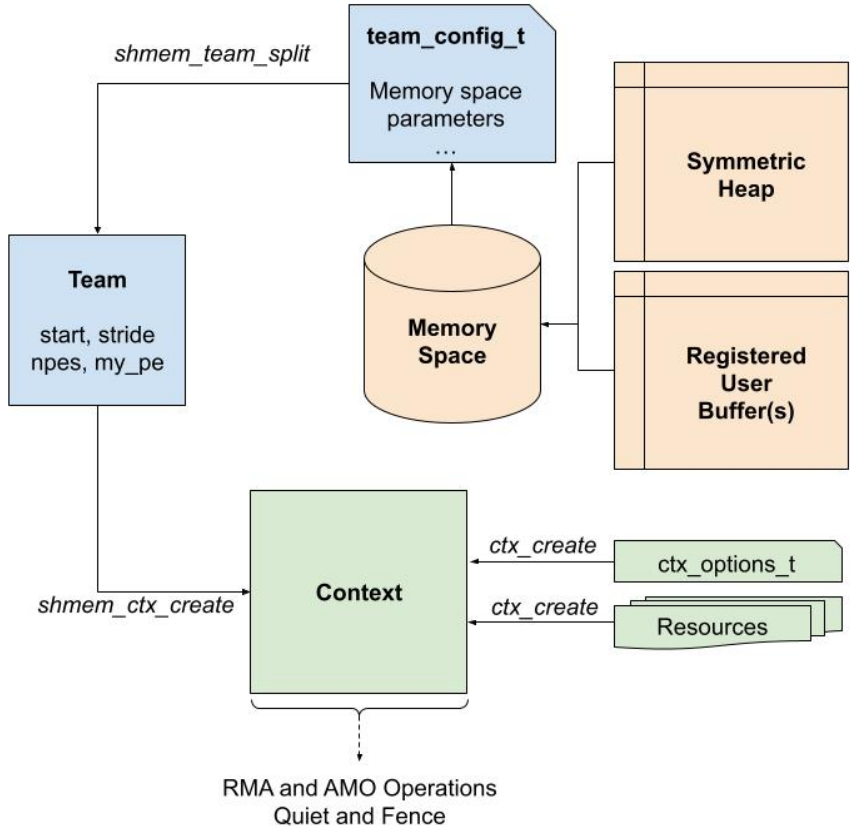


```
shmem_init_thread(...);
// Perform Communication
shmem_finalize();
```

- Example use case: interoperability, separation of concerns, library support, better software engineering

OpenSHMEM 1.7

- Non-blocking collectives
- Memory spaces
  - Multiple symmetric heaps
  - Symmetric memory registration
  - Memory kinds: GPU memory / high bandwidth memory (HBM) / Fabric-Attached Memory (FAM)
- Memory model clarifications
  - Atomics ordering
  - Interactions with C memory model
    - Sequential consistency, release/acquire, relaxed
- GPU SHMEM interface
  - NVSHMEM, ROC_SHMEM, and Intel SHMEM
  - How will this work without a common programming model?



Let’s Look at Those Practical Exercises!

Thank You

How to get involved:

- Website: http://openshmem.org/
- Github: https://github.com/openshmem-org/specification/
- Slack: email jdinan@nvidia.com
- Mailing list: http://openshmem.org/mailman/listinfo/openshmem-list/

Tutorial contacts:
Aaron Welch: welchda@ornl.gov
Oscar Hernandez: oscar@ornl.gov