Mercury: RPC for High-Performance Computing

Jerome Soumagne
The HDF Group

June 23, 2017
Remote Procedure Call (RPC)

- Allow local calls to be executed on remote resources
- Already widely used to support distributed services
  - Google Protocol Buffers, etc

Typical HPC applications are SPMD

- No need for RPC: control flow implicit on all nodes
- A series of SPMD programs sequentially produce & analyze data

Distributed HPC workflow

- Nodes/systems dedicated to specific task
- Multiple SPMD applications/jobs execute concurrently and interact

Importance of RPC growing

- Compute nodes with minimal/non-standard environment
- Heterogeneous systems (node-specific resources)
- More "service-oriented" and more complex applications
- Workflows and in-transit instead of sequences of SPMD
RPC and High-Performance Computing

Remote Procedure Call (RPC)
- Allow local calls to be executed on remote resources
- Already widely used to support distributed services
  - Google Protocol Buffers, etc
RPC and High-Performance Computing

Remote Procedure Call (RPC)
- Allow local calls to be executed on remote resources
- Already widely used to support distributed services
  - Google Protocol Buffers, etc

Typical HPC applications are *SPMD*
- No need for RPC: control flow implicit on all nodes
- A series of SPMD programs sequentially produce & analyze data
Remote Procedure Call (RPC)
- Allow local calls to be executed on remote resources
- Already widely used to support distributed services
  - Google Protocol Buffers, etc

Typical HPC applications are SPMD
- No need for RPC: control flow implicit on all nodes
- A series of SPMD programs sequentially produce & analyze data

Distributed HPC workflow
- Nodes/systems dedicated to specific task
- Multiple SPMD applications/jobs execute concurrently and interact
Remote Procedure Call (RPC)
- Allow local calls to be executed on remote resources
- Already widely used to support distributed services
  - Google Protocol Buffers, etc

Typical HPC applications are SPMD
- No need for RPC: control flow implicit on all nodes
- A series of SPMD programs sequentially produce & analyze data

Distributed HPC workflow
- Nodes/systems dedicated to specific task
- Multiple SPMD applications/jobs execute concurrently and interact

Importance of RPC growing
- Compute nodes with minimal/non-standard environment
- Heterogeneous systems (node-specific resources)
- More “service-oriented” and more complex applications
- Workflows and in-transit instead of sequences of SPMD
Objective
Create a reusable RPC library for use in HPC that can serve as a basis for services such as storage systems, I/O forwarding, analysis frameworks and other forms of inter-application communication.

Why not reuse existing RPC frameworks?
- Do not support efficient large data transfers or asynchronous calls
- Mostly built on top of TCP/IP protocols
  - Need support for native transport
  - Need to be easy to port to new systems

Similar previous approaches with some differences
- I/O Forwarding Scalability Layer (IOFSL) – ANL
- NEtwork Scalable Service Interface (Nessie) – Sandia
- Lustre RPC – Intel
Objective
Create a reusable RPC library for use in HPC that can serve as a basis for services such as storage systems, I/O forwarding, analysis frameworks and other forms of inter-application communication.
Objective
Create a reusable RPC library for use in HPC that can serve as a basis for services such as storage systems, I/O forwarding, analysis frameworks and other forms of inter-application communication

- Why not reuse existing RPC frameworks?
  - Do not support *efficient* large data transfers or asynchronous calls
  - Mostly built on top of TCP/IP protocols
    - Need support for native transport
    - Need to be easy to port to new systems
Objective
Create a reusable RPC library for use in HPC that can serve as a basis for services such as storage systems, I/O forwarding, analysis frameworks and other forms of inter-application communication

■ Why not reuse existing RPC frameworks?
  – Do not support efficient large data transfers or asynchronous calls
  – Mostly built on top of TCP/IP protocols
    ▶ Need support for native transport
    ▶ Need to be easy to port to new systems

■ Similar previous approaches with some differences
  – I/O Forwarding Scalability Layer (IOFSL) – ANL
  – NEtwork Scalable Service Interface (Nessie) – Sandia
  – Lustre RPC – Intel
Overview

- Designed to be both easily integrated and extended
  - “Client” / “Server” notions abstracted
  - “Origin” / “Target” used instead

- Basis for accessing and enabling resilient services
  - Ability to reclaim resources after failure is imperative
Overview

- Designed to be both easily integrated and extended
  - “Client” / “Server” notions abstracted
    - (Server may also act as a client and vice versa)
  - “Origin” / “Target” used instead

Compute Nodes, origin $c_1$ has target $s_2$

Service Nodes (e.g., storage, visualization, etc), $s_1$ and $s_3$ are targets of $s_2$
Overview

- Designed to be both easily integrated and extended
  - “Client” / “Server” notions abstracted
    ▶ (Server may also act as a client and vice versa)
  - “Origin” / “Target” used instead

- Basis for accessing and enabling resilient services
  - Ability to reclaim resources after failure is imperative

![Diagram showing compute nodes and service nodes connected](image-url)
Overview

- Function arguments / metadata transferred with RPC request
  - Two-sided model with unexpected / expected messaging
  - Message size limited to a few kilobytes (low-latency)
- Bulk data transferred using separate and dedicated API
  - One-sided model that exposes RMA semantics (high-bandwidth)

Network Abstraction Layer
- Allows definition of multiple network plugins
  - MPI and BMI plugins first plugins
  - Shared-memory plugin (mmap + CMA, supported on Cray w/CLE6)
  - CCI plugin contributed by ORNL
  - Libfabric plugin contributed by Intel (support for Cray GNI)
Overview

- Function arguments / metadata transferred with RPC request
  - Two-sided model with unexpected / expected messaging
  - Message size limited to a few kilobytes (low-latency)

Network Abstraction Layer

- Allows definition of multiple network plugins
  - MPI and BMI plugins first plugins
  - Shared-memory plugin (mmap + CMA, supported on Cray w/CLE6)
  - CCI plugin contributed by ORNL
  - Libfabric plugin contributed by Intel (support for Cray GNI)
Overview

- Function arguments / metadata transferred with RPC request
  - Two-sided model with unexpected / expected messaging
  - Message size limited to a few kilobytes (low-latency)
- Bulk data transferred using separate and dedicated API
  - One-sided model that exposes RMA semantics (high-bandwidth)
Overview

- Function arguments / metadata transferred with RPC request
  - Two-sided model with unexpected / expected messaging
  - Message size limited to a few kilobytes (low-latency)

- Bulk data transferred using separate and dedicated API
  - One-sided model that exposes RMA semantics (high-bandwidth)

- Network Abstraction Layer
  - Allows definition of multiple network plugins
    - MPI and BMI plugins first plugins
    - Shared-memory plugin (mmap + CMA, supported on Cray w/CLE6)
    - CCI plugin contributed by ORNL
    - Libfabric plugin contributed by Intel (support for Cray GNI)
Remote Procedure Call

- Mechanism used to send an RPC request (may also ignore response)

<table>
<thead>
<tr>
<th>id₁</th>
<th>...</th>
<th>idₙ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Origin

Target
Remote Procedure Call

- Mechanism used to send an RPC request (may also ignore response)

1. Register call and get request id

$$\text{id}_1 \ldots \text{id}_N$$

<table>
<thead>
<tr>
<th>Origin</th>
</tr>
</thead>
</table>

2. (Pre-post receive for target response) Post unexpected send with request id and serialized parameters

3. Post receive for unexpected request / Make progress

4. Execute call

4. Make progress

<table>
<thead>
<tr>
<th>Target</th>
</tr>
</thead>
</table>

$$\text{id}_1 \ldots \text{id}_N$$
Remote Procedure Call

- Mechanism used to send an RPC request (may also ignore response)

1. Register call and get request id
2. (Pre-post receive for target response) Post unexpected send with request id and serialized parameters
3. Execute call
4. Make progress (Post send with serialized response)

2. Post receive for unexpected request / Make progress
Remote Procedure Call

- Mechanism used to send an RPC request (may also ignore response)

1. Register call and get request id
2. Post receive for unexpected send with request id and serialized parameters
3. Execute call
4. Post send with serialized response

### Diagram

- **Origin**
  - id$_1$ ... id$_N$

- **Target**
  - id$_1$ ... id$_N$

3. Execute call
Remote Procedure Call

- Mechanism used to send an RPC request (may also ignore response)

1. Register call and get request id
2. Post receive for unexpected send with request id and serialized parameters
3. Post receive for unexpected request / Make progress
4. Execute call
4. Make progress

(4. Post send with serialized response)
Progress Model

- Callback-based model with completion queue
- Explicit progress with `HG_Progress()` and `HG_Trigger()`
  - Allows user to create workflow
  - No need to have an explicit `wait` call (shim layers possible)
  - Facilitate operation scheduling, multi-threaded execution and cancellation!

```c
do {
    unsigned int actual_count = 0;

    do {
        ret = HG_Trigger(context, 0, 1, &actual_count);
    } while ((ret == HG_SUCCESS) && actual_count);

    if (done)
        break;

    ret = HG_Progress(context, HG_MAX_IDLE_TIME);
} while (ret == HG_SUCCESS);
```
Remote Procedure Call: Example

Origin snippet (Callback model):

```c
open_in_t in_struct;

/* Initialize the interface and get target address */
hg_class = HG_Init("ofi+tcp://eth0:22222", HG_FALSE);
hg_context = HG_Context_create(hg_class);
[...]
HG_Addr_lookup_wait(hg_context, target_name, &target_addr);

/* Register RPC call */
rpc_id = MERCURY_REGISTER(hg_class, "open", open_in_t, open_out_t);

/* Set input parameters */
in_struct.in_param0 = in_param0;

/* Create RPC request */
HG_Create(hg_context, target_addr, rpc_id, &hg_handle);

/* Send RPC request */
HG_Forward(hg_handle, rpc_done_cb, &rpc_done_args, &in_struct);

/* Make progress */
[...]
```
Remote Procedure Call: Example

- Origin snippet (next):

```c
hg_return_t
rpc_done_cb(const struct hg_cb_info *callback_info)
{
    open_out_t out_struct;

    /* Get output */
    HG_Get_output(callback_info->handle, &out_struct);

    /* Get output parameters */
    ret = out_struct.ret;
    out_param0 = out_struct.out_param0;

    /* Free output */
    HG_Free_output(callback_info->handle, &out_struct);

    return HG_SUCCESS;
}
```

- Cancellation: HG_Cancel() on handle
  - Callback still triggered (canceled = completion)
Remote Procedure Call: Example

- Target snippet (main loop):

```c
int main(int argc, void *argv[])
{
    /* Initialize the interface and listen */
    hg_class = HG_Init("ofi+tcp://eth0:22222", HG_TRUE);
    [...]

    /* Register RPC call */
    MERCURY_REGISTER(hg_class, "open", open_in_t, open_out_t, open_rpc_cb);

    /* Make progress */
    [...]

    /* Finalize the interface */
    [...]
}
```
Remote Procedure Call: Example

- Target snippet (RPC callback):

```c
hg_return_t
open_rpc_cb(hg_handle_t handle)
{
  open_in_t in_struct;
  open_out_t out_struct;

  /* Get input */
  HG_Get_input(handle, &in_struct);
  in_param0 = in_struct.in_param0;

  /* Execute call */
  out_param0 = open(in_param0, ...);

  /* Set output */
  open_out_struct.out_param0 = out_param0;

  /* Send response back to origin */
  HG_Respond(handle, NULL, NULL, &out_struct);

  /* Free input and destroy handle */
  HG_Free_input(handle, &in_struct);
  HG_Destroy(handle);

  return HG_SUCCESS;
}
```
Bulk Data Transfers

Definition

- Bulk Data: Variable length data that is (or could be) too large to send eagerly and might need special processing.
  - Transfer controlled by target (better flow control)
  - Memory buffer(s) abstracted by handle
  - Handle must be serialized and exchanged using other means

Origin

1. Register local memory segment and get handle

Target

1. Register local memory segment and get handle
2. Send serialized memory handle
3. Post push/pull operation using local/deserialized remote handles
4. Test completion of remote put/get

June 23, 2017 CS/NERSC Data Seminar
Bulk Data Transfers

Definition

*Bulk Data:* Variable length data that is (or could be) too large to send eagerly and might need special processing.
**Bulk Data Transfers**

**Definition**

*Bulk Data*: Variable length data that is (or could be) too large to send eagerly and might need special processing.

- Transfer controlled by target (better flow control)
- Memory buffer(s) abstracted by handle
- Handle must be serialized and exchanged using other means

1. Register local memory segment and get handle

1. Register local memory segment and get handle

---

June 23, 2017 CS/NERSC Data Seminar
Bulk Data Transfers

Definition

*Bulk Data*: Variable length data that is (or could be) too large to send eagerly and might need special processing.

- Transfer controlled by target (better flow control)
- Memory buffer(s) abstracted by handle
- Handle must be serialized and exchanged using other means

1. Register local memory segment and get handle
2. Send serialized memory handle
3. Post push/pull operation using local/deserialized remote handles
4. Test completion of remote put/get
Bulk Data Transfers

Definition

**Bulk Data**: Variable length data that is (or could be) too large to send eagerly and might need special processing.

- Transfer controlled by target (better flow control)
- Memory buffer(s) abstracted by handle
- Handle must be serialized and exchanged using other means

1. Register local memory segment and get handle

2. Send serialized memory handle

3. Post push/pull operation using local/deserialized remote handles
**Bulk Data Transfers**

**Definition**

*Bulk Data*: Variable length data that is (or could be) too large to send eagerly and might need special processing.

- Transfer controlled by target (better flow control)
- Memory buffer(s) abstracted by handle
- Handle must be serialized and exchanged using other means

1. Register local memory segment and get handle
2. Send serialized memory handle
3. Post push/pull operation using local/deserialized remote handles
4. Test completion of remote put/get

---

June 23, 2017 CS/NERSC Data Seminar
**Bulk Data Transfers: Example**

- **Origin snippet (contiguous):**

  ```c
  /* Initialize the interface and get target address */
  [...]

  /* Create bulk handle (only change) */
  HG_Bulk_create(hg_info->hg_bulk_class, 1, &buf, &buf_size, HG_BULK_READ_ONLY, &bulk_handle);

  /* Attach bulk handle to input parameters */
  [...] 
  in_struct.bulk_handle = bulk_handle;

  /* Create RPC request */
  HG_Create(hg_context, target_addr, rpc_id, &hg_handle);

  /* Send RPC request */
  HG_Forward(hg_handle, rpc_done_cb, &rpc_done_args, &in_struct);

  /* Make progress */
  [...]
  ```
Target snippet (RPC callback):

```c
/* Get input parameters and bulk handle */
HG_Get_input(handle, &in_struct);
[...
origin_bulk_handle = in_struct.bulk_handle;

/* Get size of data and allocate buffer */
nbytes = HG_Bulk_get_size(bulk_handle);

/* Create block handle to read data */
HG_Bulk_create(hg_info->hg_bulk_class, 1, NULL, &nbytes,
               HG_BULK_READWRITE, &local_bulk_handle);

/* Start pulling bulk data (execute call / send response in callback) */
HG_Bulk_transfer(hg_info->bulk_context, bulk_transfer_cb,
                 bulk_args, HG_BULK_PULL, hg_info->addr, origin_bulk_handle, 0,
                 local_bulk_handle, 0, nbytes, HG_OP_ID_IGNORE);
```
Non-contiguous Bulk Data Transfers

- Non contiguous memory is registered through bulk data interface...
  
  ```c
  hg_return_t HG_Bulk_create(
      hg_bulk_class_t *hg_bulk_class,
      hg_size_t count,
      void **buf_ptrs,
      const hg_size_t *buf_sizes,
      hg_uint8_t flags,
      hg_bulk_t *handle
  );
  ```

- ...and allows for scatter/gather memory transfers using virtual memory offsets and length
- Origin unaware of target memory layout
Macros

- Generate as much boilerplate code as possible for

Serialization / deserialization of parameters
- Sending / executing RPC

- Single include header file shared between origin and target
- Make use of BOOST preprocessor for macro definition
Macros

- Generate as much boilerplate code as possible for
  - Serialization / deserialization of parameters
Macros

- Generate as much boilerplate code as possible for
  - Serialization / deserialization of parameters
  - Sending / executing RPC
Generate as much boilerplate code as possible for
- Serialization / deserialization of parameters
- Sending / executing RPC

Single include header file shared between origin and target
Macros

- Generate as much boilerplate code as possible for
  - Serialization / deserialization of parameters
  - Sending / executing RPC
- Single include header file shared between origin and target
- Make use of BOOST preprocessor for macro definition
Macros

- Generate as much boilerplate code as possible for
  - Serialization / deserialization of parameters
  - Sending / executing RPC
- Single include header file shared between origin and target
- Make use of BOOST preprocessor for macro definition
  - Generate serialization / deserialization functions and structure that contains parameters
Macros: Serialization / Deserialization

MERCURY_GEN_PROC(
    struct_type_name,
    fields
)

// Define open_in_t */
typedef struct {
    hg_string_t path;
    int32_t flags;
    uint32_t mode;
} open_in_t;

// Define hg_proc_open_in_t */
static inline hg_return_thg_proc_open_in_t(hg_proc_t proc, void *data) {
    hg_return_t ret = HG_SUCCESS;
    open_in_t *struct_data = (open_in_t *)data;

    ret = hg_proc_hg_string_t(proc, &struct_data->path);
    if (ret != HG_SUCCESS) {
        HG_LOG_ERROR("Proc error");
        ret = HG_FAIL;
        return ret;
    }

    ret = hg_proc_int32_t(proc, &struct_data->flags);
    if (ret != HG_SUCCESS) {
        HG_LOG_ERROR("Proc error");
        ret = HG_FAIL;
        return ret;
    }

    ret = hg_proc_uint32_t(proc, &struct_data->mode);
    if (ret != HG_SUCCESS) {
        HG_LOG_ERROR("Proc error");
        ret = HG_FAIL;
        return ret;
    }

    return ret;
}

Generated Code

Generates proc and struct
Mercury in HDF5 Stack
Mercury in HDF5 Stack

HDF5 API

VOL

Virtual Object Layer

VOL plugins

Native (H5)

Metadata Server

Remote

Raw Mapping

VFL

Virtual File Layer

VFL drivers

mpiio
posix
sec
split

File System

Mercury

posix
sec
split
Other projects that already use Mercury

- Mochi (ANL)
- DAOS (Intel)
- DeltaFS (CMU)
- PDC (LBNL)
- MDHIM? / Legion? (LANL)
Current and Future Work

- Support cancel operations of ongoing RPC calls: done
- Shared-memory plugin and multi-progress: done
- Transparent Shared-memory selection: ongoing
- Libfabric plugin and DRC support (auth keys): ongoing
- Group membership and Publish/subscribe model: ongoing
Current and Future Work

- Support cancel operations of ongoing RPC calls **done**
- Shared-memory plugin and multi-progress **done**
Current and Future Work

- Support cancel operations of ongoing RPC calls **done**
- Shared-memory plugin and multi-progress **done**
- Transparent Shared-memory selection **ongoing**
- Libfabric plugin and DRC support (auth keys) **ongoing**
- Group membership and Publish/subscribe model **ongoing**
Where to go next

- Mercury project page
  - http://mercury-hpc.github.io/
  - https://www.mcs.anl.gov/research/projects/mochi/tutorials/
  - https://github.com/mercury-hpc
  - Download / Documentation / Source / Mailing-lists

- Current and previous contributors (non exhaustive): Phil Carns (ANL), Rob Ross (ANL), Scott Atchley (ORNL), Chuck Cranor (CMU), Xuezhao Liu (Intel), Quincey Koziol, Mohamad Chaarawi, John Jenkins, Dries Kimpe

- Work supported by DOE Office of Science Advanced Scientific Computing Research (ASCR) research and by NSF Directorate for Computer & Information Science & Engineering (CISE) Division of Computing and Communication Foundations (CCF) core program funding