Mercury: RPC for High-Performance Computing

Jerome Soumagne, Dries Kimpe
The HDF Group, Argonne National Laboratory

July 2, 2013

RPC and Exascale Fast-forward

RPC and Exascale Fast-forward

- Typical HPC workflow:
  1. Compute and produce data
  2. Store data
  3. Analyze data
  4. Visualize data

RPC and Exascale Fast-forward

- Typical HPC workflow:
  1. Compute and produce data
  2. Store data
  3. Analyze data
  4. Visualize data
- Exascale HPC workflow (*in-transit*):
  1. Compute and produce data
  2. Store data (data staging)
  3. Analyze data in-transit
  4. Visualize data in-transit

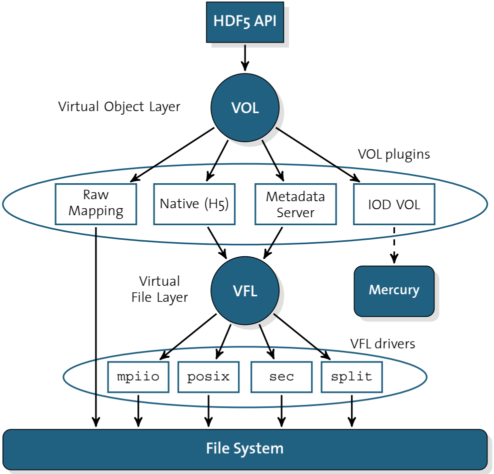
RPC and Exascale Fast-forward

- Typical HPC workflow:
  1. Compute and produce data
  2. Store data
  3. Analyze data
  4. Visualize data
- Exascale HPC workflow (*in-transit*):
  1. Compute and produce data
  2. Store data (data staging)
  3. Analyze data in-transit
  4. Visualize data in-transit
- Distributed workflow with nodes / systems dedicated to specific task
  - Compute nodes with minimal environment
  - I/O / analysis / visualization libraries only available on remote resources

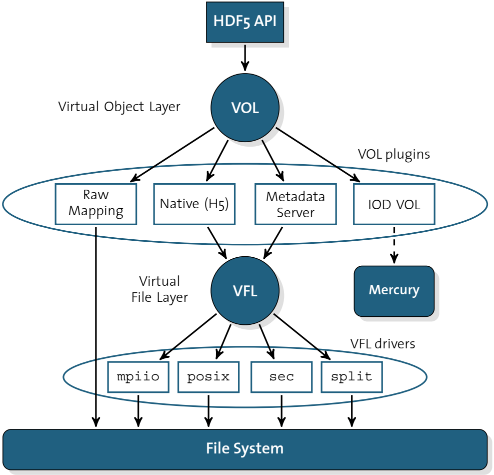
RPC and Exascale Fast-forward

- Typical HPC workflow:
  1. Compute and produce data
  2. Store data
  3. Analyze data
  4. Visualize data
- Exascale HPC workflow (*in-transit*):
  1. Compute and produce data
  2. Store data (data staging)
  3. Analyze data in-transit
  4. Visualize data in-transit
- Distributed workflow with nodes / systems dedicated to specific task
  - Compute nodes with minimal environment
  - I/O / analysis / visualization libraries only available on remote resources
- Transparent solution: Remote Procedure Call (RPC)
  - E.g., store data using HDF5 but re-route I/O calls to I/O nodes

Mercury in HDF5 Stack

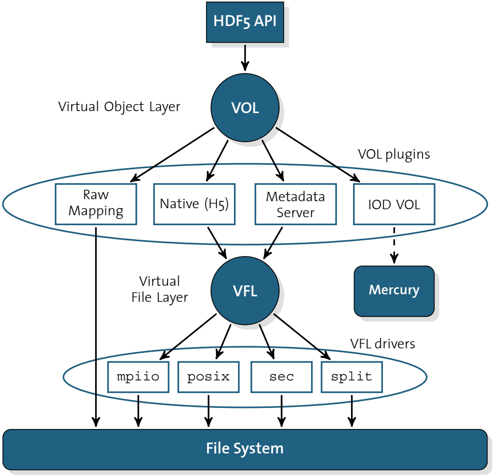


Mercury in HDF5 Stack



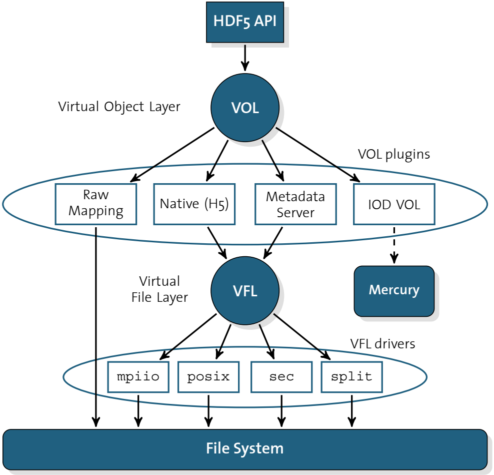
- Mercury must support

Mercury in HDF5 Stack



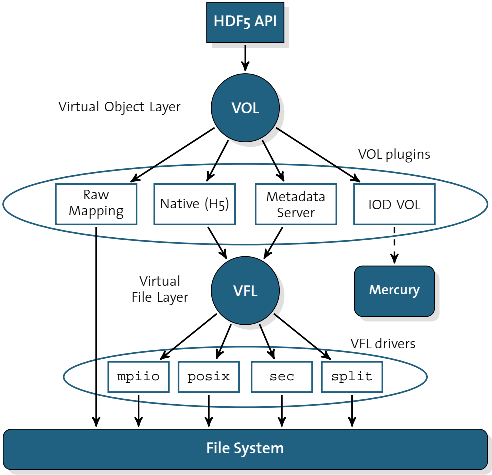
- Mercury must support
  - Non-blocking transfers

Mercury in HDF5 Stack



- Mercury must support
  - Non-blocking transfers
  - Large data arguments

Mercury in HDF5 Stack



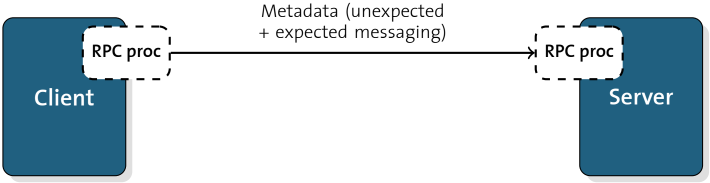
- Mercury must support
  - Non-blocking transfers
  - Large data arguments
  - Native transport protocols

Overview

RPC proc
Client
RPC proc
Server

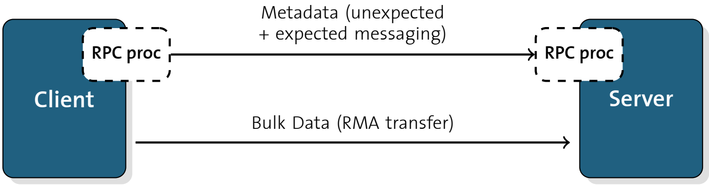
Overview

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



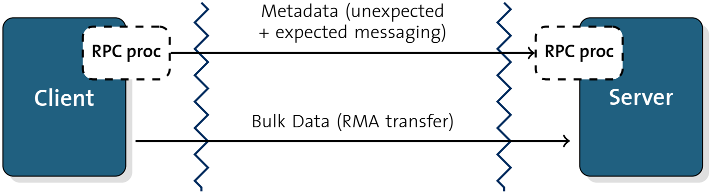
Overview

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



Overview

- Function arguments / metadata transferred with RPC request
  - Two-sided model with unexpected / expected messaging
  - Message size limited to a few kilobytes
- Bulk data transferred using separate and dedicated API
  - One-sided model that exposes RMA semantics
- Network Abstraction Layer
  - Allows definition of multiple network plugins
  - Two functional plugins MPI (MPI2) and BMI but implement one-sided over two-sided
  - More plugins to come



Network Abstraction Layer

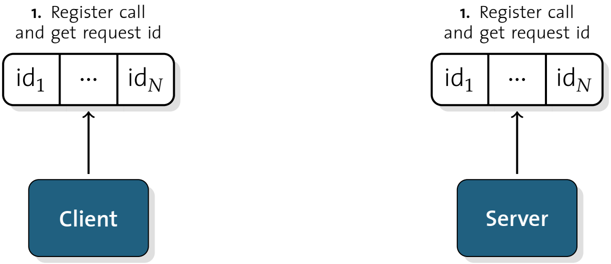
Remote Procedure Call

- Mechanism used to send an RPC request

id_1
…
id_N
id_1
…
id_N
Client
Server

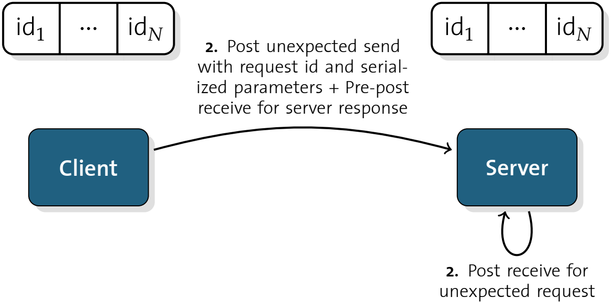
Remote Procedure Call

- Mechanism used to send an RPC request



Remote Procedure Call

- Mechanism used to send an RPC request



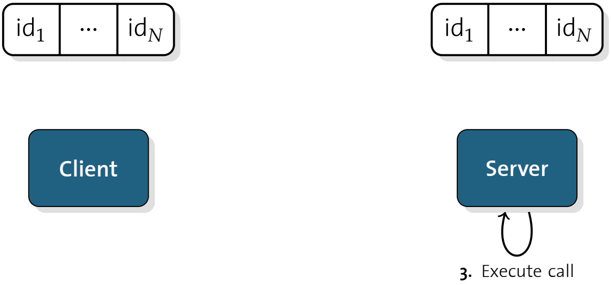
id_1 … id_N
id_1 … id_N
Client
Server

2. Post unexpected send with request id and serialized parameters + Pre-post receive for server response

2. Post receive for unexpected request

Remote Procedure Call

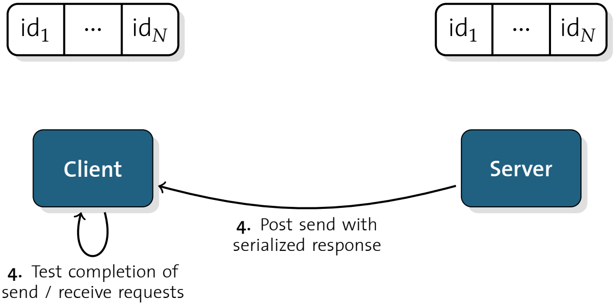
- Mechanism used to send an RPC request



id_1 | ⋯ | id_N
id_1 | ⋯ | id_N
Client
Server
3. Execute call

Remote Procedure Call

- Mechanism used to send an RPC request



id_1 | ⋯ | id_N
id_1 | ⋯ | id_N
Client
Server
4. Post send with serialized response
4. Test completion of send / receive requests

Remote Procedure Call: Example

- Client snippet:

```
open_in_t in_struct;
open_out_t out_struct;

/* Initialize the interface */
[...]
NA_Addr_lookup(network_class, server_name, &server_addr);

/* Register RPC call */
rpc_id = HG_REGISTER(”open”, open_in_t, open_out_t);

/* Fill input parameters */
[...]
in_struct.in_param0 = in_param0;

/* Send RPC request */
HG_Forward(server_addr, rpc_id, &in_struct, &out_struct,
    &rpc_request);

/* Wait for completion */
HG_Wait(rpc_request, HG_MAX_IDLE_TIME, HG_STATUS_IGNORE);

/* Get output parameters */
[...]
out_param0 = out_struct.out_param0;
```

Remote Procedure Call: Example

- Server snippet (main loop):

```
int main(int argc, void *argv[])
{
  /* Initialize the interface */
  [...]

  /* Register RPC call */
  HG_HANDLER_REGISTER(”open”, open_rpc, open_in_t, open_out_t);

  /* Process RPC calls */
  while (!finalized) {
    HG_Handler_process(timeout, HG_STATUS_IGNORE);
  }

  /* Finalize the interface */
  [...]
}
```

Remote Procedure Call: Example

- Server snippet (RPC callback):

```
int open_rpc(hg_handle_t handle)
{
  open_in_t in_struct;
  open_out_t out_struct;

  /* Get input parameters and bulk handle */
  HG_Handler_get_input(handle, &in_struct);
  [...]
  in_param0 = in_struct.in_param0;

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

  /* Fill output structure */
  open_out_struct.out_param0 = out_param0;

  /* Send response back */
  HG_Handler_start_output(handle, &out_struct);

  return HG_SUCCESS;
}
```

Bulk Data Transfers

- Mechanism used to transfer bulk data
  - Transfer controlled by server
  - Memory buffer abstracted by memory handle
  - Client memory handle must be serialized and sent to the server

Client

Server

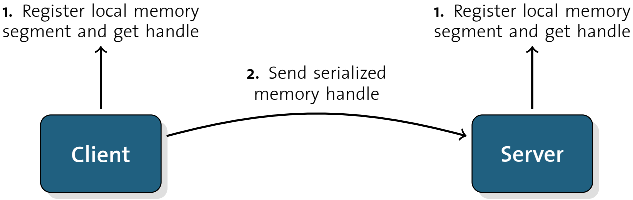
Bulk Data Transfers

- Mechanism used to transfer bulk data
  - Transfer controlled by server
  - Memory buffer abstracted by memory handle
  - Client memory handle must be serialized and sent to the server

1. Register local memory segment and get handle
Client
1. Register local memory segment and get handle
Server

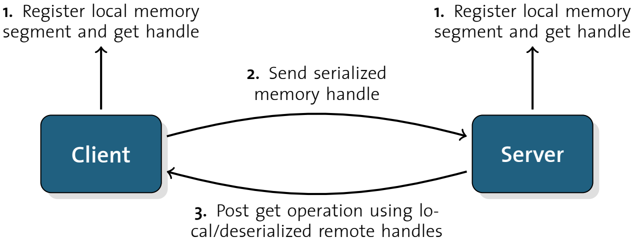
Bulk Data Transfers

- Mechanism used to transfer bulk data
  - Transfer controlled by server
  - Memory buffer abstracted by memory handle
  - Client memory handle must be serialized and sent to the server


1. Register local memory segment and get handle
2. Send serialized memory handle
1. Register local memory segment and get handle
Client
Server

Bulk Data Transfers

- Mechanism used to transfer bulk data
  - Transfer controlled by server
  - Memory buffer abstracted by memory handle
  - Client memory handle must be serialized and sent to the server



1. Register local memory segment and get handle

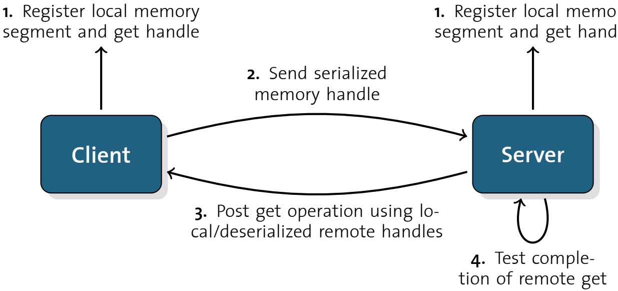
1. Register local memory segment and get handle

2. Send serialized memory handle

3. Post get operation using local/deserialized remote handles

Bulk Data Transfers

- Mechanism used to transfer bulk data
  - Transfer controlled by server
  - Memory buffer abstracted by memory handle
  - Client memory handle must be serialized and sent to the server



1. Register local memory segment and get handle

1. Register local memory segment and get handle

2. Send serialized memory handle

Client

Server

3. Post get operation using local/deserialized remote handles

4. Test completion of remote get

Bulk Data Transfers: Example

- Client snippet (contiguous):

```
/* Initialize the interface */
[...]
/* Register RPC call */
rpc_id = HG_REGISTER(”write”, write_in_t, write_out_t);

/* Create bulk handle */
HG_Bulk_handle_create(buf, buf_size,
    HG_BULK_READ_ONLY, &bulk_handle);

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

/* Send RPC request */
HG_Forward(server_addr, rpc_id, &in_struct, &out_struct,
    &rpc_request);

/* Wait for completion */
HG_Wait(rpc_request, HG_MAX_IDLE_TIME, HG_STATUS_IGNORE);
```

Bulk Data Transfers: Example

- Server snippet (RPC callback):

```
/* Get input parameters and bulk handle */
HG_Handler_get_input(handle, &in_struct);
[...]
bulk_handle = in_struct.bulk_handle;

/* Get size of data and allocate buffer */
nbytes = HG_Bulk_handle_get_size(bulk_handle);
buf = malloc(nbytes);

/* Create block handle to read data */
HG_Bulk_block_handle_create(buf, nbytes,
    HG_BULK_READWRITE, &bulk_block_handle);

/* Start reading bulk data  */
HG_Bulk_read_all(client_addr, bulk_handle,
    bulk_block_handle, &bulk_request);

/* Wait for completion */
HG_Bulk_wait(bulk_request,
    HG_MAX_IDLE_TIME, HG_STATUS_IGNORE);
```

Non-contiguous Bulk Data Transfers

- Non contiguous memory is registered through bulk data interface…

```
int HG_Bulk_handle_create_segments(
        hg_bulk_segment_t *bulk_segments,
        size_t segment_count,
        unsigned long flags,
        hg_bulk_t *handle);
```

- …which maps to network abstraction layer if plugin supports it…

```
int NA_Mem_register_segments(na_class_t *network_class,
        na_segment_t *segments,
        na_size_t segment_count,
        unsigned long flags,
        na_mem_handle_t *mem_handle);
```

- …otherwise several `na_mem_handle_t` created and `hg_bulk_t` may therefore have a variable size
    - If serialized `hg_bulk_t` too large, use bulk data API to register memory and pull memory descriptors from server

Non-contiguous Bulk Data Transfers: API

- Non-blocking read

```
int HG_Bulk_read(na_addr_t addr,
        hg_bulk_t bulk_handle,
        size_t bulk_offset,
        hg_bulk_block_t block_handle,
        size_t block_offset,
        size_t block_size,
        hg_bulk_request_t *bulk_request);
```

- Non-blocking write

```
int HG_Bulk_write(na_addr_t addr,
        hg_bulk_t bulk_handle,
        size_t bulk_offset,
        hg_bulk_block_t block_handle,
        size_t block_offset,
        size_t block_size,
        hg_bulk_request_t *bulk_request);
```

Non-contiguous Bulk Data Transfers: Example

- Client snippet:

```
/* Initialize the interface */
[...]
/* Register RPC call */
rpc_id = HG_REGISTER(”write”, write_in_t, write_out_t);

/* Provide data layout information */
for (i = 0; i < BULK_NX ; i++) {
  segments[i].address = buf[i];
  segments[i].size = BULK_NY * sizeof(int);
}

/* Create bulk handle with segment info */
HG_Bulk_handle_create_segments(segments, BULK_NX,
    HG_BULK_READ_ONLY, &bulk_handle);

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

/* Send RPC request */
HG_Forward(server_addr, rpc_id, &in_struct, &out_struct,
    &rpc_request);
```

Non-contiguous Bulk Data Transfers: Example

- Server snippet:

```
/* Get input parameters and bulk handle */
HG_Handler_get_input(handle, &in_struct);
[...]
bulk_handle = in_struct.bulk_handle;

/* Get size of data and allocate buffer */
nbytes = HG_Bulk_handle_get_size(bulk_handle);
buf = malloc(nbytes);

/* Create block handle to read data */
HG_Bulk_block_handle_create(buf, nbytes,
    HG_BULK_READWRITE, &bulk_block_handle);

/* Start reading bulk data  */
HG_Bulk_read_all(client_addr, bulk_handle,
    bulk_block_handle, &bulk_request);

/* Wait for completion */
HG_Bulk_wait(bulk_request,
    HG_MAX_IDLE_TIME, HG_STATUS_IGNORE);
```

Fine-grained Transfers

Fine-grained Transfers

- Two issues with previous example

Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited

Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read

Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution

Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call

Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call

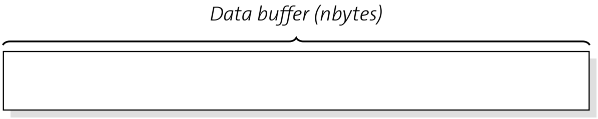
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call

Data buffer (nbytes)

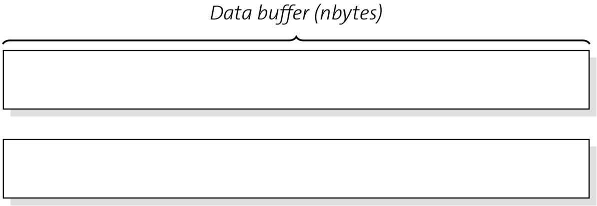
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



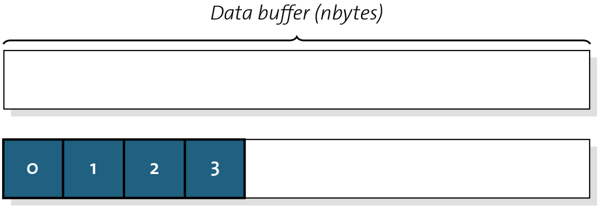
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



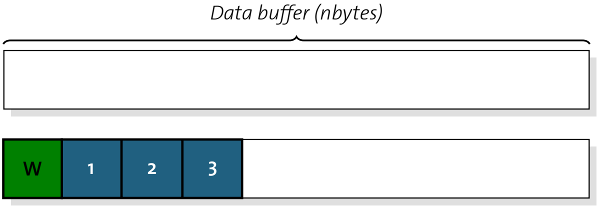
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



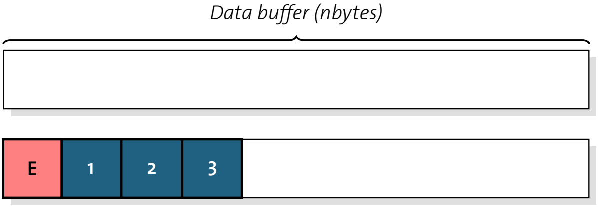
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



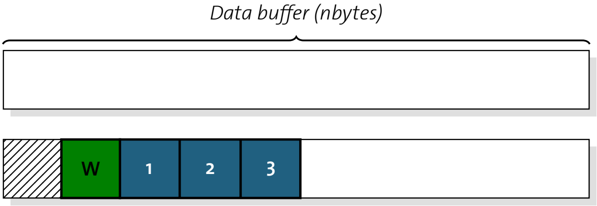
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



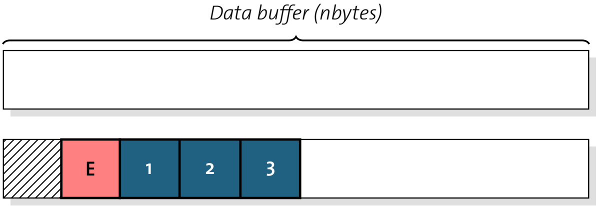
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



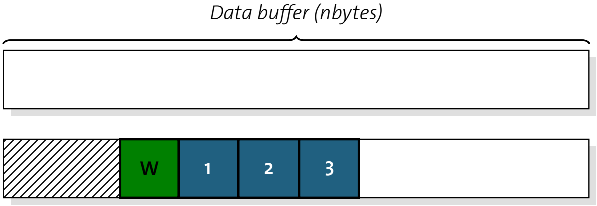
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



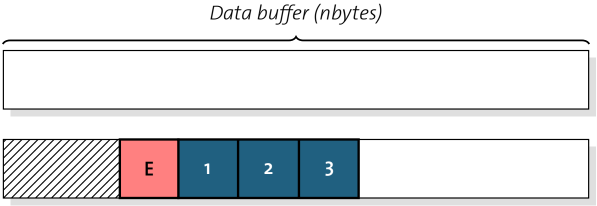
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



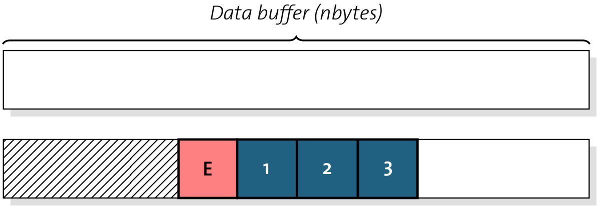
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



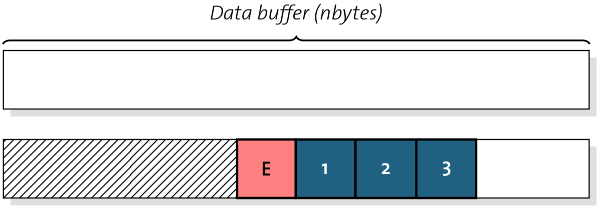
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



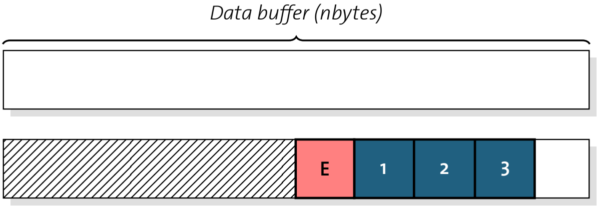
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



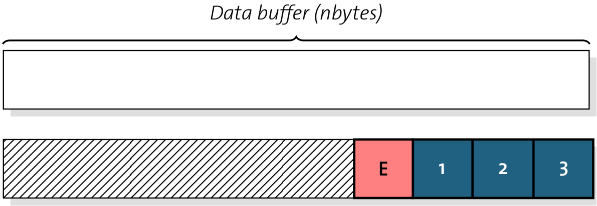
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



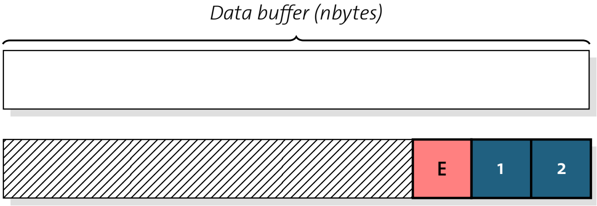
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



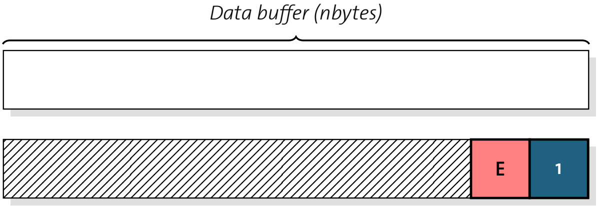
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



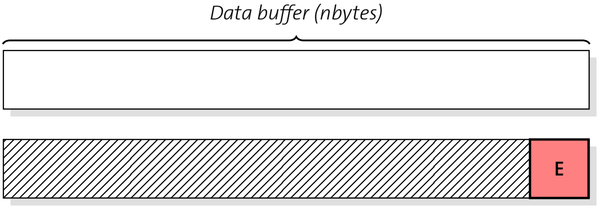
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



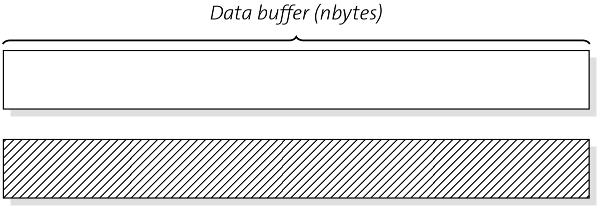
Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



Fine-grained Transfers

- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing / executing the RPC call



Fine-grained Transfers: Example

- Server snippet (part 1):

```
/* Get input parameters and bulk handle */
HG_Handler_get_input(handle, &in_struct);
[...]
bulk_handle = in_struct.bulk_handle;

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

/* Initialize pipeline and start reads */
for (p = 0; p < PIPELINE_SIZE; p++) {
  size_t offset = p * PIPELINE_BUFFER_SIZE;
  buf[p] = malloc(PIPELINE_BUFFER_SIZE);

  /* Create block handle to read data */
  HG_Bulk_block_handle_create(buf[p],
    PIPELINE_BUFFER_SIZE, HG_BULK_READWRITE,
    &bulk_block_handle[p]);

  /* Start read of data chunk */
  HG_Bulk_read(client_addr, bulk_handle,
    offset, bulk_block_handle[p], 0,
    PIPELINE_BUFFER_SIZE, &bulk_request[p]);
}
```

Fine-grained Transfers: Example

- Server snippet (part 2):

```
while (nbytes_read != nbytes) {
  for (p = 0; p < PIPELINE_SIZE; p++) {
    size_t offset = start_offset + p * PIPELINE_BUFFER_SIZE;
    /* Wait for data chunk */
    HG_Bulk_wait(bulk_request[p],
      HG_MAX_IDLE_TIME, HG_STATUS_IGNORE);
    nbytes_read += PIPELINE_BUFFER_SIZE;

    /* Do work (write data chunk) */
    write(buf[p], offset, PIPELINE_BUFFER_SIZE);

    /* Start another read */
    offset += PIPELINE_BUFFER_SIZE * PIPELINE_SIZE;
    if (offset < nbytes) {
      HG_Bulk_read(client_addr, bulk_handle, offset,
        bulk_block_handle[p], 0, PIPELINE_BUFFER_SIZE,
        &bulk_request[p]);
    } else {
      /* Start read with remaining piece */
    }
  }
  start_offset += PIPELINE_BUFFER_SIZE * PIPELINE_SIZE;
}
```

Macros

Macros

- Generate as much boilerplate code as possible for

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

Macros

- Generate as much boilerplate code as possible for
  - Serialization / deserialization of parameters
  - Sending / executing RPC
- Single include header file shared between client and server

Macros

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

Macros

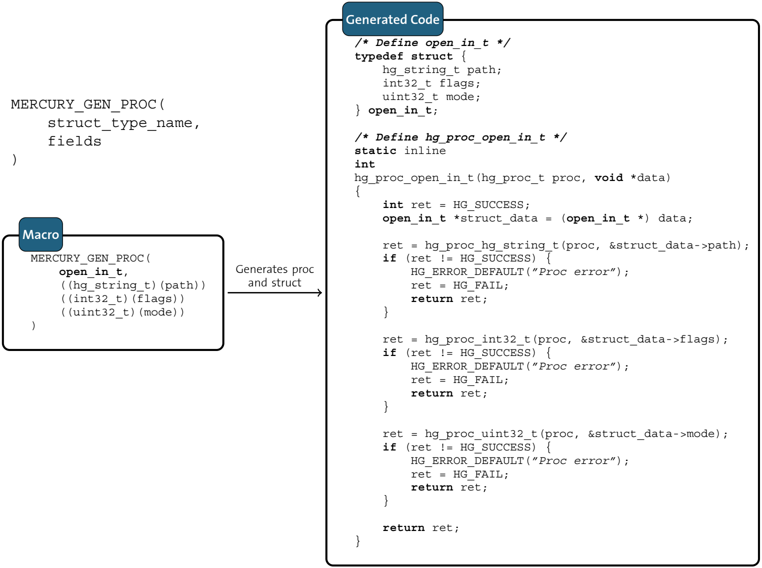
- Generate as much boilerplate code as possible for
  - Serialization / deserialization of parameters
  - Sending / executing RPC
- Single include header file shared between client and server
- Make use of BOOST preprocessor for macro definition
  - Generate serialization / deserialization functions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

Macros

- Generate as much boilerplate code as possible for
  - Serialization / deserialization of parameters
  - Sending / executing RPC
- Single include header file shared between client and server
- Make use of BOOST preprocessor for macro definition
  - Generate serialization / deserialization functions and structure that contains parameters
  - Generate synchronous RPC stub

Macros: Serialization / Deserialization

```
MERCURY_GEN_PROC(
    struct_type_name,
    fields
)
```



Macro

```
MERCURY_GEN_PROC(
    open_in_t,
    ((hg_string_t)(path))
    ((int32_t)(flags))
    ((uint32_t)(mode))
)
```

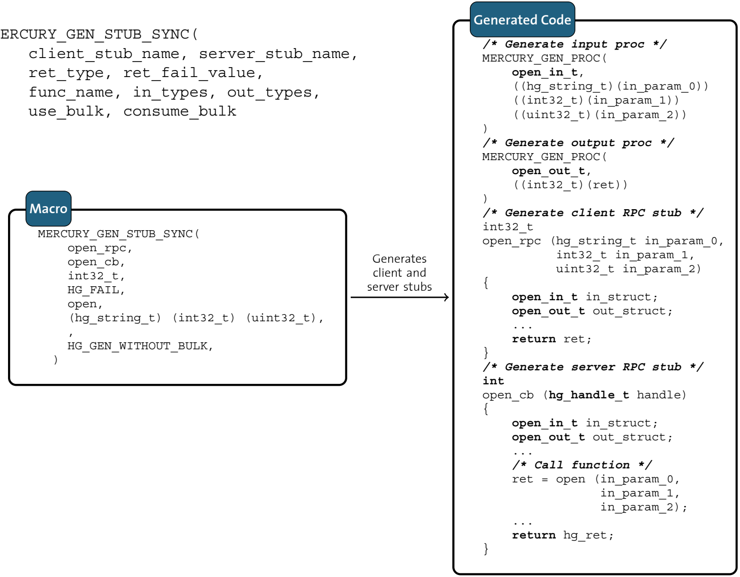
Generates proc and struct →

Generated Code

```
/* 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
int
hg_proc_open_in_t(hg_proc_t proc, void *data)
{
    int 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_ERROR_DEFAULT("Proc error");
        ret = HG_FAIL;
        return ret;
    }

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

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

    return ret;
}
```

Macros: RPC Stubs

```
MERCURY_GEN_STUB_SYNC(
    client_stub_name, server_stub_name,
    ret_type, ret_fail_value,
    func_name, in_types, out_types,
    use_bulk, consume_bulk
)
```



Macro

```
MERCURY_GEN_STUB_SYNC(
    open_rpc,
    open_cb,
    int32_t,
    HG_FAIL,
    open,
    (hg_string_t) (int32_t) (uint32_t),
    ,
    HG_GEN_WITHOUT_BULK,
  )
```

Generates
client and
server stubs

Generated Code

```
/* Generate input proc */
MERCURY_GEN_PROC(
    open_in_t,
    ((hg_string_t)(in_param_0))
    ((int32_t)(in_param_1))
    ((uint32_t)(in_param_2))
)
/* Generate output proc */
MERCURY_GEN_PROC(
    open_out_t,
    ((int32_t)(ret))
)
/* Generate client RPC stub */
int32_t
open_rpc (hg_string_t in_param_0,
          int32_t in_param_1,
          uint32_t in_param_2)
{
    open_in_t in_struct;
    open_out_t out_struct;
    ...
    return ret;
}
/* Generate server RPC stub */
int
open_cb (hg_handle_t handle)
{
    open_in_t in_struct;
    open_out_t out_struct;
    ...
    /* Call function */
    ret = open (in_param_0,
                in_param_1,
                in_param_2);
    ...
    return hg_ret;
}
```

Future Work

Future Work

- Apply macros to POSIX API
  - Apply to HDF5 VFD
  - Reroute calls to RPC calls using dynamic linking

Future Work

- Apply macros to POSIX API
  - Apply to HDF5 VFD
  - Reroute calls to RPC calls using dynamic linking
- Implement plugin that makes use of true RMA capability

Future Work

- Apply macros to POSIX API
    - Apply to HDF5 VFD
    - Reroute calls to RPC calls using dynamic linking
- Implement plugin that makes use of true RMA capability
- Support cancel operations of ongoing RPC calls

Early Performance Evaluation

Early Performance Evaluation

- RPC request execution on Cray XE6

Early Performance Evaluation

- RPC request execution on Cray XE6
  - With XDR encoding: 23 μs

Early Performance Evaluation

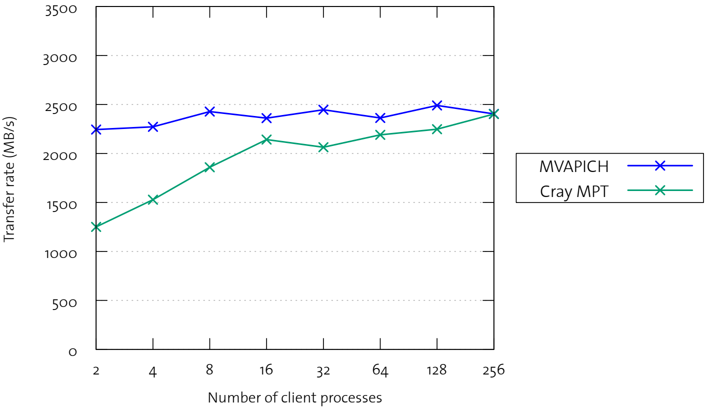
- RPC request execution on Cray XE6
  - With XDR encoding: 23 μs
  - Without XDR encoding: 20 μs

Early Performance Evaluation

- RPC request execution on Cray XE6
  - With XDR encoding: 23 µs
  - Without XDR encoding: 20 µs
- Scalability / aggregate bandwidth of RPC requests to single server with bulk data transfer (on Infiniband cluster and on Cray XE6)

Early Performance Evaluation

- RPC request execution on Cray XE6
  - With XDR encoding: 23 µs
  - Without XDR encoding: 20 µs
- Scalability / aggregate bandwidth of RPC requests to single server with bulk data transfer (on Infiniband cluster and on Cray XE6)



Acknowledgments

- Work supported by:
  - The Exascale FastForward project, LLNS subcontract no. B599860
  - The Office of Advanced Scientific Computer Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357
- Contributors (alphabetical order):
  - Ahmad Afsahiz
  - Mohamad Chaarawi
  - Quincey Koziol
  - Robert Ross
  - Judicael Zounmevo

Questions

- Mercury project page:
  - `http://trac.mcs.anl.gov/projects/mercury`