# Mochi Tutorial

## Building Efficient Distributed Microservices for Exascale


ROB ROSS, PHILIP CARNS, MATTHIEU DORIER, KEVIN HARMS, ROB LATHAM, AND SHANE SNYDER — Argonne National Laboratory

GARTH GIBSON, GEORGE AMVROSIADIS, CHUCK CRANOR, SAURABH KADEKODI, AND QING ZHENG — Carnegie Mellon University

JEROME SOUMAGNE AND JOE LEE — The HDF Group

GALEN SHIPMAN AND BRAD SETTLEMYER — Los Alamos National Laboratory

# Outline of the tutorial

- Introduction & Motivation
- Examples of Mochi-based services
- Mercury tutorial
- Argobots tutorial
- Margo tutorial
- Hands-on exercises (optional)
- Other uses of Mercury and Argobots

# I. Introduction

What is the Mochi project?

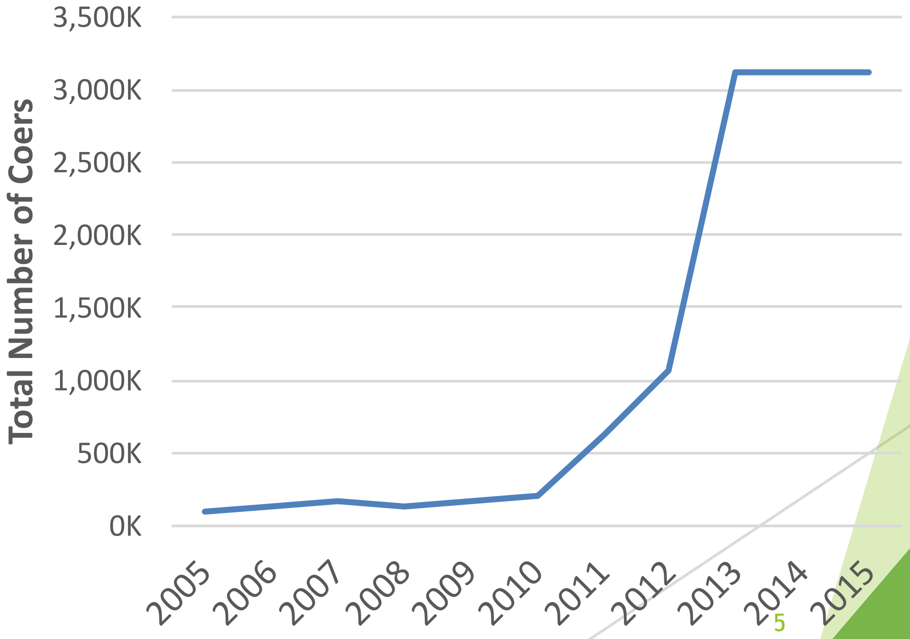
# Challenges of Future Exascale Systems

- Heterogeneous, massively parallel compute nodes
- High performance, complex network topologies
- Tens of thousands of compute nodes to manage
- Constrained resources (power, I/O)


# Top500 Supercomputers Today

1. Tianhe-2 (Intel Xeon + Xeon Phi): 3,120,000 cores
2. Titan (Cray XK7 + Nvidia K20x): 560,640 cores
3. Sequoia (BlueGene/Q): 1,572,864 cores
4. K computer (SPARC64): 705,024 cores
5. Mira (BlueGene/Q): 786,432 cores

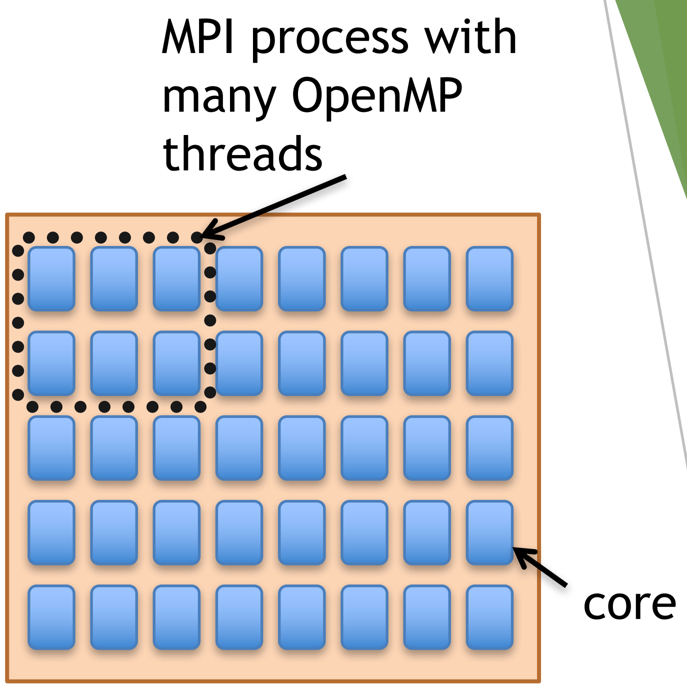

Number of cores per socket in Top500




Total number of cores in Top500 rank #1

# Massive On-node Parallelism

- To address massive on-node parallelism, the number of work units (e.g., threads) must increase by 100X
- MPI+OpenMP is sufficient for many apps, but implementation is poor
  - Today MPI+OpenMP == MPI+Pthreads
- Pthread abstraction is too generic, not suitable for HPC
  - Lack of fine-grained scheduling, memory management, network management, signaling, etc.
- Better runtime can significantly improve MPI+OpenMP performance and support other emerging programming models




Current situation:

- One or more MPI processes per node
- Each MPI process has limited internal parallelism typically with OpenMP
- MPI Process communication is often serialized

# More storage/memory layers…

## HPC Before 2016

| | |
|---|---|
| DRAM | Memory |
| Lustre Parallel File System | Parallel File System |
| HPSS Parallel Tape | Archive |

## HPC After 2016

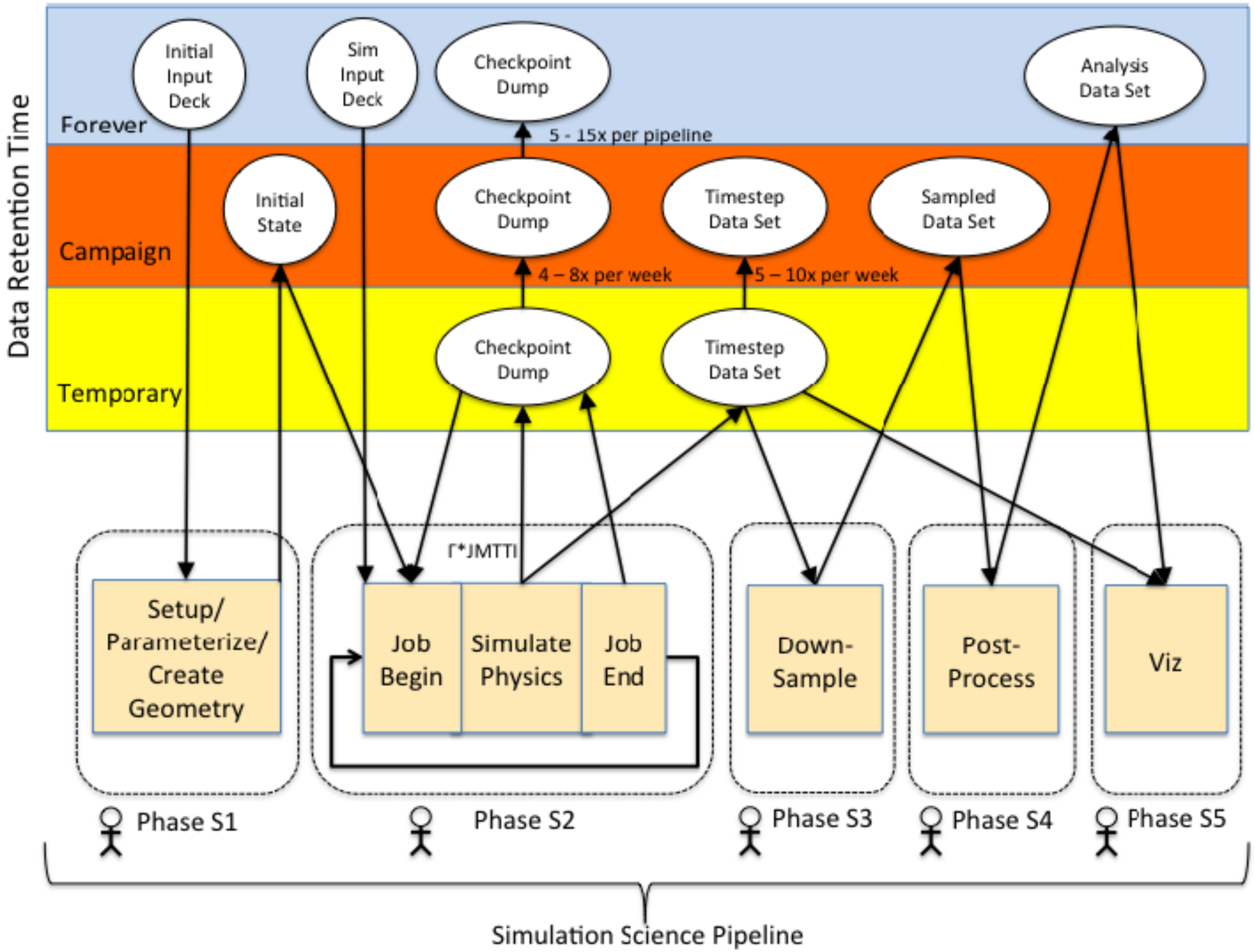
| | |
|---|---|
| Memory | 1-2 PB/sec<br>Residence - hours<br>Overwritten - continuous |
| Burst Buffer | 4-6 TB/sec<br>Residence - hours<br>Overwritten - hours |
| Parallel File System | 1-2 TB/sec<br>Residence - days/weeks<br>Flushed - weeks |
| Campaign Storage | 100-300 GB/sec<br>Residence - months-year<br>Flushed - months-year |
| Archive | 10s GB/sec (parallel tape<br>Residence - forever |

- Why
  - BB: Economics (disk bw/iops too expensive)
  - PFS: Maturity and BB capacity too small
  - Campaign: Economics (tape bw too expensive)
  - Archive: Maturity and we really do need a "forever"

Slide from Gary Grider (LANL)

APEX Workflows, LANL, NERSC, SNL, SAND2015-10342 O, LA-UR-15-29113

# Simulation workflow



# specialization of Data services

Application

**Executables and Libraries**

SPINDLE

**Checkpoints**

SCR

FTI

**Intermediate Data Products**

DataSpaces

Kelpie

MDHIM

| | Provisioning | Comm. | Local Storage | Fault Mgmt. and Group Membership | Security |
|---|---|---|---|---|---|
| **ADLB**<br>*Data store and pub/sub.* | MPI ranks | MPI | RAM | N/A | N/A |
| **DataSpaces**<br>*Data store and pub/sub.* | Indep. job | Dart | RAM (SSD) | Under devel. | N/A |
| **DataWarp**<br>*Burst Buffer mgmt.* | Admin./ sched. | DVS/ lnet | XFS, SSD | Ext. monitor | Kernel, lnet |
| **FTI**<br>*Checkpoint/restart mgmt.* | MPI ranks | MPI | RAM, SSD | N/A | N/A |
| **Kelpie**<br>*Dist. in-mem. key/val store* | MPI ranks | Nessie | RAM (Object) | N/A | Obfusc. IDs |
| **SPINDLE**<br>*Exec. and library mgmt.* | Launch MON | TCP | RAMdisk | N/A | Shared secret |

We need service-oriented tools to program at Exascale!

# The Mochi project: Exascale services for Science

### Motivation and Approach

- Science teams have rich data service needs not satisfied by any single approach (e.g., file system, database)
- Approach is to enable data services to be rapidly <u>**built**</u> and <u>**composed**</u> to meet science needs
- Building blocks developed with HPC systems as target: fast and scalable

### Impact

- Building block components developed and in use by multiple teams
- Three technology demonstrations underway
  - Fast, remote objects backed to DRAM or nonvolatile storage
  - Computation caching for multi-scale simulations
  - Highly scalable metadata service enabling new organizations of output data under file model

# Terminology

- **Building blocks**
  - Set of libraries that have been designed to be compatible with one another
  - Examples: an RPC library, a threading library, a group membership library, etc.

- **Service**
  - A software component designed to help another software component in accomplishing a task
  - Examples: a file system, a key-val store, a distributed database, a computation cache, etc.

# Some of the Mochi building blocks: Mercury, Argobots, Margo

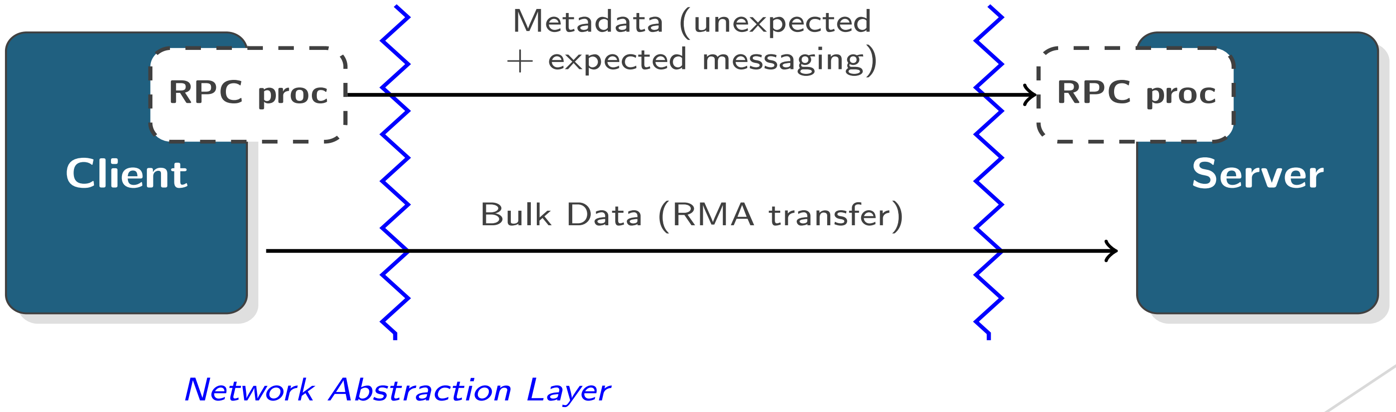
- **Mercury**
  - High performance RPC framework
  - Network abstraction, many transport methods supported
  - One-sided (RDMA) communication for large data transfers
- **Argobots**
  - Lightweight threading/tasking framework suited for massively multicore systems
  - Managing OS-level and user-level threads and tasks
- **Margo**
  - Bridge between Mercury and Argobots
  - Makes it REALLY easy to use both!

# Mercury
# A high performance RPC framework

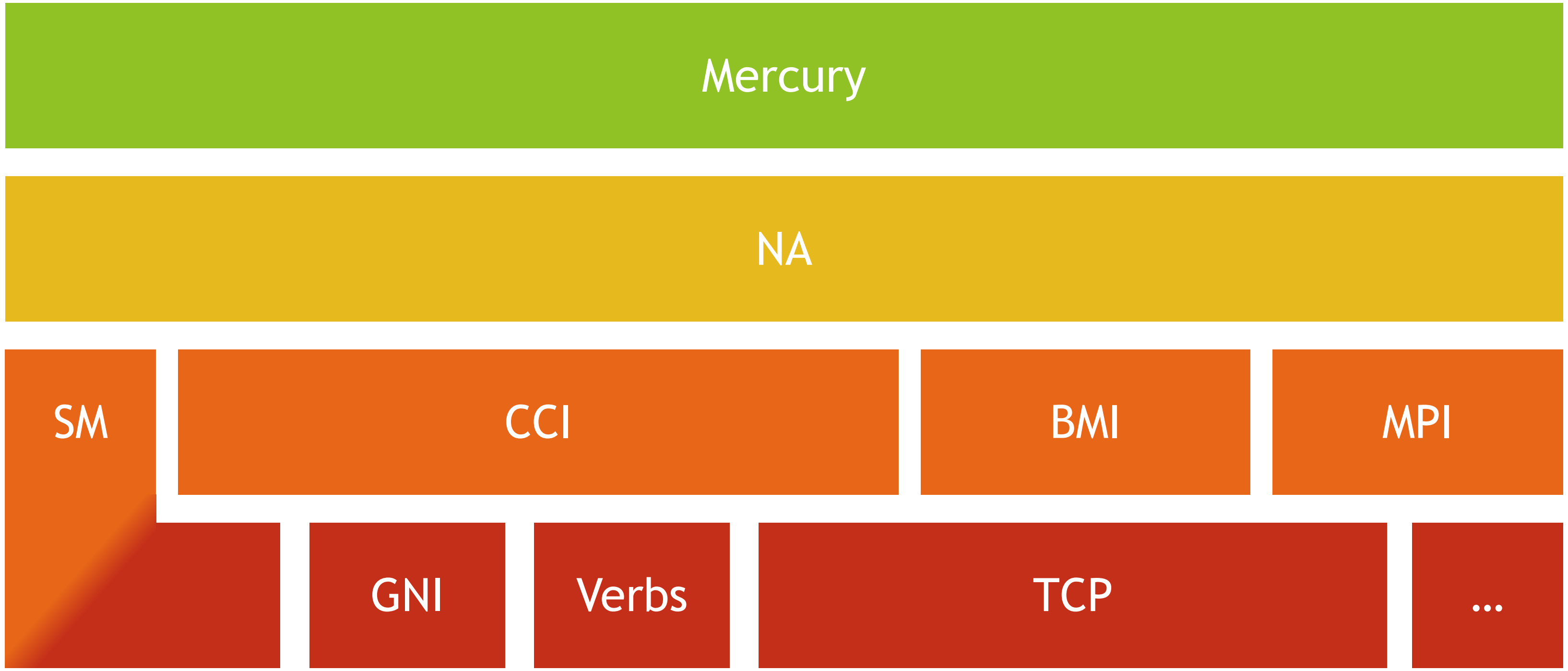
- https://mercury-hpc.github.io/

**Mercury is an RPC system for use in the development of high performance system services. Development is driven by the HDF Group with Argonne participation.**

- Portable across systems and network technologies
- Efficient bulk data movement to complement control messages
- Builds on lessons learned from IOFSL, Nessie, lnet, and others



# Mercury supports many network transport methods



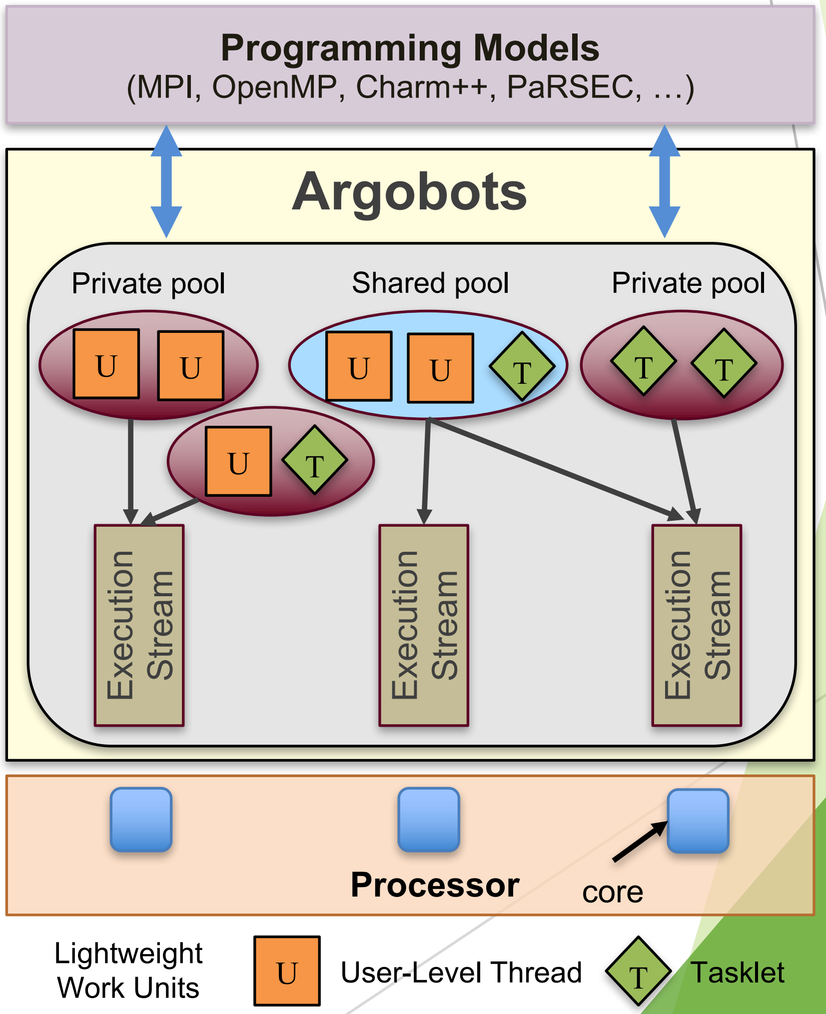
# Argobots
# A lightweight threading/tasking framework

## Overview

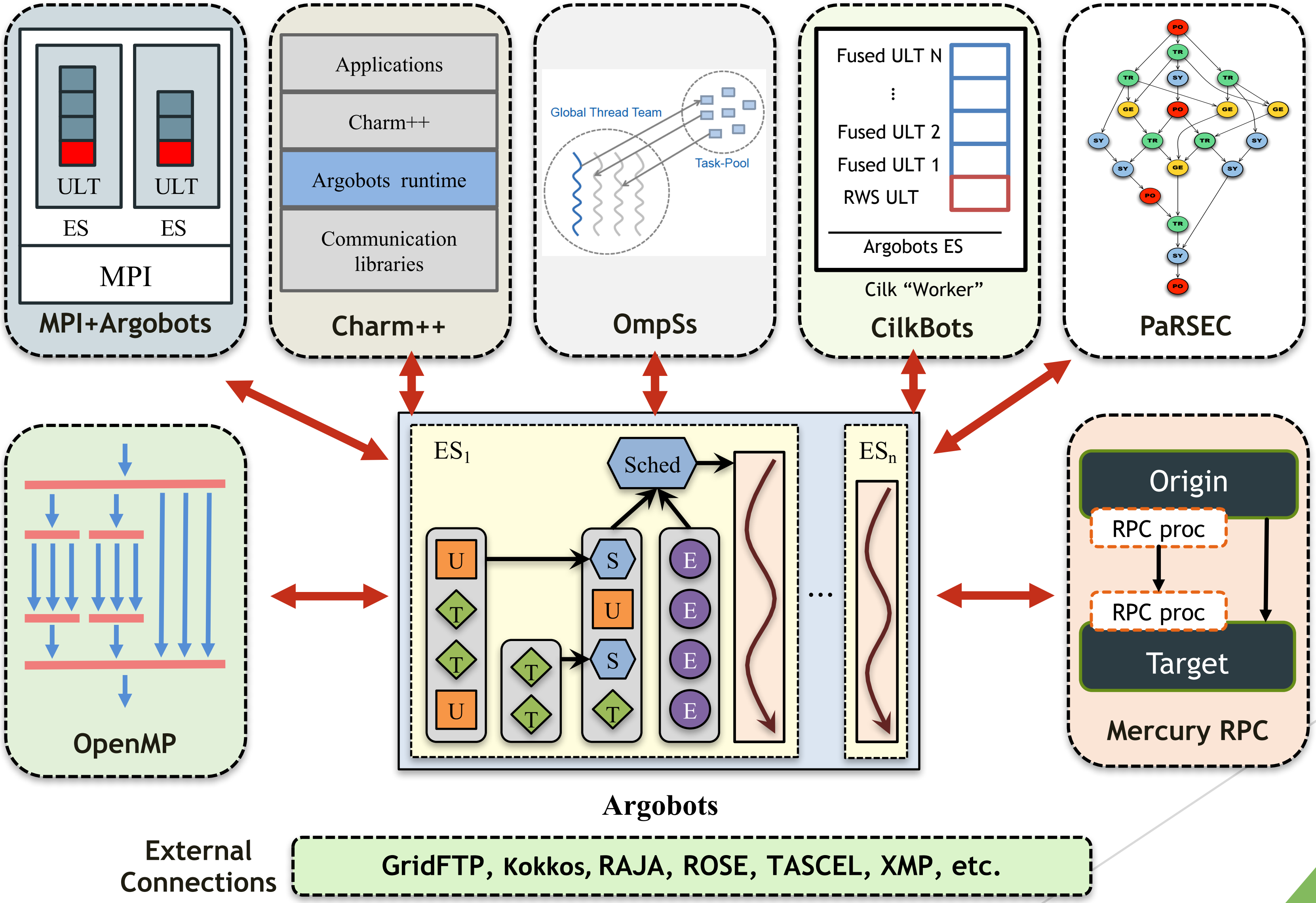
- Separation of mechanisms and policies
- Massive parallelism
  - **Exec. Streams** guarantee progress
  - **Work Units** execute to completion
    - User-level threads (ULTs) vs. Tasklet

## Argobots Innovations

- **Enabling technology, but not a policy maker**
  - High-level languages/libraries such as OpenMP, Charm++ have more information about the user application (data locality, dependencies)
- **Explicit model:**
  - Enables dynamism, but always managed by high-level systems



# Argobot’s ecosystem

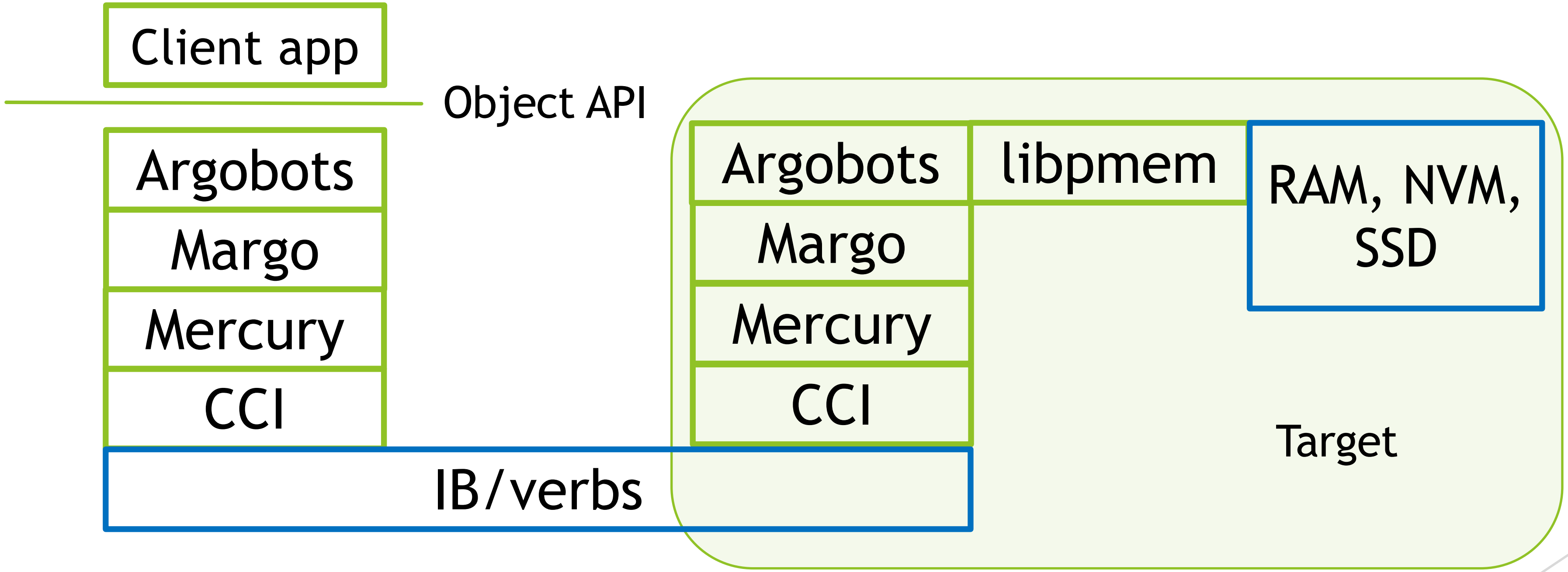


# II. Examples of Mochi-based services

The following examples are developed in the context of the DOE project

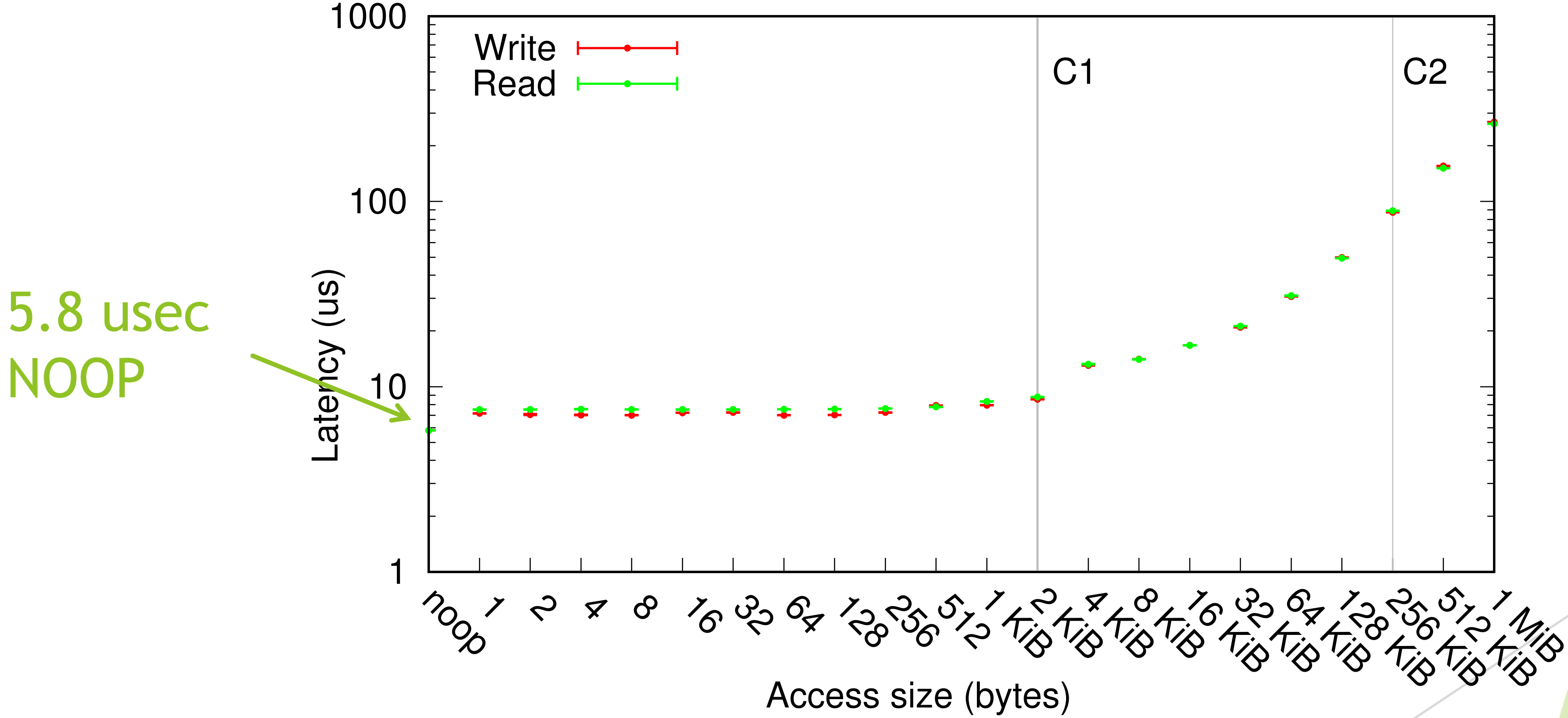
# 1. Remotely accessible objects

- API for remotely creating, reading, writing, destroying fixed-size objects/extents
- libpmem (http://pmem.io/nvml/libpmemobj/) for management of data on device



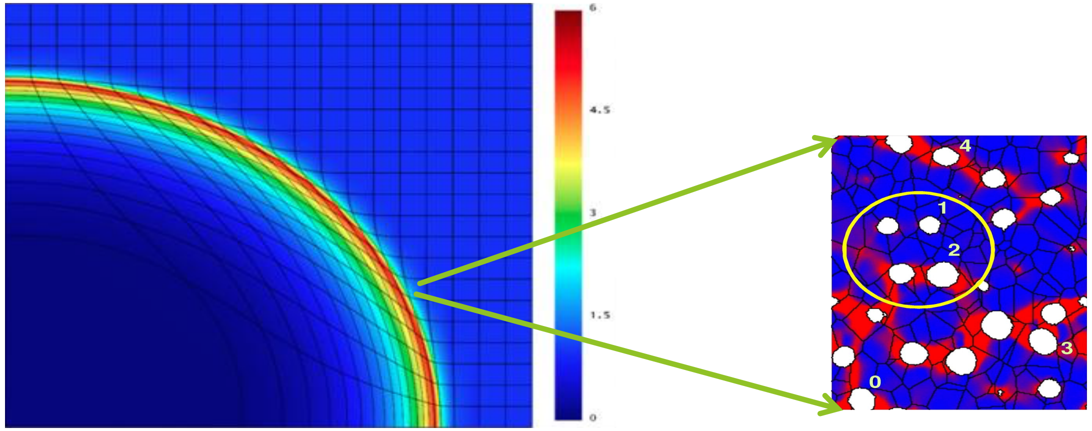

P. Carns et al. “Enabling NVM for Data-Intensive Scientific Services.” INFLOW 2016, November 2016.

# 1. Remotely accessible objects: How much latency in the stack?



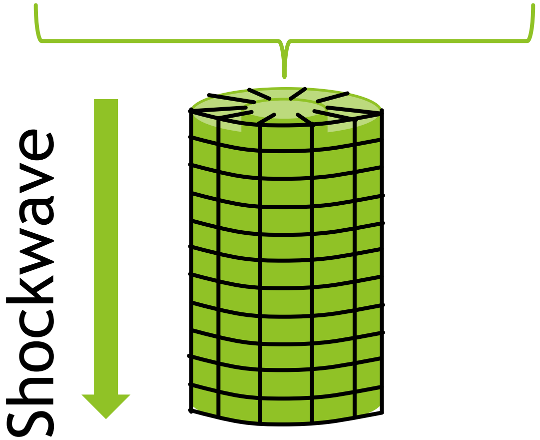

FDR IB, RAM disk, 2.6 usec round-trip (MPI) latency measured separately

# 2. Continuum model coupled with Viscoplasticity model




Lulesh continuum model:
- Lagrangian hydro dynamics
- Unstructured mesh

Viscoplasticity model [1]:
- FFT based PDE solver
- Structured sub-mesh

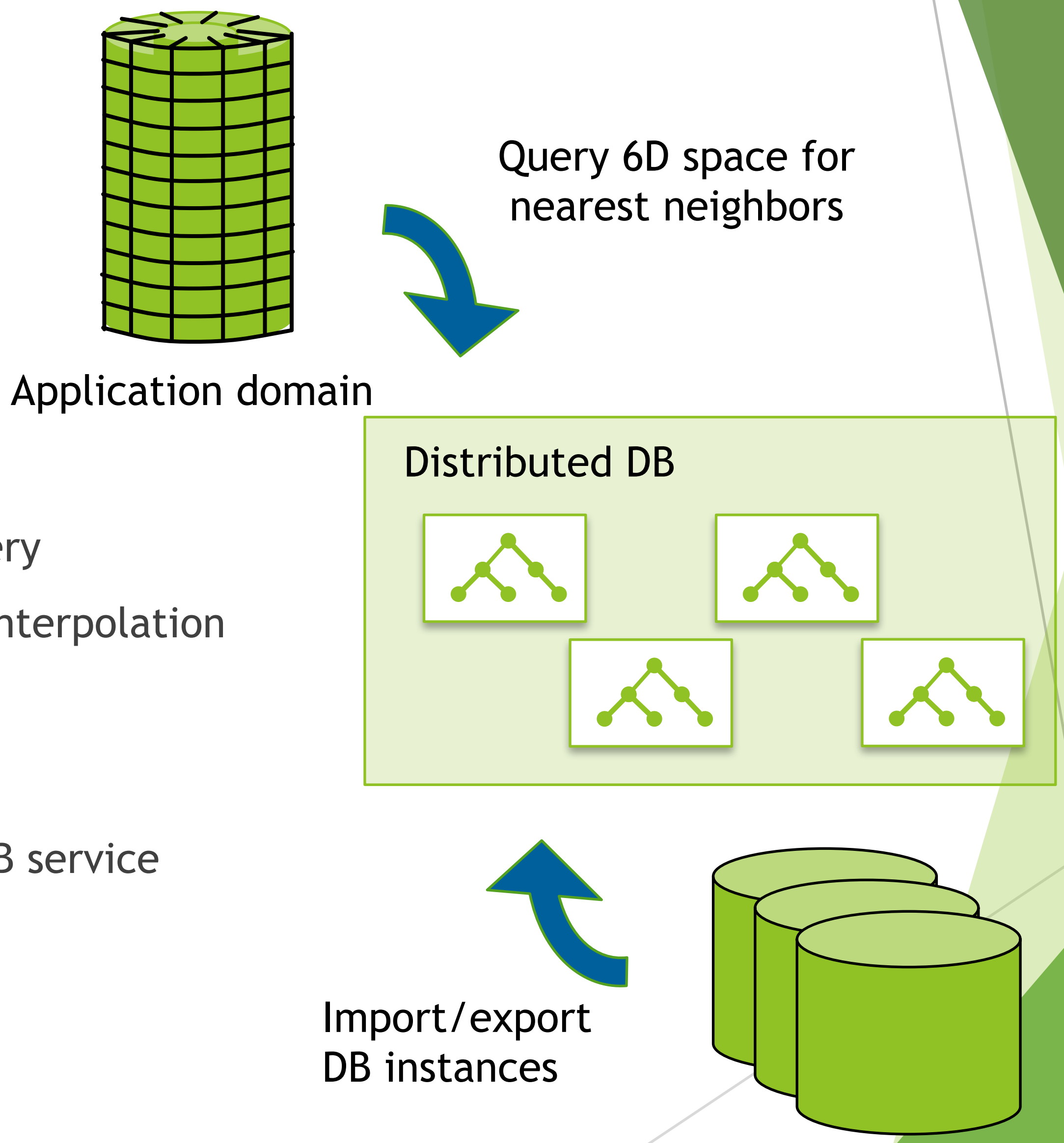



- Future applications are exploring the use of multi-scale modeling
- As an example: Loosely coupling continuum scale models with more realistic constitutive/response properties
    - e.g., Lulesh from ExMatEx
- Fine scale model results can be cached and new values interpolated from similar prior model calculations

R. Lebensohn et al, Modeling void growth in polycrystalline materials, Acta Materialia, http://dx.doi.org/10.1016/j.actamat.2013.08.004.

# 2. fine scale model Database

- Goals
  - Minimize fine scale model executions
  - Minimize query/response time
  - Load balance DB distribution
- Approach
  - Start with a key/value store
  - Distributed approx. nearest-neighbor query
  - Data distributed to co-locate values for interpolation
  - Import/export to persistent store
- Status
  - Mercury-based, centralized in-memory DB service
  - Investigating distributed, incremental nearest-neighbor indexing



# Before we start diving into codes…

- Connect (ssh) to the machine(s) provided by your teacher
- The machine has been setup with an OS image including all the necessary libraries (installed in /usr, so you don't have to do anything to have your Makefiles find them)
  - git clone https://xgitlab.cels.anl.gov/sds/sds-examples.git
  - cd sds-examples
  - mkdir build
  - cd build
  - cmake ..
  - Make
- Throughout this tutorial, feel free to look at the codes and try them!
- **IMPORTANT**: if you are several user sharing a machine, change the port number used by your programs (they are generally hard-coded) and recompile

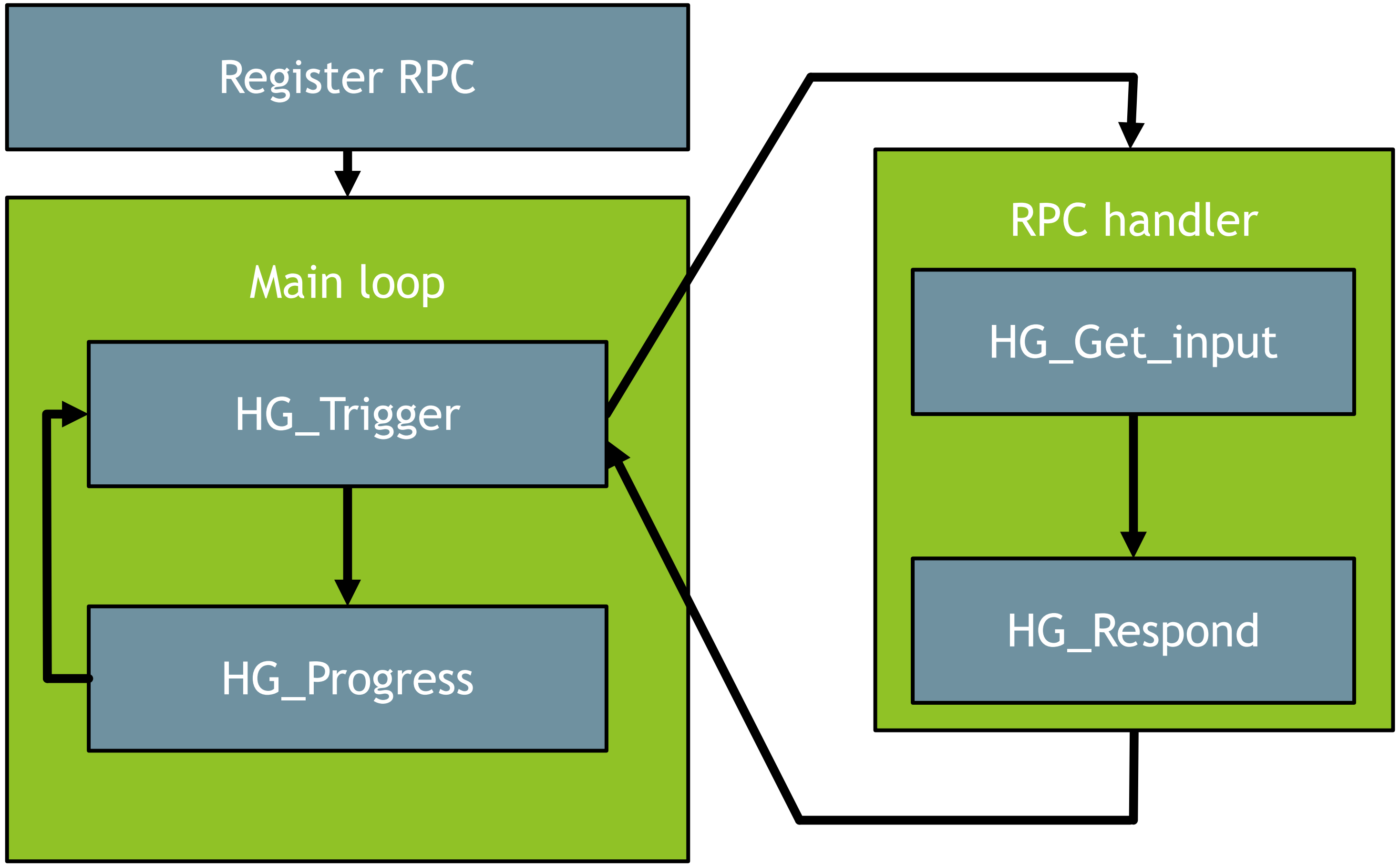
# III. Mercury

High-performance RPC

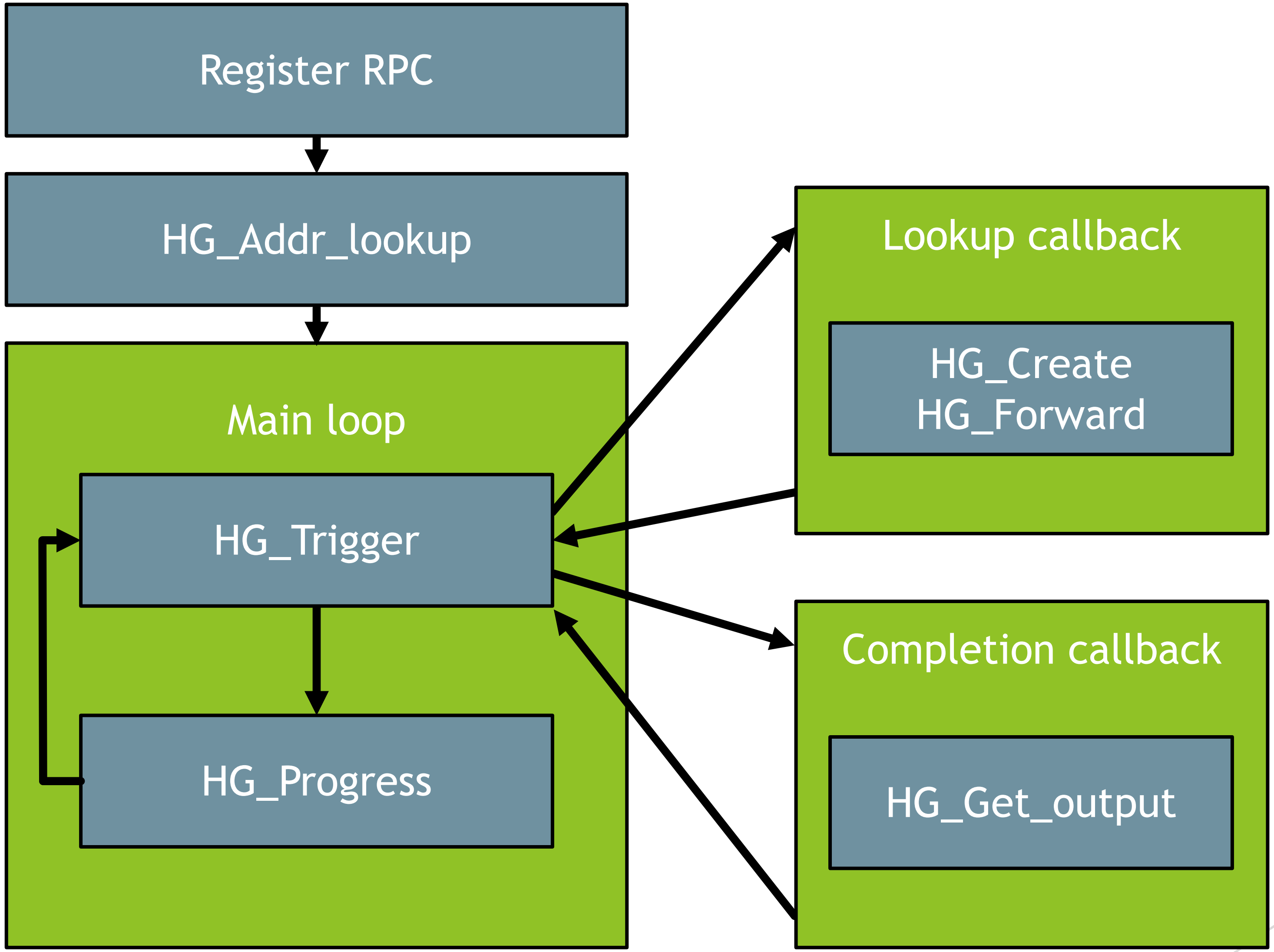
# What is a RPC?

- Remote Procedure Call
  - The client serializes the function’s argument into a buffer
  - The buffer is sent to a server
  - The server executes the function, serializes the return value
  - The server sends the return value back to the client
- RPC Registration
  - Registering a name/id to identify a particular function. On servers, this implies providing the function pointer as well.
- RPC handle
  - An opaque object representing an on-going RPC

# Anatomy of a Mercury server



# Anatomy of a Mercury client



# Diving into examples

- All the following examples are available at
  - https://xgitlab.cels.anl.gov/sds/sds-examples
  - (they are cleaner and more commented there!)

- Example 1: a simple “Hello World” RPC server
- Example 2: sending arguments, returning values
- Example 3: bulk data transfers (RDMA)

- Note: unless you plan to use Mercury alone, no need to dive too much into those examples. With Argobots+Margo, the ugly progress loop will go away ;-)

# Example 1: “Hello World” (server)

(see mercury/01_hello/hello_server.c)

```c
hg_return_t hello_world(hg_handle_t h)
{
    printf("Hello World!\n");
    HG_Destroy(h);
    return HG_SUCCESS;
}
```

# Example 1: “Hello World” (server)

```c
#include <mercury.h>

static hg_class_t*     hg_class   = NULL;
static hg_context_t*   hg_context = NULL;

int main(int argc, char** argv)
{
    hg_return_t ret;

    hg_class = HG_Init("bmi+tcp://localhost:1234", HG_TRUE);
    hg_context = HG_Context_create(hg_class);

    hg_id_t rpc_id = HG_Register_name(hg_class, "hello",
                                      NULL, NULL, hello_world);

    HG_Registered_disable_response(hg_class, rpc_id, HG_TRUE);
```

# Example 1: “Hello World” (server)

```
    /* Main loop listening for incoming RPCs. */
    do
    {
        unsigned int count;
        do {
            ret = HG_Trigger(hg_context, 0, 1, &count);
        } while((ret == HG_SUCCESS) && count);
        HG_Progress(hg_context, 100);
    } while(!stopped);

    ret = HG_Context_destroy(hg_context);
    ret = HG_Finalize(hg_class);

    return 0;
}
```

# Example 1: “Hello World” (client)

(see mercury/01_hello/hello_client.c)

```c
#include <mercury.h>

static hg_class_t*     hg_class   = NULL;
static hg_context_t*   hg_context = NULL;
static hg_id_t         hello_rpc_id;
static int completed = 0;

hg_return_t lookup_callback(const struct hg_cb_info *callback_info);

int main(int argc, char** argv)
{
    hg_return_t ret;
    hg_class = HG_Init("bmi+tcp", HG_FALSE);
    hg_context = HG_Context_create(hg_class);
    hello_rpc_id = HG_Register_name(hg_class, "hello", NULL, NULL, NULL);
    HG_Registered_disable_response(hg_class, hello_rpc_id, HG_TRUE);

    HG_Addr_lookup(hg_context, lookup_callback, NULL,
    "bmi+tcp://localhost:1234", HG_OP_ID_IGNORE);
```

# Example 1: “Hello World” (client)

```
    while(!completed)
    {
        unsigned int count;
        do {
            ret = HG_Trigger(hg_context, 0, 1, &count);
        } while((ret == HG_SUCCESS) && count && !completed);
        HG_Progress(hg_context, 100);
    }

    HG_Context_destroy(hg_context);
    HG_Finalize(hg_class);
    return 0;
}
```

# Example 1: “Hello World” (client)

```
hg_return_t lookup_callback(const struct hg_cb_info
                              *callback_info)
{
    hg_addr_t addr = callback_info->info.lookup.addr;

    hg_handle_t handle;
    HG_Create(hg_context, addr, hello_rpc_id, &handle);

    HG_Forward(handle, NULL, NULL, NULL);

    HG_Addr_free(hg_class, addr);
    HG_Destroy(handle);

    completed = 1;
    return HG_SUCCESS;
}
```

# Example 2: a “sum” server

(see mercury/02_sum/types.h)

```
#include <mercury.h>
#include <mercury_macros.h>

MERCURY_GEN_PROC(sum_in_t,
    ((int32_t)(x))\
    ((int32_t)(y)))

MERCURY_GEN_PROC(sum_out_t, ((int32_t)(ret)))
```

This will generate

- The sum_in_t and sum_out_t structures
- The hg_proc_sum_in_t and hg_proc_sum_out_t functions to serialize/deserialize these structures

# Example 2: a “sum” server

(see mercury/02_sum/sum_server.c)

```
hg_return_t sum(hg_handle_t handle)
{
    sum_in_t in;    /* input structure for the RPC  */
    sum_out_t out; /* output structure for the RPC */

    HG_Get_input(handle, &in);

    out.ret = in.x + in.y;

    HG_Respond(handle,NULL,NULL,&out);

    HG_Free_input(handle, &in);
    HG_Destroy(handle);

    return HG_SUCCESS;
}
```

# Example 2: a “sum” server
(see mercury/02_sum/sum_server.c)

```
MERCURY_REGISTER(hg_class, "sum", sum_in_t, sum_out_t, sum);
```

instead of

```
rpc_id = HG_Register_name(hg_class, "hello", NULL, NULL, hello_world);
HG_Registered_disable_response(hg_class, rpc_id, HG_TRUE);
```

# Example 2: a “sum” server (client)

(see mercury/02_sum/sum_client.c)

```
sum_rpc_id = MERCURY_REGISTER(hg_class, "sum",
                    sum_in_t, sum_out_t, NULL);
```

# Example 2: a “sum” server (client)

(see mercury/02_sum/sum_client.c)

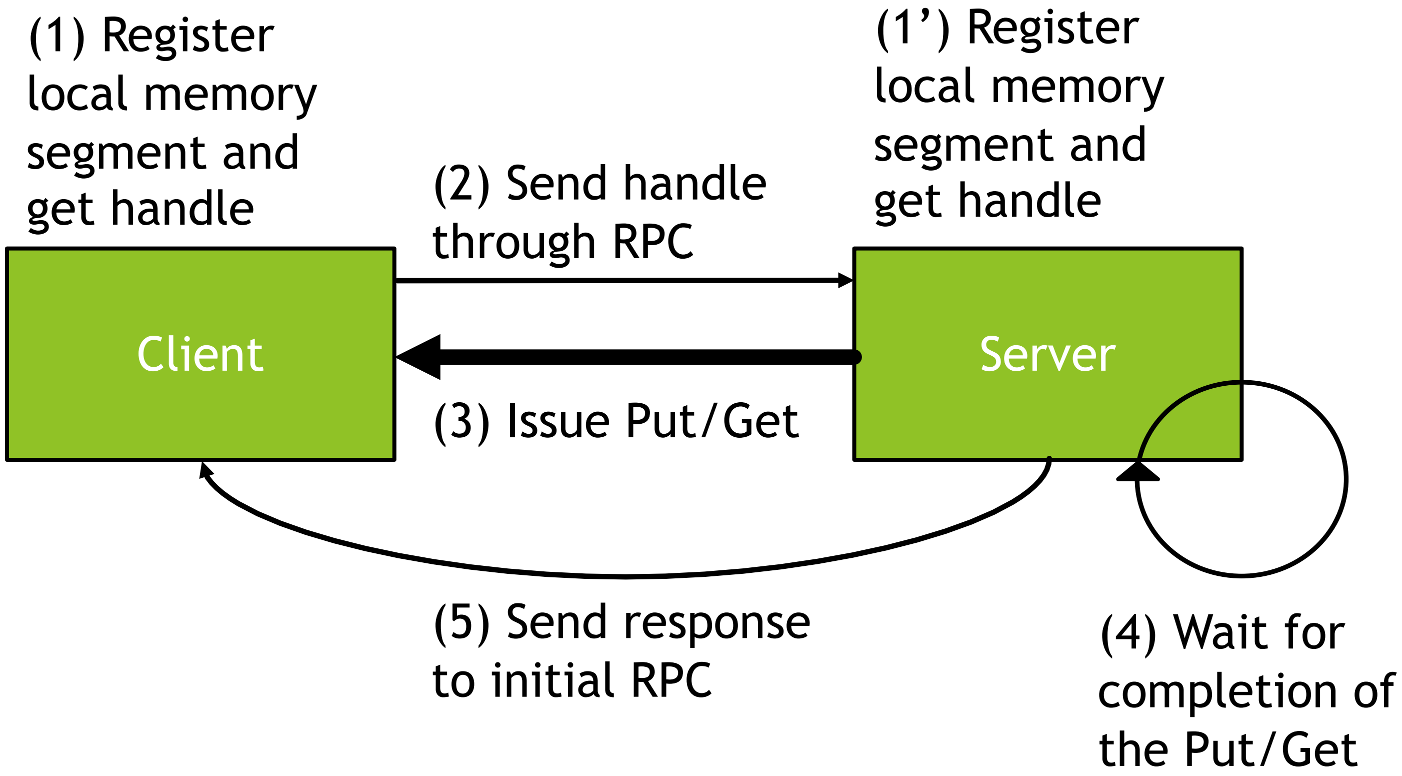
```c
hg_return_t sum_completed(const struct hg_cb_info *info)
{
    sum_out_t out;
    HG_Get_output(info->info.forward.handle, &out);

    printf("Got response: %d\n", out.ret);

    HG_Free_output(info->info.forward.handle, &out);

    HG_Destroy(info->info.forward.handle);
    completed = 1;
    return HG_SUCCESS;
}
```

# Example 3: bulk transfers



# Example 3: bulk transfers (types)

(see mercury/04_bulk/types.h)

```
#include <mercury.h>
#include <mercury_bulk.h>
#include <mercury_proc_string.h>
#include <mercury_macros.h>

MERCURY_GEN_PROC(save_in_t,
    ((hg_string_t)(filename))\
    ((hg_size_t)(size)) \
    ((hg_bulk_t)(bulk_handle)))

MERCURY_GEN_PROC(save_out_t,
((int32_t)(ret)))
```

The client sends a filename, a size and the bulk handle to access the data

The server return 0 or an error code

# Example 3: bulk transfers (client)

(see mercury/04_bulk/save_client.c)

In the lookup callback:

```
save_in_t in;
hg_bulk_t bulk_handle;
HG_Bulk_create(hg_class, 1, (void**) &buffer, &size,
                    HG_BULK_READ_ONLY, &bulk_handle);
in.bulk_handle = bulk_handle;
```

# Example 3: bulk transfers (server)

(see mercury/04_bulk/save_server.c)

RPC handler:

```
hg_return_t save(hg_handle_t handle)
{
    hg_return_t ret;
    save_in_t in;

    struct hg_info* info = HG_Get_info(handle);

    HG_Get_input(handle, &in);

    rpc_state* my_rpc_state = (rpc_state*)calloc(1,sizeof(rpc_state));
    my_rpc_state->handle = handle;
    my_rpc_state->filename = strdup(in.filename);
    my_rpc_state->size = in.size;
    my_rpc_state->buffer = calloc(1,in.size);
```

# Example 3: bulk transfers (server)

(see mercury/04_bulk/save_server.c)

RPC handler:

```
…
    HG_Bulk_create(stt->hg_class, 1, &(my_rpc_state->buffer),
              &(my_rpc_state->size), HG_BULK_WRITE_ONLY,
              &(my_rpc_state->bulk_handle));

    HG_Bulk_transfer(stt->hg_context, save_bulk_completed,
              my_rpc_state, HG_BULK_PULL,
              info->addr, in.bulk_handle, 0,
              my_rpc_state->bulk_handle, 0, my_rpc_state->size,
              HG_OP_ID_IGNORE);

    return HG_SUCCESS;
}
```

# Example 3: bulk transfers (server)

(see mercury/04_bulk/save_server.c)

Bulk completion callback:

```
hg_return_t save_bulk_completed(const struct hg_cb_info *info)
{
    rpc_state* my_rpc_state = info->arg;

    printf("Writing file %s\n", my_rpc_state->filename);
    /* write file here */

    save_out_t out;
    out.ret = 0;

    HG_Respond(my_rpc_state->handle, NULL, NULL, &out);

    HG_Bulk_free(my_rpc_state->bulk_handle);

    return HG_SUCCESS;
}
```

# Example 3: bulk transfers (client)

(see mercury/04_bulk/save_client.c)

```
hg_return_t lookup_callback(const struct hg_cb_info *callback_info)
{
    …
    hg_handle_t handle;
    HG_Create(state->hg_context, addr, save_rpc_id, &handle);

    save_in_t in;
    in.filename = …;
    in.size     = …;

    HG_Bulk_create(hg_class, 1, (void**) &buffer, &size,
                   HG_BULK_READ_ONLY, &in.bulk_handle);

    HG_Forward(handle, save_completed, …, &in);

    …
    return HG_SUCCESS;
}
```

# Example 3: bulk transfers (client)

(see mercury/04_bulk/save_client.c)

```
hg_return_t save_completed(const struct hg_cb_info *info)
{

    HG_Get_output(info->info.forward.handle, &out);
    printf("Got response: %d\n", out.ret);
    HG_Bulk_free(bulk_handle);

    return HG_SUCCESS;
}
```

# Some notes on bulk transfer

- When creating a bulk handle, you give Mercury information regarding how it will access the buffer
  - `HG_BULK_READ_ONLY`: Buffer will only be read until bulk handle is destroyed
  - `HG_BULK_WRITE_ONLY`: Buffer will only be written until bulk handle is destroyed
  - `HG_BULK_READWRITE`: a wild guess, anyone?
- Mercury can sometimes guess what you intend to do when creating a bulk handle, and if the buffer is small enough, it can send the data along with the RPC request
  - E.g. the buffer is exposed with `HG_BULK_READ_ONLY` and you send the handler to the server : Mercury will guess that you want the server to pull from it

# Some last notes about these examples

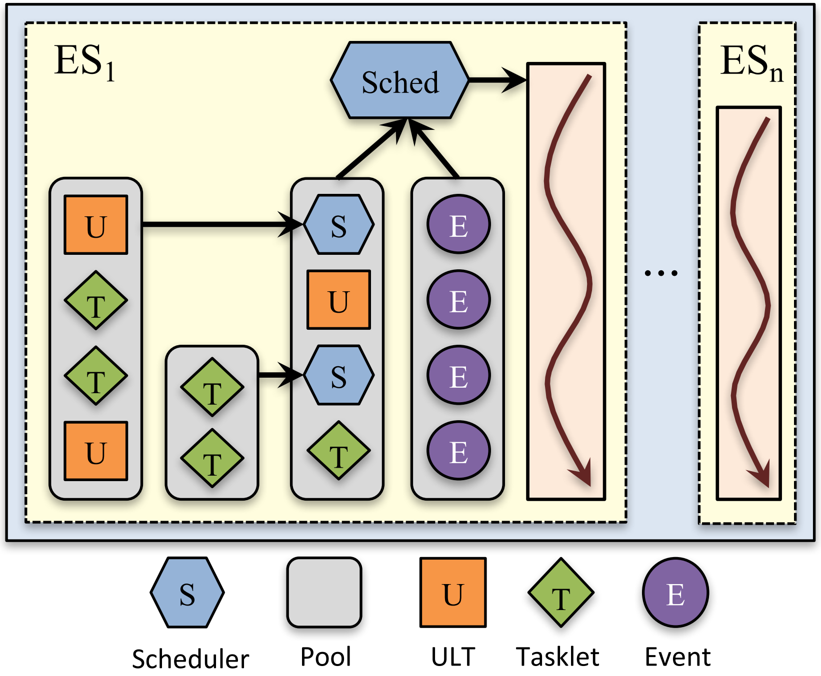
- For clarity
  - Many error-checking lines of code (e.g. assert(ret == HG_SUCCESS)) have been removed
  - Many resource-destruction function calls have been omitted as well
  - Global variables were used
- All callback functions (lookup, RPC completion, bulk transfer completion) have a void* argument to pass along user-data (a way of creating a closure) in order to avoid global variables. See mercury/03_uargs for the “sum server” example rewritten in a “clean” way with these user arguments.

# IV. Argobots

Lightweight threading/tasking

# Argobots Execution Model

- **Execution Streams (ES)**
  - Sequential instruction stream
    - Can consist of one or more work units
  - Mapped efficiently to a hardware resource
  - Implicitly managed progress semantics
    - One blocked ES cannot block other ESs
- **User-level Threads (ULTs)**
  - Independent execution units in user space
  - Associated with an ES when running
  - Yieldable and migratable
  - Can make blocking calls
- **Tasklets**
  - Atomic units of work
  - Asynchronous completion via notifications
  - Not yieldable, migratable before execution
  - Cannot make blocking calls

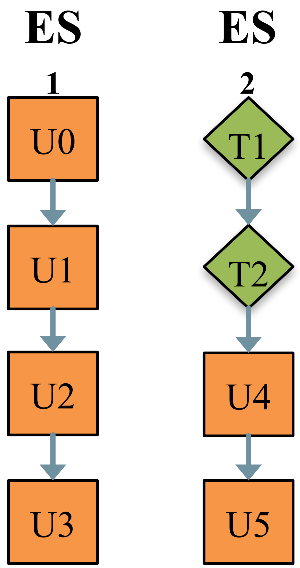



**Argobots Execution Model**

- **Scheduler**
  - Stackable scheduler with pluggable strategies
- **Synchronization primitives**
  - Mutex, condition variable, barrier, future
- **Events**
  - Communication triggers

# Explicit Mapping ULT/Tasklet to ES

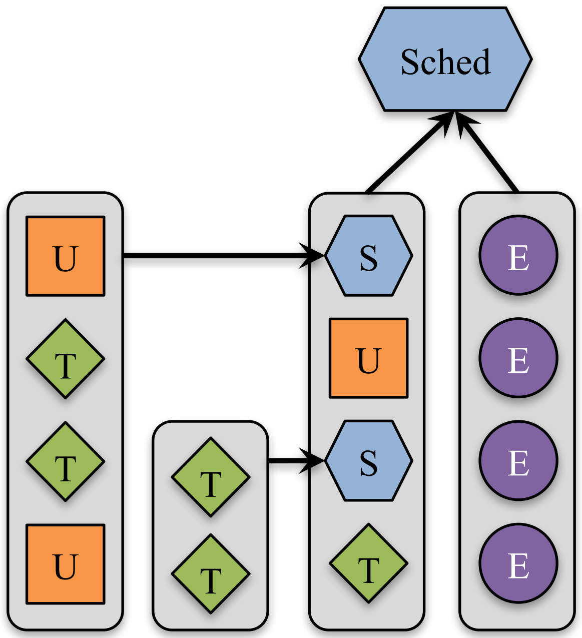
- The user needs to map work units to ESs
- No smart scheduling, no work-stealing unless the user wants to use




- Benefits
  - Allow locality optimization
    - Execute work units on the same ES
  - No expensive lock is needed between ULTs on the same ES
    - They do not run concurrently
    - A flag is enough

# Stackable Scheduler with Pluggable Strategies

- Associated with an ES
- Can handle ULTs and tasklets
- *Can handle schedulers*
  - Allows to stack schedulers hierarchically
- Can handle asynchronous events
- *Users can write schedulers*
  - Provides **mechanisms**, not policies
  - Replace the default scheduler
    - E.g., FIFO, LIFO, Priority Queue, etc.
- ULT can explicitly *yield to* another ULT
  - Avoid scheduler overhead

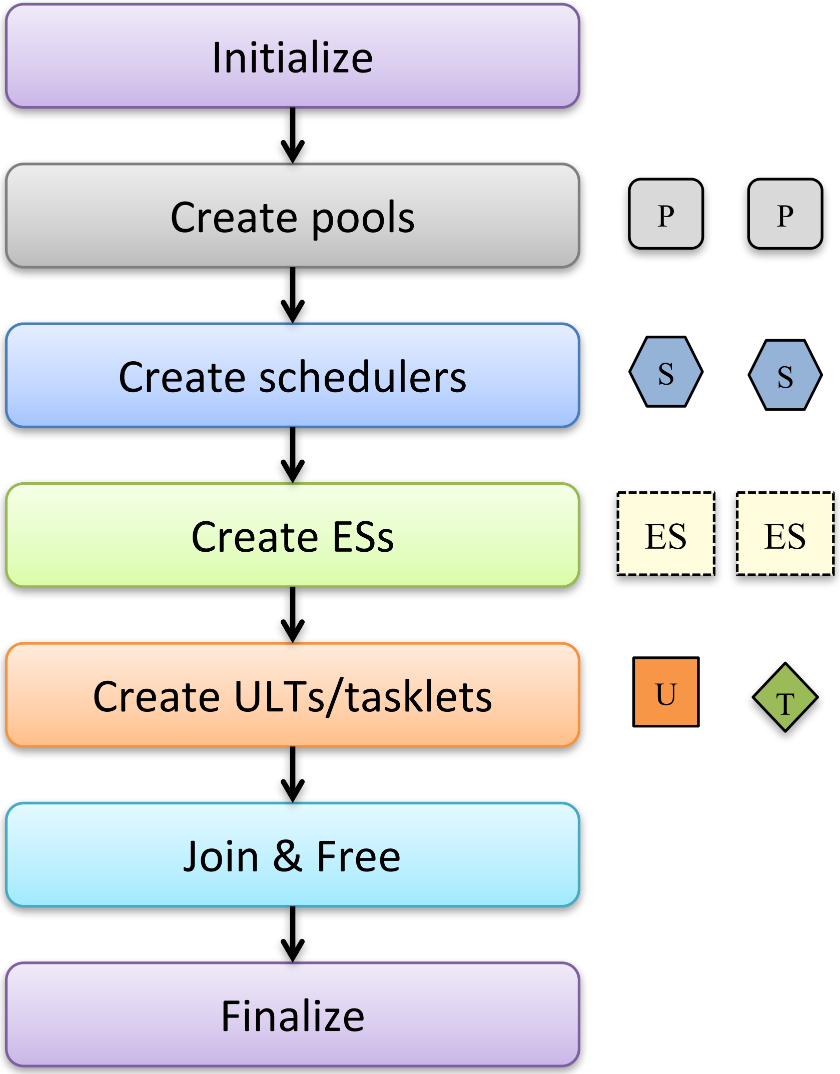


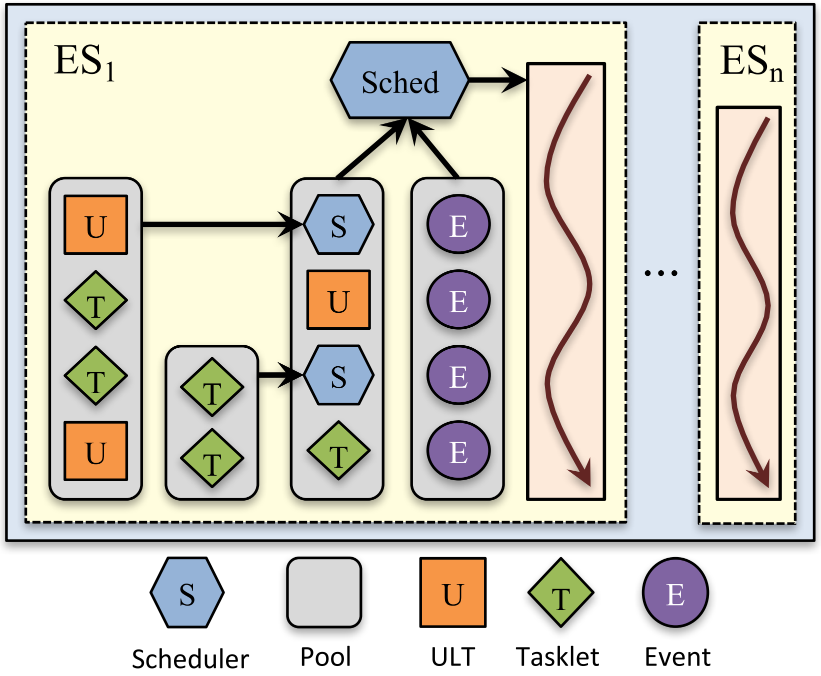


yield()

yield_to(target)

# How to Write Argobots Code




**Argobots Execution Model**

# Diving into examples

- All the following examples are available at
  - https://xgitlab.cels.anl.gov/sds/sds-examples
  - (they are cleaner and more commented there!)

- Example 1: Execution streams and threads
- Example 2: Tasks
- Example 3: Work stealing with shared pool
- Example 4: Mutex and condition variables
- Example 5: Eventuals and futures

# Example 1: ES and ULT

(see argobots/01_threads/01_threads.c)

```
#include <abt.h>

#define NUM_XSTREAMS     4

/* This is the function executed by each of the threads */
void thread_hello(void *arg)
{
    /* Get the rank of the ES */
    int rank;
    ABT_xstream_self_rank(&rank);
    printf("ULT %d in XSTREAM %d: Hello, world!\n",
(int)(size_t)arg, rank);
}
```

# Example 1: ES and ULT

(see argobots/01_threads/01_threads.c)

```
int main(int argc, char *argv[])
{
    ABT_xstream xstreams[NUM_XSTREAMS];
    ABT_pool    pools[NUM_XSTREAMS];
    ABT_thread  threads[NUM_XSTREAMS];
    size_t i;

    /* Initialize Argobots */
    ABT_init(argc, argv);

    /* Execution Streams
     * xtrseam[0] will be the current ES, no need to create it.
     */
    ABT_xstream_self(&xstreams[0]);
    for (i = 1; i < NUM_XSTREAMS; i++) {
        ABT_xstream_create(ABT_SCHED_NULL, &xstreams[i]);
    }
```

# Example 1: ES and ULT

(see argobots/01_threads/01_threads.c)

```
/* Get the first pool associated with each ES */
for (i = 0; i < NUM_XSTREAMS; i++) {
    ABT_xstream_get_main_pools(xstreams[i], 1, &pools[i]);
}

/* Create ULTs */
for (i = 0; i < NUM_XSTREAMS; i++) {
    ABT_thread_create(pools[i], thread_hello, (void *)i,
                      ABT_THREAD_ATTR_NULL, &threads[i]);
}
```

# Example 1: ES and ULT

(see argobots/01_threads/01_threads.c)

```c
    /* Join & Free */
    for (i = 0; i < NUM_XSTREAMS; i++) {
        ABT_thread_join(threads[i]);
        ABT_thread_free(&threads[i]);
    }
    for (i = 1; i < NUM_XSTREAMS; i++) {
        ABT_xstream_join(xstreams[i]);
        ABT_xstream_free(&xstreams[i]);
    }

    /* Finalize */
    ABT_finalize();

    return 0;
}
```

# Example 2: Tasks

(see argobots/02_tasks/02_tasks.c)

```
void task_hello(void *arg)
{
        printf("TASK%d: Hello, world!\n", (int)(size_t)arg);
}


/* Create Tasks */
for (i = 0; i < (NUM_XSTREAMS*TASKS_PER_XTREAM); i++) {
        ABT_task_create(pools[i % NUM_XSTREAMS],
                        task_hello, (void *)i, NULL);
}
```

# Example 3: Work stealing with shared pool

(see argobots/03_shared_pool/03_shared_pool.c)

```
ABT_pool shared_pool;

/* Create a shared pool */
ABT_pool_create_basic(ABT_POOL_FIFO, ABT_POOL_ACCESS_MPMC,
                      ABT_TRUE, &shared_pool);

/* Create schedulers */
for (i = 0; i < NUM_XSTREAMS; i++) {
    ABT_sched_create_basic(ABT_SCHED_DEFAULT, 1,
            &shared_pool, ABT_SCHED_CONFIG_NULL, &scheds[i]);
}
```

- All the schedulers are created with the same pool; tasks and ULTs are added to this common pool and the ES execute them

# Example 4: Mutex and condition variables

(see argobots/04_mutex/04_mutex.c and argobots/09_cond_var/09_cond_var.c)

```
ABT_mutex my_mutex = ABT_MUTEX_NULL;
ABT_mutex_create(&my_mutex);
ABT_mutex_lock(my_mutex);
ABT_mutex_unlock(my_mutex);
ABT_mutex_free(&my_mutex);
```

```
ABT_cond  my_cond;
ABT_cond_create(&my_cond);
ABT_mutex_lock(my_mutex);
ABT_cond_wait(my_cond,my_mutex);
ABT_cond_signal(my_cond);
ABT_mutex_unlock(my_mutex);
```

# Example 5: Eventuals and futures

(see argobots/05_eventual/05_eventual.c and argobots/06_future/06_future.c)

```
ABT_eventual my_eventual;
ABT_eventual_create(sizeof(int), &my_eventual);
ABT_eventual_set(my_eventual, &r, sizeof(r));
ABT_eventual_wait(my_eventual, (void**)&r);
ABT_eventual_free(&my_eventual);

void future_is_ready(void** args) { … }

ABT_future my_future;
ABT_future_create(num_elements, future_is_ready, &my_future);
ABT_future_set(my_future,(void*)x);
ABT_future_wait(my_future);
ABT_future_free(&my_future);
```

# Other features

- Not described in this tutorial
  - Writing you own scheduler
  - Stacking schedulers
  - Migrating tasks and threads across execution streams

# V. Margo

Bridging Mercury and Argobots

# Margo: linking Mercury and Argobots

- Motivation
  - Ugly progress loop and callbacks in Mercury
  - Argobots provides a great opportunity for changing the way Mercury is used
- In practice
  - Mercury progress loop placed in a separate execution stream
  - RPC calls dispatched to a set of execution streams sharing a common pool
- What’s next
  - Example 1: Mercury “hello world” revisited
  - Example 2: Mercury “bulk transfers” revisited

# Example 1: Hello World (server)

(see margo/01_hello/hello_server.c)

```c
hg_return_t hello_world(hg_handle_t h)
{
    printf("Hello World!\n");

    HG_Destroy(h);

    if(some condition) {
        margo_finalize(mid);
    }

    return HG_SUCCESS;
}
DEFINE_MARGO_RPC_HANDLER(hello_world)
```

# Example 1: Hello World (server)

(see margo/01_hello/hello_server.c)

```
#include <abt.h>
#include <abt-snoozer.h>
#include <margo.h>

static hg_class_t*      hg_class   = NULL;
static hg_context_t*    hg_context    = NULL;
static margo_instance_id mid       = MARGO_INSTANCE_NULL;

hg_return_t hello_world(hg_handle_t h);
DECLARE_MARGO_RPC_HANDLER(hello_world)

int main(int argc, char** argv)
{
    hg_return_t ret;
    hg_class = HG_Init("bmi+tcp://localhost:1234", HG_TRUE);
    hg_context = HG_Context_create(hg_class);
```

# Example 1: Hello World (server)

(see margo/01_hello/hello_server.c)

```
…
    ABT_init(argc, argv);
    ABT_snoozer_xstream_self_set();

    mid = margo_init(0, 0, hg_context);

    hg_id_t rpc_id = HG_Register_name(hg_class, "hello", NULL, NULL,
        hello_world_handler);

    HG_Registered_disable_response(hg_class, rpc_id, HG_TRUE);

    margo_wait_for_finalize(mid);

    ABT_finalize();
    HG_Context_destroy(hg_context);
    HG_Finalize(hg_class);

    return 0;
}
```

# Example 1: Hello World (client)

(see margo/01_hello/hello_client.c)

```
#include <abt.h>
#include <abt-snoozer.h>
#include <margo.h>

static hg_class_t*      hg_class   = NULL;
static hg_context_t*    hg_context = NULL;
static margo_instance_id mid       = MARGO_INSTANCE_NULL;
static hg_id_t          hello_rpc_id;
static hg_addr_t        svr_addr;

static void run_my_rpc(void *arg);

int main(int argc, char** argv)
{
    ABT_xstream xstream;
    ABT_pool pool;
    ABT_thread threads[4];
```

# Example 1: Hello World (client)

(see margo/01_hello/hello_client.c)

…

```
hg_class = HG_Init("bmi+tcp", HG_FALSE);
hg_context = HG_Context_create(hg_class);
ABT_init(argc, argv);

ABT_snoozer_xstream_self_set();
ABT_xstream_self(&xstream);
ABT_xstream_get_main_pools(xstream, 1, &pool);

mid = margo_init(0, 0, hg_context);

hello_rpc_id = HG_Register_name(hg_class, "hello", NULL, NULL, NULL);

HG_Registered_disable_response(hg_class, hello_rpc_id, HG_TRUE);

margo_addr_lookup(mid, "bmi+tcp://localhost:1234", &svr_addr);
```

# Example 1: Hello World (client)

(see margo/01_hello/hello_client.c)

```
    int i;
    for(i=0; i<4; i++) {
            ABT_thread_create(pool, run_my_rpc, NULL,
                      ABT_THREAD_ATTR_NULL, &threads[i]);
    }

    ABT_thread_yield_to(threads[0]);

    for(i=0; i<4; i++) {
        ABT_thread_join(threads[i]);
        ABT_thread_free(&threads[i]);
    }

    margo_finalize(mid);
    ABT_finalize();
    HG_Context_destroy(hg_context);
    HG_Finalize(hg_class);
    return 0;
}
```

# Example 1: Hello World (client)

(see margo/01_hello/hello_client.c)

```
void run_my_rpc(void *arg)
{
    hg_handle_t handle;
    hg_return_t ret;
    ABT_thread self;
    ABT_thread_id id;

    HG_Create(hg_context, svr_addr, hello_rpc_id, &handle);

    margo_forward(mid, handle, NULL);

    HG_Destroy(handle);

    ABT_thread_self(&self);
    ABT_thread_get_id(self,&id);

    printf("ULT [%d] done.\n", (int)id);
}
```

# Example 2: Bulk transfers

(see margo/03_bulk/save_server.c)

```
ret = HG_Bulk_transfer(hg_context, callback,
        args, HG_BULK_PULL, origin_addr, origin_bulk_handle,
        origin_offset, local_bulk_handle, local_offset, size,
        op_id);
```

```
ret = margo_bulk_transfer(mid, HG_BULK_PULL,
        origin_addr, origin_bulk_handle, origin_offset,
        local_bulk_handle, local_offset, size);
```

No more callback, this call hands the current ES to other threads/tasks until the transfer is completed, then continues.

# Some additional notes

- Doing POSIX I/O inside a RPC handler will block the current ES instead of context-switching to other tasks/threads
  - Solution: https://xgitlab.cels.anl.gov/sds/abt-io
  - ABT-IO has an interface similar to POSIX I/O but will allow other tasks/threads to progress on the ES while I/O is being performed
- Distributed services: requires creating multiple servers that “know” each other
  - Solution: https://xgitlab.cels.anl.gov/sds/ssg
  - SSG (Simple Static Grouping)
  - Allows to bootstrap a set of servers using either MPI or a configuration file

# VI. Related work

Some projects using Mercury, Argobots, Margo, etc.

# BOLT: OpenMP over Lightweight Threads

- About BOLT
  - BOLT is a recursive acronym that stands for "BOLT is OpenMP over Lightweight Threads”
  - https://bolt-omp.org
- Goal
  - OpenMP framework that exploits lightweight threads and tasks

Nested Massive Parallelism

Fine-grained Task Parallelism

Interoperability with MPI and Other Internode Programming Models

# Motivation

- Fine-grained parallelism in OpenMP
  - Nested parallelism
  - Task parallelism
- Better interoperability between OpenMP and MPI
  - Needs lightweight context-switch on blocking operations

- Pthread-based OpenMP implementations may not be efficient to handle those issues
  - The context-switch overhead of Pthread is high
  - Oversubscription with Pthreads is too expensive

- BOLT utilizes *lightweight threads and tasks* instead of Phtreads
  - Oversubscription becomes much less costly because of very low context-switch overhead of lightweight threads

# Nested Parallel Loop: Microbenchmark

```
int in[1000][1000], out[1000][1000];

#pragma omp parallel for
for (i = 0; i < 1000; i++) {
   lib_compute(i);
}

lib_compute(int x)
{
   #pragma omp parallel for
   for (j = 0; j < 1000; j++)
      out[x][j] = cosine(in[x][j]);
}
```

A thread for each CPU is created by default

Each thread executes a portion

Each thread creates more threads for the second loop

Each inner thread executes a portion

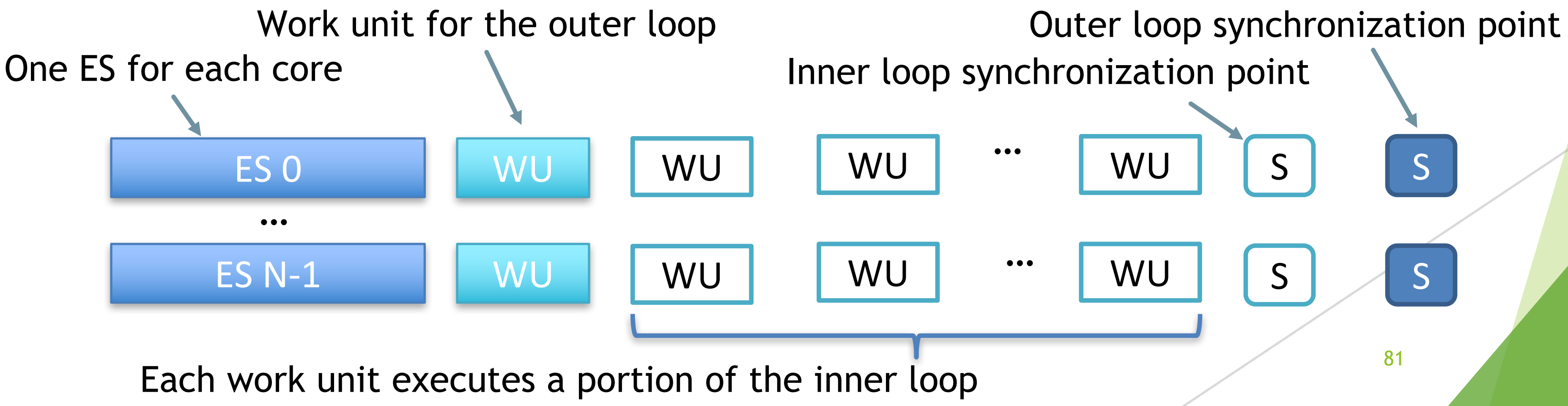


Contribution: Adrian Castello (Universitat Jaume I)

# Nested Parallel Loop: Implementations

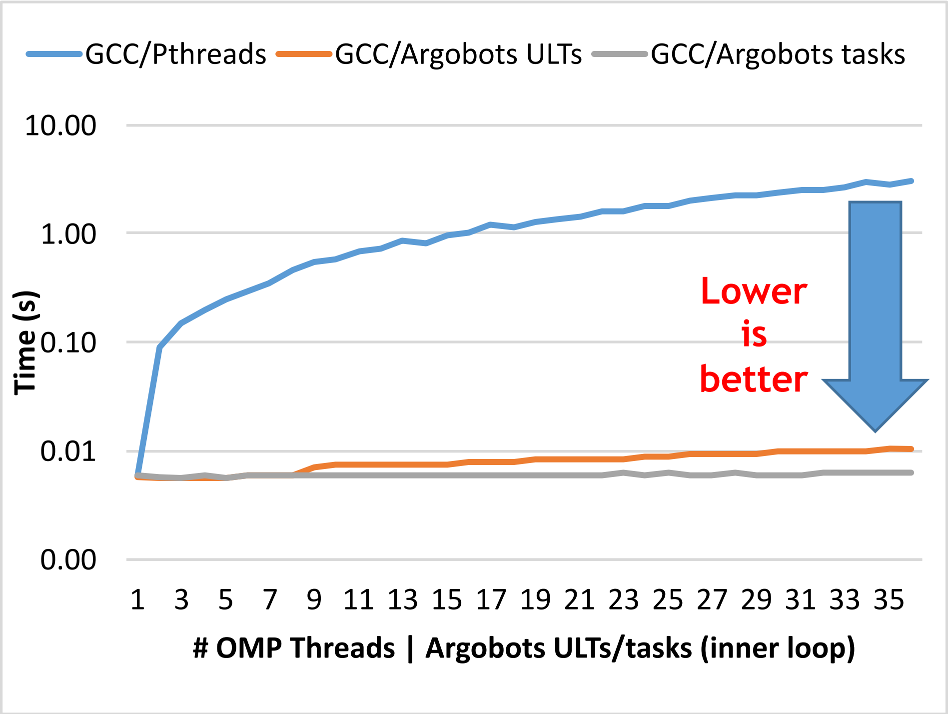
- GCC
  - Does not reuse the idle threads in nested parallel constructs
  - All thread teams inside a parallel region need to be created
- ICC
  - Reuse idle threads
    - If there are not more threads available, new threads are created
- *All created threads are OS threads and add overhead*

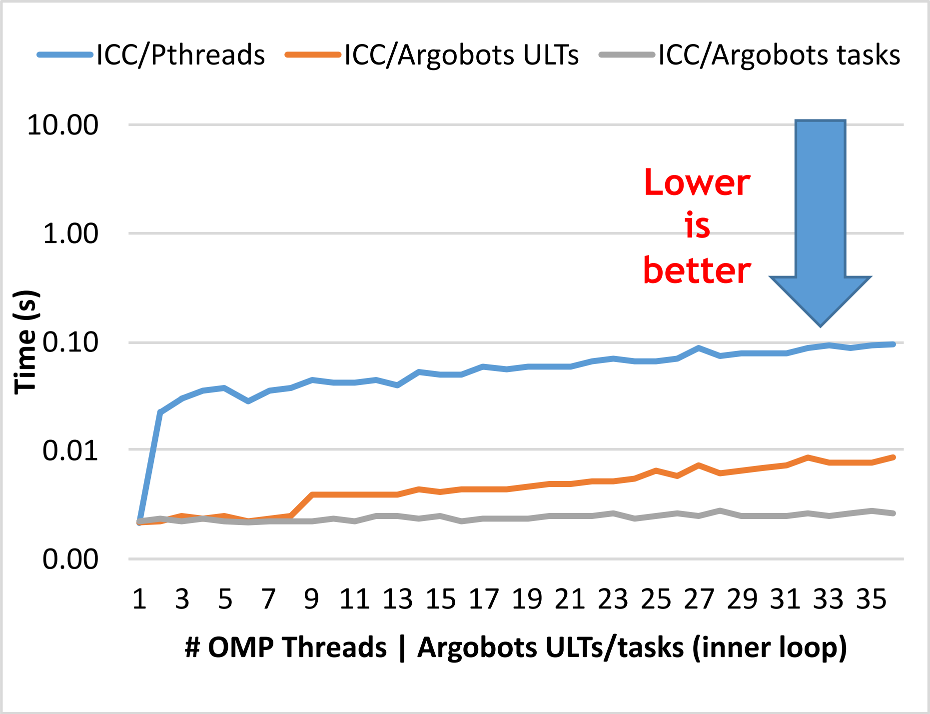
- Implementation using Argobots
  - Creates ULTs or tasklets for both outer loop and inner loop



# Nested Parallel Loop: Performance

Execution time for 36 threads in the outer loop




GCC OpenMP implementation does not reuse idle threads in nested parallel regions, all the teams of threads need to be created in each iteration

Some overhead is added by creating ULTs instead of tasks

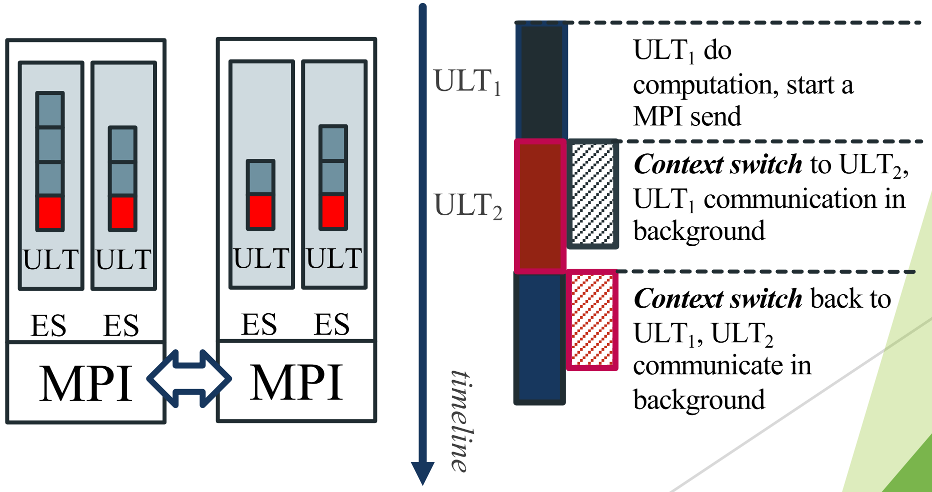
# MPI+Argobots

- Exploiting in MPI the advanced features of Argobots
  - Reduce locking frequency in the MPI runtime
    - The cost of consistency within an ES is lower than across ESs
    - A ULT can give up the critical section without giving up the lock
    - Low latency ownership passing: Less atomics and memory barriers
  - Advanced synchronization mechanisms (e.g. message-driven synchronization)
- Interaction with other programming systems
  - Argobots = threading layer; MPI = communication layer
- Porting more applications to the MPI+Argobots model
  - Several irregular applications (fine-grained concurrency and communication) can benefit from this model
- Publication
  - Follow up publication which aims at a full integration of MPI and Argobots
- Software
  - First release in the upcoming months

# Argobots-Aware MPI Runtime

- Problem
  - Traditional MPI implementations are only aware of kernel threads
  - Thread-synchronization costly to ensure thread-safety and progress requirement from MPI
  - Wasted resources if a kernel thread blocks for MPI communication
- Solution
  - An MPI implementation aware of Argobots threads
  - Lightweight context switching to overlap costly blocking operations (communication, locks, etc.)
  - Reduced thread-synchronization opportunities (guaranteed consistency within an ES without locks or memory barriers)

- Recent results
  - Developed an MPICH+Argobots prototype
  - Demonstrated the ability to overlap blocking communication with HPCG, SpMV, etc.
  - Deployed successfully a fully threaded Graph500 benchmark implementation
- Impact/Potential
  - The new MPI+Argobots model has the potential to overcome the long lasting multithreaded MPI communication challenge




**MPI+Argobots Execution Model**

Contact:
- Pavan Balaji balaji@anl.gov
- Abdelhalim Amer aamer@anl.gov

This work is supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-06CH11357.

**Acknowledgements**

This document was prepared using materials and works from the following contributors

Abdelhalim Amer, George Amvrosiadis, Pavan Balaji, Philip Carns, Chuck Cranor, Matthieu Dorier, Garth Gibson, Kevin Harms, Saurabh Kadekodi, Rob Latham, Joe Lee, Rob Ross, Brad Settlemyer, Galen Shipman, Shane Snyder, Jerome Soumagne, Qing Zheng

**All feedbacks on this tutorial are appreciated:** mdorier@anl.gov

Exercises follow! Stay around!...

# Exercise 1: a phone book

- Goal
  - A server maintains a table associating names (strings) with phone numbers (string)
  - Clients can connect to this server and add/modify/request entries
- How we will implement that
  - A server listens for 2 kinds of RPC: “set_num” and “get_num”
  - The client presents a prompt to the user, the user can type the following commands:
    - set name phone
    - get name
- Provided
  - sds_example/margo/exercises/phone_book
  - The phonebook structure and its functions (in phone_book.h)
  - Skeletons of clients and servers
  - TODO: fills the TODO in phone_client.c, phone_server.c and types.h

# Exercise 2: a forwarding server

- Goal
  - We take the example of the “backup server” again, but this time and intermediate server relays the request to the actual backup server
- Scenario
  - Client C sends a request to save a file to a “forward server” F
  - F forwards the request to a backup server B
  - B issues an RDMA “pull” operation to get the data from the client
  - B sends a response to F
  - F sends a response to C
- Where to look
  - sds_example/margo/exercises/forward_save

# Exercise 2: a forwarding server

- In Mochi terms
  - The client sends a “forward_save” RPC to the forwarder
  - The forwarder sends a “save” RPC to the backup server
- Initially, the client is setup to send a “save” to the server directly
  - Find the TODO in save_server.c so that it listens to a different port
- The “save_in_t” and “save_out_t” types can be used by both the save and forward_save RPCs, however the client will have to add its address
  - Look at types.h and add a string field for the address
  - Look at the TODOs in save_client.c to have the address serialized and sent through the RPC, and have the save_client issue a forward_save instead of a save
- Fill up the TODOs in forwarder.c