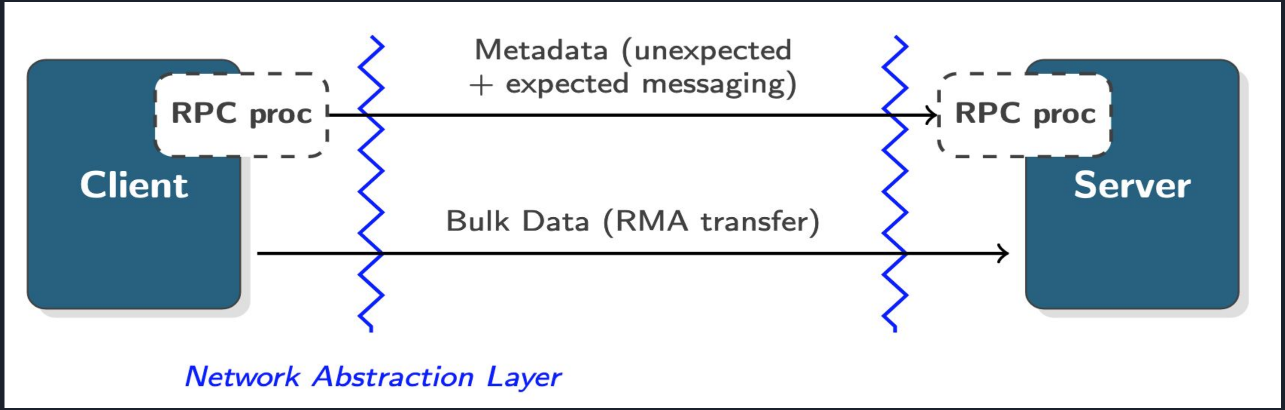
Margo and Thallium

Mochi Bootcamp
February 6, 2020

Argonne NATIONAL LABORATORY

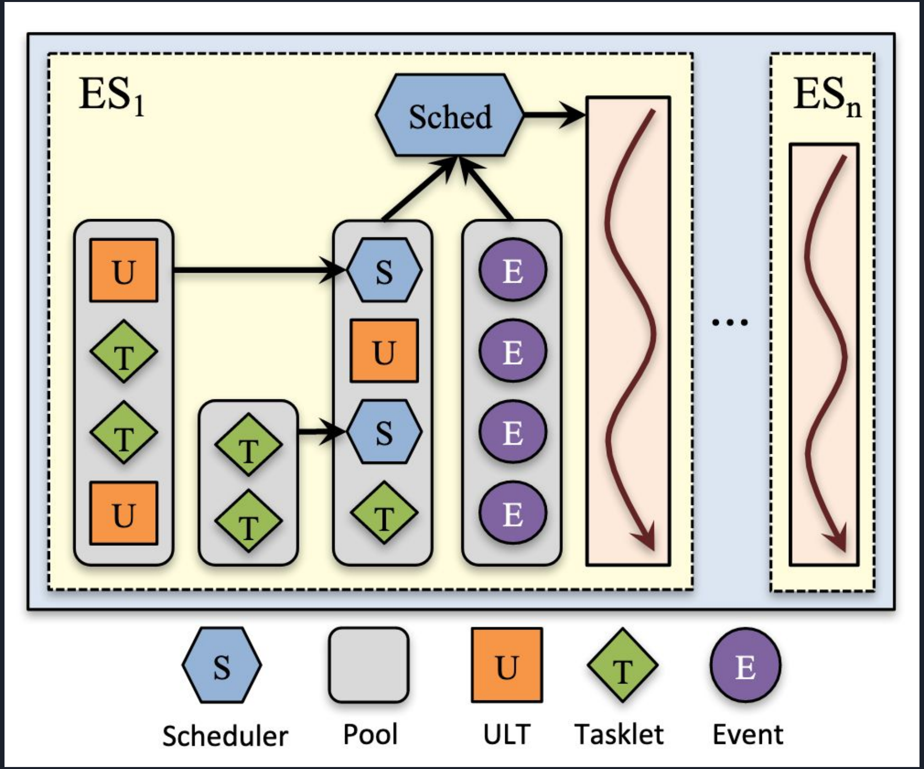
A little bit about Mercury and Argobots

Mercury: RPC and RDMA for HPC



- Multiple network abstractions
- Data serialization using macros
- Bulk data transfers (using RDMA when available)
- Explicit progress loop
- Callback-driven programming model

Argobots' execution model



Important concepts

- Execution streams
- User level threads
- Pools
- Schedulers

Argobots+Mercury = Margo

- Mercury progress placed in a ULT
- More natural programming model based on ULTs rather than callbacks

Getting started with Margo

Where to find the material

- https://mochi.readthedocs.io
- https://xgitlab.cels.anl.gov/sds/mochi-doc

```
git clone https://xgitlab.cels.anl.gov/sds/mochi-doc.git
cd mochi-doc/code/margo
```

Margo initialization

Server initialization

Server

```c
#include <margo.h>

int main(int argc, char** argv)
{
    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 0, -1);
    assert(mid);

    hg_addr_t my_address;
    margo_addr_self(mid, &my_address);
    char addr_str[128];
    size_t addr_str_size = 128;
    margo_addr_to_string(mid, addr_str, &addr_str_size, my_address);
    margo_addr_free(mid,my_address);
    printf("Server running at address %s\n", addr_str);

    margo_wait_for_finalize(mid);

    return 0;
}
```

margo/01_init/server.c

Server

Server initialization

```
#include <margo.h>

int main(int argc, char** argv)
{
    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 0, -1);
    assert(mid);

    hg_addr
    margo_ad
    char add
    size_t a
    margo_addr_to_string(mid, addr_str, &addr_str_size, my_address);
    margo_addr_free(mid,my_address);
    printf("Server running at address %s\n", addr_str);

    margo_wait_for_finalize(mid);

    return 0;
}
```

How many execution streams for RPC handlers?

Which protocol should we use? (tcp, na+sm, ofi+gni, etc.)

Should we run the progress loop in a separate execution stream?

Wait here until another thread calls `margo_finalize(mid)`

margo/01_init/server.c

Client initialization

Client

```
#include <margo.h>

int main(int argc, char** argv)
{
    margo_instance_id mid = margo_init("tcp", MARGO_CLIENT_MODE, 0, 0);
    assert(mid);

    margo_finalize(mid);

    return 0;
}
```

Should we run the progress loop in a separate execution stream?

margo/01_init/client.c

“Hello World” RPC

Simple “Hello World” RPC

Server

```
#include <margo.h>

static const int TOTAL_RPCS = 4;
static int num_rpcs = 0;

static void hello_world(hg_handle_t h);
DECLARE_MARGO_RPC_HANDLER(hello_world)

int main(int argc, char** argv)
{
    ...
    hg_id_t rpc_id = MARGO_REGISTER(mid, "hello", void, void, hello_world);
    margo_registered_disable_response(mid, rpc_id, HG_TRUE);

    margo_wait_for_finalize(mid);
    return 0;
}
```

margo/02_hello/server.c

Simple “Hello World” RPC

Server

All RPC handlers have this signature

Use this macro to declare the RPC handler

Register your RPC handler

This RPC doesn’t send a response back to the client

```
static const int TOT          = 4;
static int num_rpcs =

static void hello_world(hg_handle_t h);
DECLARE_MARGO_RPC_HANDLER(hello_world)

int main(int argc, char** argv)
{
    ...
    hg_id_t rpc_id = MARGO_REGISTER(mid, "hello", void, void, hello_world);
    margo_registered_disable_response(mid, rpc_id, HG_TRUE);

    margo_wait_for_finalize(mid);
    return 0;
}
```

Server

Simple “Hello World” RPC (cont’d)

```
static void hello_world(hg_handle_t h)
{
    hg_return_t ret;
    margo_instance_id mid = margo_hg_handle_get_instance(h);
    printf("Hello World!\n");
    num_rpcs += 1;

    ret = margo_destroy(h);
    assert(ret == HG_SUCCESS);

    if(num_rpcs == TOTAL_RPCS) {
        margo_finalize(mid);
    }
}
DEFINE_MARGO_RPC_HANDLER(hello_world)
```

Don’t forget to destroy the handle

The margo instance will be finalized after this RPC finishes, and `margo_wait_for_finalize` in main will return

Define the RPC handler

margo/02_hello/server.c

Simple “Hello World” RPC (cont’d)

Client

```c
#include <margo.h>

int main(int argc, char** argv)
{
    ...
    hg_id_t hello_rpc_id = MARGO_REGISTER(mid, "hello", void, void, NULL);
    margo_registered_disable_response(mid, hello_rpc_id, HG_TRUE);

    hg_addr_t svr_addr;
    ret = margo_addr_lookup(mid, "tcp://localhost:1234", &svr_addr);

    hg_handle_t handle;
    ret = margo_create(mid, svr_addr, hello_rpc_id, &handle);
    ret = margo_forward(handle, NULL);
    ret = margo_destroy(handle);

    ret = margo_addr_free(mid, svr_addr);
    ...
}
```

margo/02_hello/client.c

Simple “Hello World” RPC (cont’d)

Client

```
#include <margo.h>

int main(int
{
    ...
    hg_id_t hello_rpc_     = MARGO_REGISTER(mid, "hello", void, void, NULL);
    margo_registered_disable_response(mid, hello_rpc_id, HG_TRUE);

    hg_addr_t svr_addr;
    ret = margo_addr_lookup(mid, "tcp://localhost:1234", &svr_addr);

    hg_handle_t handle;
    ret = margo_create(mid, svr_addr, hello_rpc_id, &handle);
    ret = margo_forward(handle, NULL);
    ret = margo_destroy(handle);

    ret = margo_addr_free(mid, svr_addr);
    ...
}
```

This RPC doesn’t expect a response from the server

NULL used instead of function

Lookup the server’s address

Create, forward (send), and destroy the RPC

Sending arguments, returning values

Sending arguments, returning values

Internal header

```
#ifndef PARAM_H
#define PARAM_H

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

/* We use the Mercury macros to define the input
 * and output structures along with the serialization functions.
 */
MERCURY_GEN_PROC(sum_in_t,
        ((int32_t)(x))\
        ((int32_t)(y)))

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

#endif
```

Include the Mercury macros

Generate the definition of the `sum_in_t` structure and its serialization functions

margo/03_sum/types.h

Server

Sending arguments, returning values (cont'd)

```
#include <margo.h>
#include "types.h"

static void sum(hg_handle_t h);
DECLARE_MARGO_RPC_HANDLER(sum)

int main(int argc, char** argv) {
    ...
    hg_id_t rpc_id = MARGO_REGISTER(mid, "sum", sum_in_t, sum_out_t, sum);
    ...
}
```

Use the newly defined types when registering the RPC

margo/03_sum/server.c

Sending arguments, returning values (cont'd)

Server

```
static void sum(hg_handle_t h)
{
    hg_return_t ret;

    sum_in_t in;
    sum_out_t out;

    ret = margo_get_input(h, &in);

    out.ret = in.x + in.y;
    printf("Computed %d + %d = %d\n",in.x,in.y,out.ret);

    ret = margo_respond(h, &out);
    ret = margo_free_input(h, &in);
    ret = margo_destroy(h);
}
DEFINE_MARGO_RPC_HANDLER(sum)
```

Read the input sent by the client

Send a response with the output

Don't forget to free the input!

margo/03_sum/server.c

Client

Sending arguments, returning values (cont’d)

```
#include "types.h"

    ...
    hg_id_t sum_rpc_id = MARGO_REGISTER(mid, "sum", sum_in_t, sum_out_t, NULL);

    sum_in_t args;
    args = { .x = 42, .y = 58 };
    hg_handle_t h;
    margo_create(mid, svr_addr, sum_rpc_id, &h);
    margo_forward(h, &args);

    sum_out_t resp;
    margo_get_output(h, &resp);
    printf("Got response: %d+%d = %d\n", args.x, args.y, resp.ret);

    margo_free_output(h,&resp);
    margo_destroy(h);
    …
```

Pass the input to forward

Read the RPC’s output

Free the RPC’s output

margo/03_sum/client.c

Attaching data to RPC handlers

Attaching data to RPC handlers

Client

```
typedef struct {
...
} mydata;

...
mydata* data = malloc(...);
hg_id_t sum_rpc_id = MARGO_REGISTER(mid, "sum", sum_in_t, sum_out_t, NULL);
margo_register_data(mid, sum_rpc_id, (void*)data, free);
...

const struct hg_info* info = margo_get_info(handle);
mydata* data = (mydata*)margo_registered_data(mid, info->id);
```

Attach the pointer to the RPC

Give it the function to use to free the data (or NULL)

Inside the RPC handler, retrieve the pointer

Bulk data transfers

Bulk data transfers

Internal header

```
#ifndef PARAM_H
#define PARAM_H

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

MERCURY_GEN_PROC(sum_in_t,
        ((int32_t)(n))\
        ((hg_bulk_t)(bulk)))

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

#endif
```

We will send *n* integers using bulk data transfer

margo/04_bulk/types.h

Bulk data transfers (cont'd)

Client

```
#include "types.h"


    ...
    hg_id_t sum_rpc_id = MARGO_REGISTER(mid, "sum", sum_in_t, sum_out_t, NULL);

    sum_in_t args;

    int32_t values[10] = { 1,4,2,5,6,3,5,3,2,5 };
    hg_size_t size = 10*sizeof(int32_t);

    hg_bulk_t local_bulk;
    margo_bulk_create(mid, 1, (void**)&values, &size, HG_BULK_READ_ONLY, &local_bulk);

    sum_in_t args = { .n = 10, .bulk = local_bulk };
    ...
    margo_bulk_free(local_bulk);

    ...
```

Expose the array by creating a bulk handle

Don't forget to free the bulk handle

margo/04_bulk/client.c

Bulk data transfers (cont’d)

Server

```
#include "types.h"

static void sum(hg_handle_t h)
{
    hg_return_t ret;
    sum_in_t in;
    sum_out_t out;
    int32_t* values;
    hg_bulk_t local_bulk;

    margo_instance_id mid = margo_hg_handle_get_instance(h);

    const struct hg_info* info = margo_get_info(h);
    hg_addr_t client_addr = info->addr;

    ret = margo_get_input(h, &in);

    values = calloc(in.n, sizeof(*values));
    hg_size_t buf_size = in.n * sizeof(*values);
```

You can get the client’s address this way

Allocate a local buffer to receive the values from the client

margo/04_bulk/server.c

Bulk data transfers (cont'd)

Server

```
    ret = margo_bulk_create(mid, 1, (void**)&values, &buf_size,
            HG_BULK_WRITE_ONLY, &local_bulk);

    ret = margo_bulk_transfer(mid, HG_BULK_PULL, client_addr,
            in.bulk, 0, local_bulk, 0, buf_size);

    out.ret = 0;
    int i;
    for(i = 0; i < in.n; i++) {
        out.ret += values[i];
    }

    ret = margo_respond(h, &out);
    ret = margo_bulk_free(local_bulk);
    free(values);
    ret = margo_free_input(h, &in);
    ret = margo_destroy(h);
}
DEFINE_MARGO_RPC_HANDLER(sum)
```

Expose the local buffer

Pull data from the client

Free the bulk handle

margo/04_bulk/server.c

Non-blocking RPC

Non-blocking RPC

Client

```
sum_in_t args = { .x = 42, .y = 58 };

hg_handle_t h;
...
margo_request req;
margo_iforward(h, &args, &req);

printf("Waiting for reply...\n");

margo_wait(req);

sum_out_t resp;
margo_get_output(h, &resp);
```

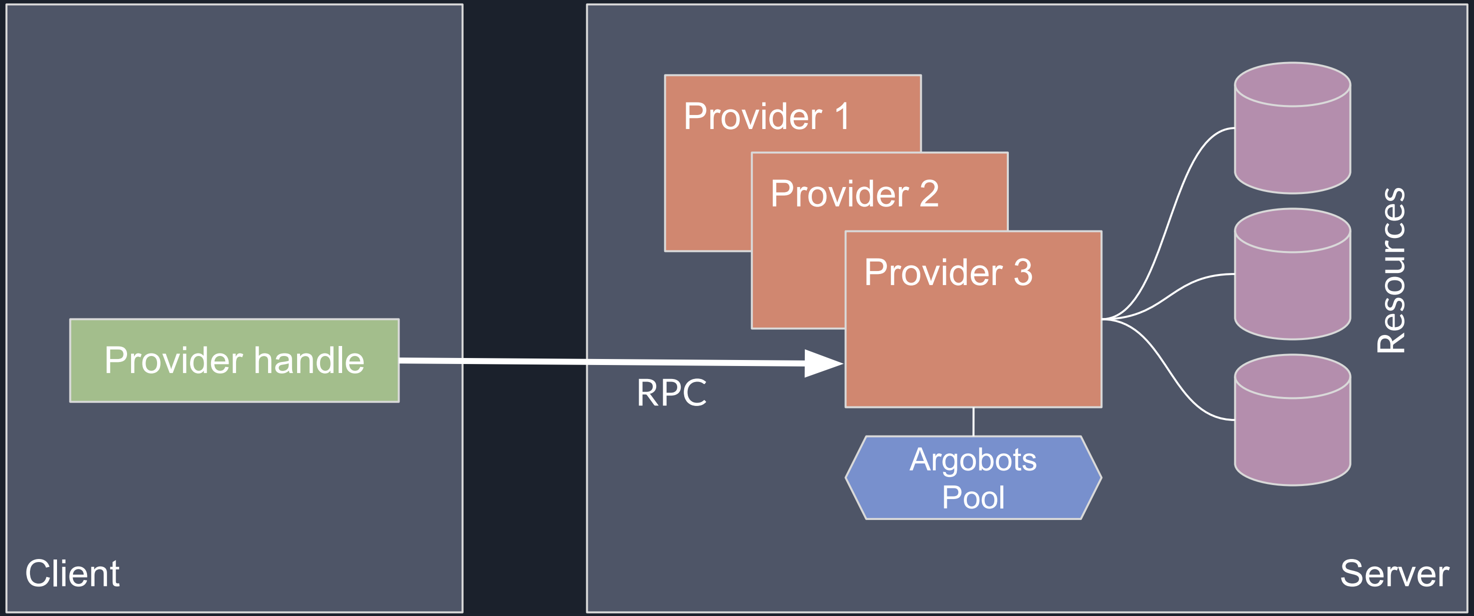
Use `margo_iforward` to send and return immediately

Wait for the operation to complete (this will also free the request)

margo/05_async/client.c

Developing providers

What is a provider?



Developing providers

Internal header

```
#ifndef PARAM_H
#define PARAM_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)))

#endif
```

Just a reminder

margo/06_provider/types.h

Developing providers (cont'd)

Public header

```
#ifndef __ALPHA_COMMON_H
#define __ALPHA_COMMON_H

#define ALPHA_SUCCESS  0
#define ALPHA_FAILURE -1

#endif
```

margo/06_provider/alpha-common.h

Developing providers (cont'd)

Client

```
typedef struct alpha_client* alpha_client_t;
#define ALPHA_CLIENT_NULL ((alpha_client_t)NULL)

typedef struct alpha_provider_handle *alpha_provider_handle_t;
#define ALPHA_PROVIDER_HANDLE_NULL ((alpha_provider_handle_t)NULL)

int alpha_client_init(margo_instance_id mid, alpha_client_t* client);

int alpha_client_finalize(alpha_client_t client);

int alpha_provider_handle_create(alpha_client_t client, hg_addr_t addr, uint16_t provider_id,
        alpha_provider_handle_t* handle);

int alpha_provider_handle_ref_incr(alpha_provider_handle_t handle);

int alpha_provider_handle_release(alpha_provider_handle_t handle);

int alpha_compute_sum(alpha_provider_handle_t handle, int32_t x, int32_t y, int32_t* result);
```

margo/06_provider/alpha-client.h

Developing providers (cont’d)

Client

```c
#include "types.h"
#include "alpha-client.h"

struct alpha_client {
   margo_instance_id mid;
   hg_id_t           sum_id;
   uint64_t          num_prov_hdl;
};

struct alpha_provider_handle {
    alpha_client_t client;
    hg_addr_t      addr;
    uint16_t       provider_id;
    uint64_t       refcount;
};
```

margo/06_provider/alpha-client.c

Developing providers (cont'd)

Client

```c
int alpha_client_init(margo_instance_id mid, alpha_client_t* client)
{
    int ret = ALPHA_SUCCESS;
    alpha_client_t c = (alpha_client_t)calloc(1, sizeof(*c));
    if(!c) return ALPHA_FAILURE;
    c->mid = mid;
    hg_bool_t flag;
    hg_id_t id;
    margo_registered_name(mid, "alpha_sum", &id, &flag);
    if(flag == HG_TRUE) {
        margo_registered_name(mid, "alpha_sum", &c->sum_id, &flag);
    } else {
        c->sum_id = MARGO_REGISTER(mid, "alpha_sum", sum_in_t, sum_out_t, NULL);
    }
    *client = c;
    return ALPHA_SUCCESS;
}
```

margo/06_provider/alpha-client.c

Client

Developing providers (cont'd)

```
int alpha_client_init(margo_instance_id mid, alpha_client_t* client)
{
    int ret = ALPHA_SUCCESS;
    alpha_client_t c = (alpha                              (*c));
    if(!c) return ALPHA_FAILU
    c->mid = mid;
    hg_bool_t flag;
    hg_id_t id;
    margo_registered_name(mid, "alpha_sum", &id, &flag);
    if(flag == HG_TRUE) {
        margo_registered_name(mid, "alpha_sum", &c->sum_id, &flag);
    } else {
        c->sum_id = MARGO_REGISTER(mid, "alpha_sum", sum_in_t, sum_out_t, NULL);
    }
    *client = c;
    return ALPHA_SUCCESS;
}
```

Check if the RPC is already registered

If it is, get the ID

Otherwise, register it

margo/06_provider/alpha-client.c

Developing providers (cont'd)

Client

```
int alpha_client_finalize(alpha_client_t client)
{
    if(client->num_prov_hdl != 0) {
        fprintf(stderr,
        "Warning: %d provider handles not released when alpha_client_finalize was called\n",
                client->num_prov_hdl);
    }
    free(client);
    return ALPHA_SUCCESS;
}
```

margo/06_provider/alpha-client.c

Developing providers (cont'd)

Client

```
int alpha_provider_handle_create(alpha_client_t client, hg_addr_t addr, uint16_t provider_id,
        alpha_provider_handle_t* handle)
{
    if(client == ALPHA_CLIENT_NULL) return ALPHA_FAILURE;
    alpha_provider_handle_t ph = (alpha_provider_handle_t)calloc(1, sizeof(*ph));
    if(!ph) return ALPHA_FAILURE;
    hg_return_t ret = margo_addr_dup(client->mid, addr, &(ph->addr));
    if(ret != HG_SUCCESS) {
        free(ph);
        return ALPHA_FAILURE;
    }
    ph->client      = client;
    ph->provider_id = provider_id;
    ph->refcount    = 1;
    client->num_prov_hdl += 1;
    *handle = ph;
    return ALPHA_SUCCESS;
}
```

Duplicate the address, so the caller can safely free the original

margo/06_provider/alpha-client.c

Developing providers (cont'd)

Client

```
int alpha_provider_handle_ref_incr(alpha_provider_handle_t handle)
{
    handle->refcount += 1;
    return ALPHA_SUCCESS;
}

int alpha_provider_handle_release(alpha_provider_handle_t handle)
{
    handle->refcount -= 1;
    if(handle->refcount == 0) {
        margo_addr_free(handle->client->mid, handle->addr);
        handle->client->num_prov_hdl -= 1;
        free(handle);
    }
    return ALPHA_SUCCESS;
}
```

Free the provider handle's underlying address

margo/06_provider/alpha-client.c

Developing providers (cont'd)

Client

```c
int alpha_compute_sum(alpha_provider_handle_t handle, int32_t x, int32_t y, int32_t* result)
{
    hg_handle_t   h;
    sum_in_t      in = { .x = x, .y = y };
    sum_out_t   out;
    hg_return_t ret;
    ret = margo_create(handle->client->mid, handle->addr, handle->client->sum_id, &h);
    ret = margo_provider_forward(handle->provider_id, h, &in);
    ret = margo_get_output(h, &out);
    *result = out.ret;
    margo_free_output(h, &out);
    margo_destroy(h);
    return ALPHA_SUCCESS;
}
```

Use `margo_provider_forward` instead of `margo_forward`

margo/06_provider/alpha-client.c

Developing providers (cont’d)

```
#define ALPHA_ABT_POOL_DEFAULT ABT_POOL_NULL

typedef struct alpha_provider* alpha_provider_t;
#define ALPHA_PROVIDER_NULL ((alpha_provider_t)NULL)
#define ALPHA_PROVIDER_IGNORE ((alpha_provider_t*)NULL)

int alpha_provider_register(
        margo_instance_id mid,
        uint16_t provider_id,
        ABT_pool pool,
        alpha_provider_t* provider);

int alpha_provider_destroy(
        alpha_provider_t provider);
```

margo/06_provider/alpha-server.h

Server

Developing providers (cont'd)

```
#include "alpha-server.h"
#include "types.h"

struct alpha_provider {
    margo_instance_id mid;
    hg_id_t sum_id;
    /* other provider-specific data */
};

static void alpha_finalize_provider(void* p);

DECLARE_MARGO_RPC_HANDLER(alpha_sum_ult);
static void alpha_sum_ult(hg_handle_t h);
/* add other RPC declarations here */
```

This function will be called either by `alpha_provider_destroy` or when the Margo instance is finalized

Server

Developing providers (cont'd)

```c
int alpha_provider_register(margo_instance_id mid, uint16_t provider_id, ABT_pool pool,
        alpha_provider_t* provider)
{
    alpha_provider_t p;
    hg_id_t id;
    hg_bool_t flag;

    flag = margo_is_listening(mid);
    if(flag == HG_FALSE) {
        fprintf(stderr, "alpha_provider_register(): margo instance is not a server.");
        return ALPHA_FAILURE;
    }

    margo_provider_registered_name(mid, "alpha_sum", provider_id, &id, &flag);
    if(flag == HG_TRUE) {
        fprintf(stderr, "a provider with the same provider id already exists.\n");
        return ALPHA_FAILURE;
    }
```

No point in creating a provider if it cannot receive anything

Check if RPC already registered for this provider id

margo/06_provider/alpha-server.c

Developing providers (cont'd)

Server

```
    p = (alpha_provider_t)calloc(1, sizeof(*p));
    if(p == NULL)
        return ALPHA_FAILURE;

    p->mid = mid;

    id = MARGO_REGISTER_PROVIDER(mid, "alpha_sum",
            sum_in_t, sum_out_t,
            alpha_sum_ult, provider_id, pool);
    margo_register_data(mid, id, (void*)p, NULL);
    p->sum_id = id;
    /* add other RPC registration here */

    margo_provider_push_finalize_callback(mid, p, &alpha_finalize_provider, p);

    *provider = p;
    return ALPHA_SUCCESS;
}
```

margo/06_provider/alpha-server.c

Developing providers (cont'd)

Server

```
    p = (alpha_provider_t)calloc(1, sizeof(*p));
    if(p == NULL)
        return ALPHA_FAILURE;

    p->mid = mid;

    id = MARGO_REGISTER_PROVIDER(mid, "alpha_sum",
            sum_in_t, sum_out_t,
            alpha_sum_ult, provider_id, pool);
    margo_register_data(mid, id, (void*)p, NULL);
    p->sum_id = id;
    /* add other RPC registration here */

    margo_provider_push_finalize_callback(mid, p, &alpha_finalize_provider, p);

    *provider = p;
    return ALPHA_SUCCESS;
}
```

Use `MARGO_REGISTER_PROVIDER` instead of `MARGO_REGISTER`

Pass an Argobots pool to run RPC

Attach a pointer to the provider to the RPC

Tell Margo to destroy the provider if the Margo instance if finalized

margo/06_provider/alpha-server.c

Developing providers (cont'd)

Server

```
static void alpha_finalize_provider(void* p)
{
    alpha_provider_t provider = (alpha_provider_t)p;
    margo_deregister(provider->mid, provider->sum_id);
    /* deregister other RPC ids ... */
    free(provider);
}

int alpha_provider_destroy(
        alpha_provider_t provider)
{
    /* pop the finalize callback */
    margo_provider_pop_finalize_callback(provider->mid, provider);
    /* call the callback */
    alpha_finalize_provider(provider);

    return ALPHA_SUCCESS;
}
```

Deregister the RPC

Tell Margo there is no need anymore to call the finalize callback

margo/06_provider/alpha-server.c

Developing providers (cont'd)

Server

```
static void alpha_sum_ult(hg_handle_t h)
{
    hg_return_t ret;
    sum_in_t     in;
    sum_out_t   out;

    margo_instance_id mid = margo_hg_handle_get_instance(h);
    const struct hg_info* info = margo_get_info(h);
    alpha_provider_t provider = (alpha_provider_t)margo_registered_data(mid, info->id);

    ret = margo_get_input(h, &in);

    out.ret = in.x + in.y;
    printf("Computed %d + %d = %d\n",in.x,in.y,out.ret);

    ret = margo_respond(h, &out);
    ret = margo_free_input(h, &in);
}
DEFINE_MARGO_RPC_HANDLER(alpha_sum_ult)
```

This is how to retrieve the pointer to the provider that we have attached earlier

margo/06_provider/alpha-server.c

Getting started with Thallium

Where to find the material

- https://mochi.readthedocs.io
- https://xgitlab.cels.anl.gov/sds/mochi-doc

```
git clone https://xgitlab.cels.anl.gov/sds/mochi-doc.git
cd mochi-doc/code/thallium
```

Initializing Thallium

Initializing Thallium

Server

```
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {

    tl::engine myEngine("tcp", THALLIUM_SERVER_MODE);
    std::cout << "Server running at address " << myEngine.self() << std::endl;

    return 0;
}
```

The engine is the core class running Mercury progress

You may add two more parameters:

- `bool use_progress_thread`
- `int num_rpc_threads`

The engine's destructor will block until finalized

thallium/01_init/server.cpp

Client

Initializing Thallium (cont'd)

```
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {

    tl::engine myEngine("tcp", THALLIUM_CLIENT_MODE);

    return 0;
}
```

When initialized as a client, the engine will finalize itself when its destructor is called, instead of blocking

thallium/01_init/client.cpp

“Hello World” RPC

“Hello world” RPC

Server

```cpp
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

void hello(const tl::request& req) {
    std::cout << "Hello World!" << std::endl;
}

int main(int argc, char** argv) {

    tl::engine myEngine("tcp://127.0.0.1:1234", THALLIUM_SERVER_MODE);
    myEngine.define("hello", hello).disable_response();

    return 0;
}
```

RPC handlers must take a reference to a request object

Defines the “hello” RPC

Tell thallium this RPC doesn’t send a response

thallium/02_hello/server.cpp

“Hello world” RPC (cont’d)

Client

```
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {

    tl::engine myEngine("tcp", THALLIUM_CLIENT_MODE);
    tl::remote_procedure hello = myEngine.define("hello").disable_response();
    tl::endpoint server = myEngine.lookup("tcp://127.0.0.1:1234");
    hello.on(server)();

    return 0;
}
```

Define the RPC without the function pointer

Lookup the server’s address

Call the RPC on the server

thallium/02_hello/client.cpp

Sending arguments, returning values

Sending arguments, returning values

Server

```
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

void sum(const tl::request& req, int x, int y) {
    std::cout << "Computing " << x << "+" << y << std::endl;
    req.respond(x+y);
}

int main(int argc, char** argv) {

    tl::engine myEngine("tcp://127.0.0.1:1234", THALLIUM_SERVER_MODE);
    myEngine.define("sum", sum);

    return 0;
}
```

RPC handlers may take additional arguments

And send a response

thallium/03_args/server.cpp

Sending arguments, returning values (cont'd)

Client

```
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {

    tl::engine myEngine("tcp", THALLIUM_CLIENT_MODE);
    tl::remote_procedure sum = myEngine.define("sum");
    tl::endpoint server = myEngine.lookup("tcp://127.0.0.1:1234");
    int ret = sum.on(server)(42,63);
    std::cout << "Server answered " << ret << std::endl;

    return 0;
}
```

thallium/03_args/client.cpp

Sending arguments, returning values (cont'd)

Client

```
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {

    tl::engine myEngine("tcp", THALLIUM_      _MODE);
    tl::remote_procedure sum = myEngin  efine("s  um");
    tl::endpoint server = myEngine.l  okup("tcp://12     0.1:1234");
    int ret = sum.on(server)(42,63);
    std::cou  << "Server answered " << ret << std::endl;

    return 0;
}
```

We should now call the RPC with the extra arguments

And receive the result

Warning: make sure your types match between RPC handlers and RPC calls!

thallium/03_args/client.cpp

Registering lambdas as RPC handlers

Registering lambdas as RPC handlers

Server

```cpp
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {

    tl::engine myEngine("tcp://127.0.0.1:1234", THALLIUM_SERVER_MODE);

    std::function<void(const tl::request&, int, int)> sum =
        [](const tl::request& req, int x, int y) {
            std::cout << "Computing " << x << "+" << y << std::endl;
            req.respond(x+y);
        };

    myEngine.define("sum", sum);

    return 0;
}
```

thallium/04_lambdas/server.cpp

Non-blocking RPCs

Non-blocking RPCs

Client

```
tl::engine myEngine("tcp", THALLIUM_CLIENT_MODE);
tl::remote_procedure sum = myEngine.define("sum");
tl::endpoint server = myEngine.lookup("tcp://127.0.0.1:1234");
auto request = sum.on(server).async(42,63);
// do something else ...
// check if request completed
bool completed =  request.received();
// ...
// actually wait on the request and get the result out of it
int ret = request.wait();
std::cout << "Server answered " << ret << std::endl;
```

thallium/14_async/client.cpp

Non-blocking RPCs

Client

```cpp
tl::engine myEngine("tcp", THALLIUM_CLIENT_MODE);
tl::remote_procedure sum = myEngine.define("sum");
tl::endpoint server = myEngine.lookup("tcp://127.0.0.1:1234");
auto request = sum.on(server).async(42,63);
// do something else ...
// check if request completed
bool completed =  request.received();
// ...
// actually wait on the request and get the resu
int ret = request.wait();
std::cout << "Server answered " << ret << std::endl;
```

Non-blocking test for completion

`wait()` is required even if `received()` returned `true`

thallium/14_async/client.cpp

Properly finalizing a server

Server

Properly finalizing a server

```
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {

    tl::engine myEngine("tcp://127.0.0.1:1234", THALLIUM_SERVER_MODE);

    std::function<void(const tl::request&, int, int)> sum =
        [&myEngine](const tl::request& req, int x, int y) {
            std::cout << "Computing " << x << "+" << y << std::endl;
            req.respond(x+y);
            myEngine.finalize();
        };
    myEngine.define("sum", sum);

    return 0;
}
```

`finalize()` can be called from an RPC handler

thallium/05_stop/server.cpp

Sending/receiving STL data structures

All the C++14 data structures are supported

- Any basic type (`int`, `float`,...)
- `std::array<T>`
- `std::complex<T>`
- `std::deque<T>`
- `std::forward_list<T>`
- `std::list<T>`
- `std::map<K,V>`
- `std::multimap<K,V>`
- `std::multiset<T>`
- `std::pair<U,V>`
- `std::set<T>`
- `std::string`
- `std::tuple<T...>`
- `std::unordered_map<K,V>`
- `std::unordered_multimap<K,V>`
- `std::unordered_multiset<T>`
- `std::unordered_set<T>`
- `std::vector<T>`

Any composition of those types can be serialized by Thallium as well, for example
`std::vector<std:tuple<std::pair<int,double>,std::list<int>>>`

Example: sending/receiving a `std::string`

Server

```
#include <string>
#include <iostream>
#include <thallium.hpp>
#include <thallium/serialization/stl/string.hpp>

namespace tl = thallium;

void hello(const tl::request& req, const std::string& name) {
    std::cout << "Hello " << name << std::endl;
}

int main(int argc, char** argv) {

    tl::engine myEngine("tcp://127.0.0.1:1234", THALLIUM_SERVER_MODE);
    myEngine.define("hello", hello).disable_response();

    return 0;
}
```

Don't forget to include the right header!

thallium/06_stl/server.cpp

Client

Example: sending/receiving a `std::string`

```cpp
#include <string>
#include <thallium.hpp>
#include <thallium/serialization/stl/string.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {

    tl::engine myEngine("tcp", THALLIUM_CLIENT_MODE);
    tl::remote_procedure hello = myEngine.define("hello").disable_response();
    tl::endpoint server = myEngine.lookup("tcp://127.0.0.1:1234");
    std::string name = "Matthieu";
    hello.on(server)(name);

    return 0;
}
```

Don't forget to include the right header!

`hello.on(server)("Matthieu")` would not work because template type deduction would infer a `const char*` instead of an `std::string`

thallium/06_stl/client.cpp

Sending/receiving instances of custom class

Example: a `point` class

```
class point {

    private:

        double x, y, z;

    public:

        point(double a=0.0, double b=0.0, double c=0.0)
        : x(a), y(b), z(c) {}

        template<typename A>
        void serialize(A& ar) {
             ar & x;
             ar & y;
             ar & z;
        }
};
```

margo/02_hello/server.c

Other serialization methods

- template “`serialize`” member function
- template “`save`” and “load” member functions
- template “`serialize`” external function
- template “`save`” and “`load`” external functions
- “`read`” and “`write`” methods on archive type to access raw data

See https://mochi.readthedocs.io for more information.

Bulk data transfers

Bulk data transfers

Client

```
tl::engine myEngine("tcp", MARGO_CLIENT_MODE);
tl::remote_procedure remote_do_rdma = myEngine.define("do_rdma").disable_response();
tl::endpoint server_endpoint = myEngine.lookup("tcp://127.0.0.1:1234");

std::string buffer = "Matthieu";
std::vector<std::pair<void*,std::size_t>> segments(1);
segments[0].first  = (void*)(&buffer[0]);
segments[0].second = buffer.size();

tl::bulk myBulk = myEngine.expose(segments, tl::bulk_mode::read_only);

remote_do_rdma.on(server_endpoint)(myBulk);
```

Expose one or more segments by using a vector of <pointer, size> pairs

The server is only going to read from this memory

thallium/08_rdma/client.cpp

Bulk data transfers (cont’d)

Server

```
tl::engine myEngine("tcp://127.0.0.1:1234", THALLIUM_SERVER_MODE);

std::function<void(const tl::request&, tl::bulk&)> f =
    [&myEngine](const tl::request& req, tl::bulk& b) {
        tl::endpoint ep = req.get_endpoint();
        std::string s(6,0);
        std::vector<std::pair<void*,std::size_t>> segments(1);
        segments[0].first  = (void*)(s.data());
        segments[0].second = s.size();
        tl::bulk local = myEngine.expose(segments, tl::bulk_mode::write_only);
        b.on(ep) >> local;
        std::cout << "Server received bulk: ";
        std::cout << s;
        std::cout << std::endl;
    };

myEngine.define("do_rdma",f).disable_response();
```

Bulk data transfers (cont'd)

Server

```
tl::engine myEngine("tcp://127.0.0.1:1234", THALLIUM_SERVER_MODE);

std::function<void(const tl::request&, tl::bulk&)> f =
    [&myEngine](const tl::request& req, tl::bulk& b) {
        tl::endpoint ep = req.get_endpoint();
        std::string s(6,0);
        std::vector<std::pair<void*,std::size_t>> segments(1);
        segments[0].first  = (void*)(s.data());
        segments[0].second = s.size();
        tl::bulk local = myEngine.expose(segments, tl::bulk_mode::write_only);
        b.on(ep) >> local;
        std::cout << "Server received bulk: ";
        std::cout << s;
        std::cout << std::endl;
    };

myEngine.define("do_rdma",f).disable_response();
```

Stream operators are defined to represent "push" and "pull" operations

thallium/08_rdma/server.cpp

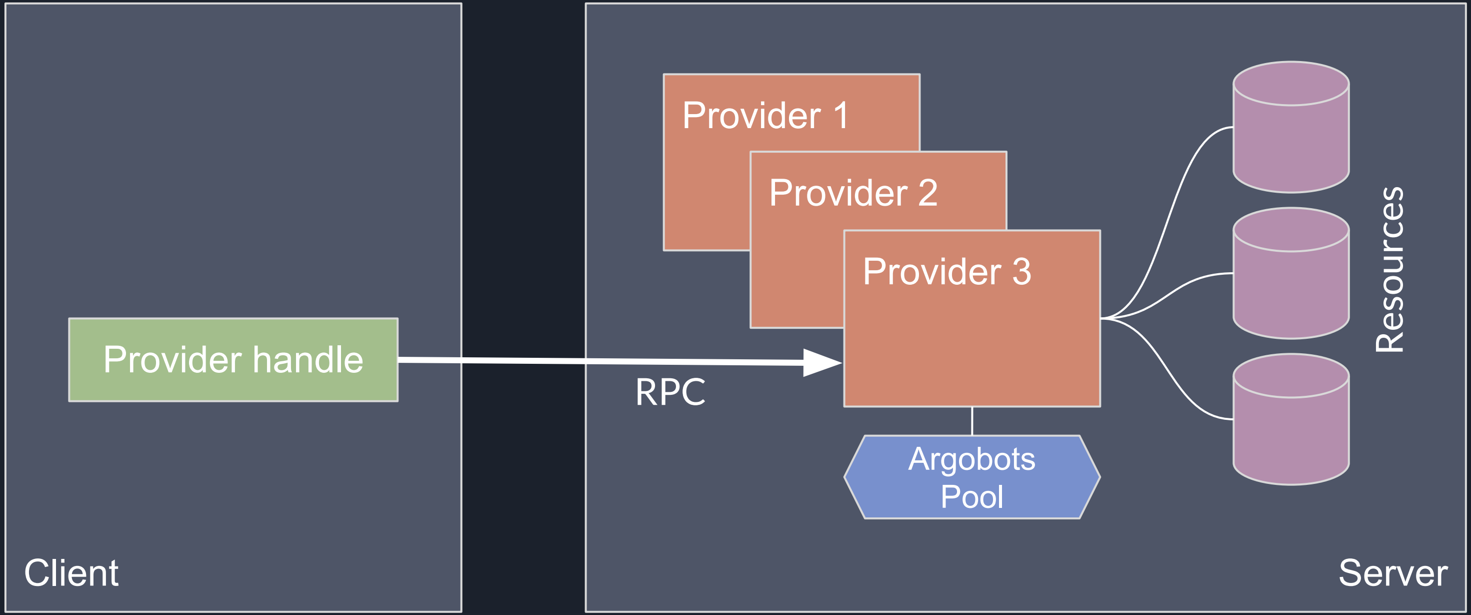
Bulk data transfers (cont’d)

Transferring subsegments of bulk regions

```
myRemoteBulk(3,45).on(myRemoteProcess) >> myLocalBulk(13,45);
```

Working in terms of providers

What is a provider?



Server

Working in terms of providers

```
#include <iostream>
#include <thallium.hpp>
#include <thallium/serialization/stl/string.hpp>

namespace tl = thallium;

class my_sum_provider : public tl::provider<my_sum_provider> {

    private:

    void sum(const tl::request& req, int x, int y) {
        std::cout << "Computing " << x << "+" << y << std::endl;
        req.respond(x+y);
    }

    int print(const std::string& word) {
        std::cout << "Printing " << word << std::endl;
        return word.size();
    }
```

Providers must inherit from `tl::provider`

You may omit the request argument if it's not used

By default, the non-void return value is treated as a response

thallium/09_providers/server.cpp

Working in terms of providers (cont'd)

Server

```
    public:

    my_sum_provider(tl::engine& e, uint16_t provider_id=1)
    : tl::provider<my_sum_provider>(e, provider_id) {
        define("sum", &my_sum_provider::sum);
        define("print", &my_sum_provider::print, tl::ignore_return_value());
    }

    ~my_sum_provider() {
        wait_for_finalize();
    }
};
```

thallium/09_providers/server.cpp

Working in terms of providers (cont'd)

Server

```
    public:

    my_sum_provider(tl::engine& e, uint16_t
    : tl::provider<my_sum_provider>(e, provid  _id) {
        define("sum", &my_sum_provider::sum);
        define("print", &my_sum_provider::print, tl::ignore_return_value());
    }

    ~my_sum_provider() {
        wait_for_finalize();
    }
};
```

Call define from the based class

Use this to indicate that although `print()` returns an `int`, we don't want to send this `int` back as a response

thallium/09_providers/server.cpp

Working in terms of providers (cont'd)

Server

```
int main(int argc, char** argv) {

    uint16_t provider_id = 22;
    tl::engine myEngine("tcp", THALLIUM_SERVER_MODE);
    std::cout << "Server running at address " << myEngine.self()
              << " with provider id " << provider_id << std::endl;
    my_sum_provider myProvider(myEngine, provider_id);

    return 0;
}
```

thallium/09_providers/server.cpp

Working in terms of providers (cont’d)

Client

```
tl::engine myEngine("tcp", THALLIUM_CLIENT_MODE);
tl::remote_procedure sum   = myEngine.define("sum");
tl::remote_procedure print = myEngine.define("print").disable_response();
tl::endpoint server = myEngine.lookup(argv[1]);
uint16_t provider_id = atoi(argv[2]);
tl::provider_handle ph(server, provider_id);
int ret = sum.on(ph)(42,63);
std::cout << "(sum) Server answered " << ret << std::endl;
std::string name("Matthieu");
print.on(ph)(name);
```

RPCs on clients should still be defined in the engine directly

A provider handle consists of an endpoint and a provider id

Call the RPC on a specific provider handle

thallium/09_providers/client.cpp

Properly finalizing providers

Properly finalizing providers

```
class my_sum_provider : public tl::provider<my_sum_provider> {
    ...
    tl::remote_procedure m_sum;
    ...
    my_sum_provider(tl::engine& e, uint16_t provider_id=1)
    : tl::provider<my_sum_provider>(e, provider_id)
      // keep the RPCs in remote_procedure objects so we can deregister them.
    , m_sum(define("sum", &my_sum_provider::sum))
    {
        // setup a finalization callback for this provider
        get_engine().push_finalize_callback(this, [p=this]() { delete p; });
    }
    public:
    // this factory method and the private constructor prevent users
    // from putting an instance  of my_sum_provider on  the stack.
    static my_sum_provider* create(tl::engine& e, uint16_t provider_id=1) {
        return new my_sum_provider(e, provider_id);
    }
```

thallium/15_finalize/server.cpp

Properly finalizing providers

```
    ~my_sum_provider() {
        m_sum.deregister();
        get_engine().pop_finalize_callback(this);
    }
};

int main(int argc, char** argv) {
    tl::engine myEngine("tcp", THALLIUM_SERVER_MODE);
    myEngine.enable_remote_shutdown();

    my_sum_provider* myProvider22 = my_sum_provider::create(myEngine, 22);
    my_sum_provider* myProvider23 = my_sum_provider::create(myEngine, 23);

    return 0;
}
```

thallium/15_finalize/server.cpp

Argobots wrappers

Argobots wrappers

● barrier	➢ ABT_barrier
● condition_variable	➢ ABT_cond
● eventual	➢ ABT_eventual
● future	➢ ABT_future
● mutex	➢ ABT_mutex
● pool	➢ ABT_pool
● recursive_mutex	➢ ABT_recursive_mutex
● rwlock	➢ ABT_rwlock
● scheduler	➢ ABT_scheduler
● task	➢ ABT_task
● thread	➢ ABT_thread
● timer	➢ ABT_timer
● xstream	➢ ABT_xstream
● xstream_barrier	➢ ABT_xstream_barrier

Common pitfalls

Common pitfalls

- Blocking MPI primitives (`MPI_Recv`, `MPI_Barrier`, etc.) don't yield to the progress loop. Use a dedicated progress thread, or a call MPI functions from a dedicated ES.
- Other threading frameworks (pthreads, OpenMP, etc.) don't play well with Argobots threads. Mix with caution.
- Don't use `std::mutex`, remember to use `thallium::mutex.`
- Don't use `std::thread`, or `std::async`. Use corresponding Argobots equivalents.
- All the Argobots and Mercury objects should be released before margo/thallium is finalized.