Mercury: Enabling Remote Procedure Call for High-Performance Computing

J. Soumagne, **D. Kimpe**, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross

The HDF Group, **Argonne National Laboratory**, Queen's University

November 26, 2013

RPC and High-Performance Computing

Remote Procedure Call (RPC)

- Allow local calls to be transparently executed on remote resources
- Already widely used to support distributed services
  - Google Protocol Buffers, Facebook Thrift, CORBA, Java RMI, etc.

RPC and High-Performance Computing

Remote Procedure Call (RPC)

- Allow local calls to be transparently executed on remote resources
- Already widely used to support distributed services
  - Google Protocol Buffers, Facebook Thrift, CORBA, Java RMI, 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

RPC and High-Performance Computing

Remote Procedure Call (RPC)

- Allow local calls to be transparently executed on remote resources
- Already widely used to support distributed services
  - Google Protocol Buffers, Facebook Thrift, CORBA, Java RMI, 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

RPC and High-Performance Computing

Remote Procedure Call (RPC)

- Allow local calls to be transparently executed on remote resources
- Already widely used to support distributed services
  - Google Protocol Buffers, Facebook Thrift, CORBA, Java RMI, 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-situ instead of sequences of SPMD

Mercury

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

Mercury

Objective

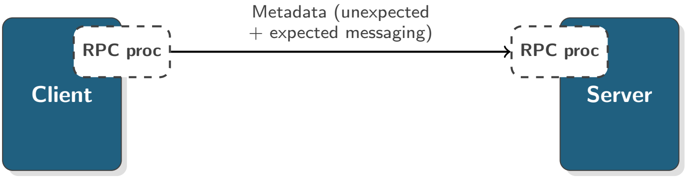
Create a reusable RPC library for use in HPC that can serve as a basis for services such as storage systems, I/O forwarding, analysis frameworks and other forms of inter-application communication

- 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 machines
- Similar approaches with some differences
  - *I/O Forwarding Scalability Layer* (IOFSL)
  - *NEtwork Scalable Service Interface* (Nessie)
  - Lustre RPC

Overview

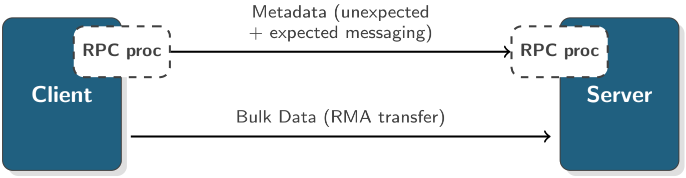
Overview

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



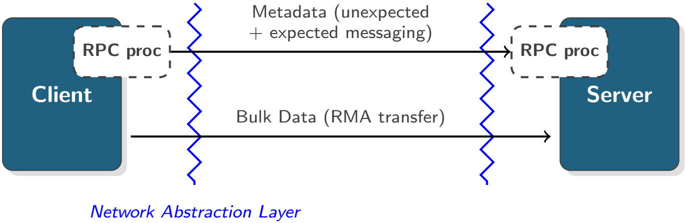
Overview

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



Remote Procedure Call

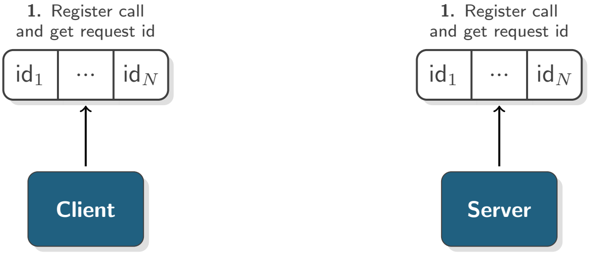
Internal Details: Please forget soon!

- Mechanism used to send an RPC request

Remote Procedure Call

Internal Details: Please forget soon!

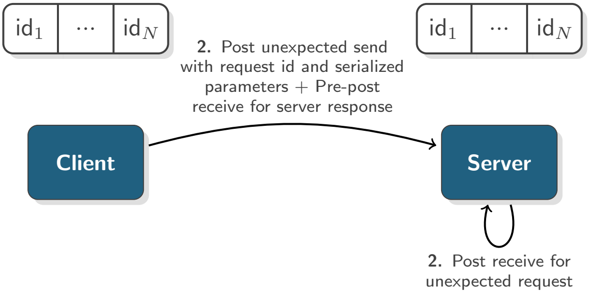
- Mechanism used to send an RPC request



Remote Procedure Call

Internal Details: Please forget soon!

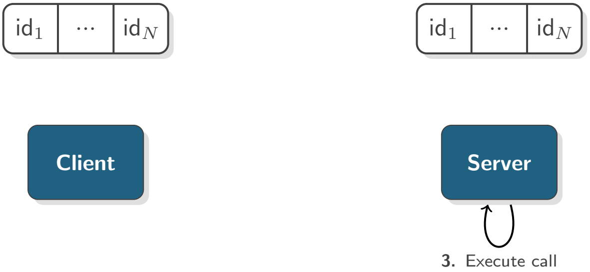
- Mechanism used to send an RPC request



Remote Procedure Call

Internal Details: Please forget soon!

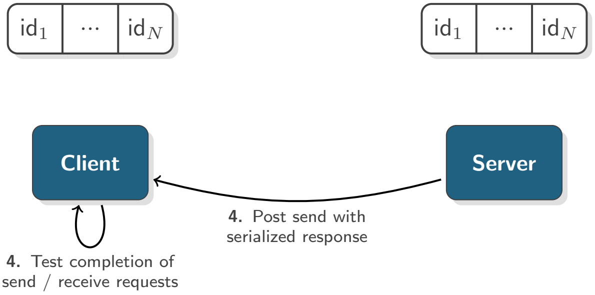
- Mechanism used to send an RPC request



Remote Procedure Call

Internal Details: Please forget soon!

- Mechanism used to send an RPC request



Remote Procedure Call: Example Code

- 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 Code

- 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 Code

- 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

Definition

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

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

Bulk Data Transfers

Definition

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

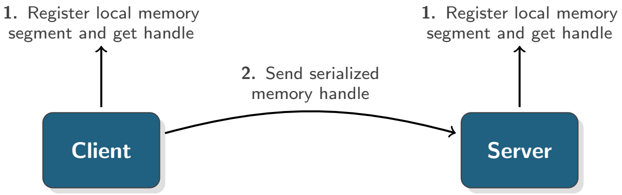
- Transfer controlled by server (better flow control)
- Memory buffer(s) abstracted by handle
- handles must be serialized and exchanged using other means

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 server (better flow control)
- Memory buffer(s) abstracted by handle
- handles must be serialized and exchanged using other means

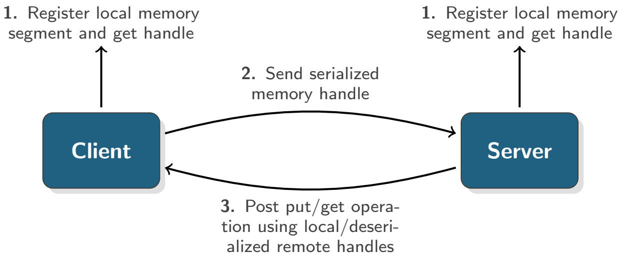


Bulk Data Transfers

Definition

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

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

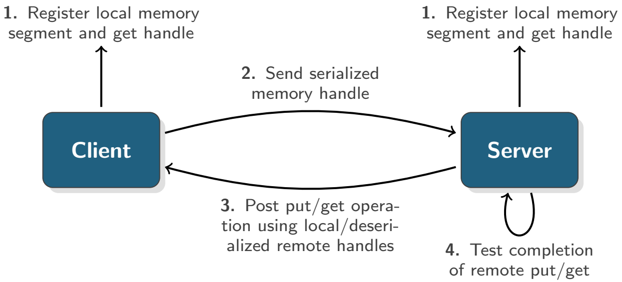


Bulk Data Transfers

Definition

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

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



Bulk Data Transfers: Example

- Client snippet (contiguous):
  Note: no client changes

```
/* 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
  - In both cases, origin unaware of target memory layout

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

- 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

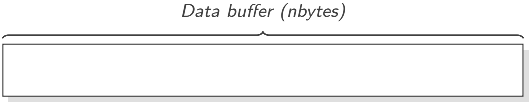
- Two issues with previous example
  1. Server memory size is limited
  2. Server waits for all the data to arrive before writing
     - Makes us pay the latency of an entire RMA read
- Solution
  - Pipeline transfers and overlap communication / execution
    - Transfers can complete while writing /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

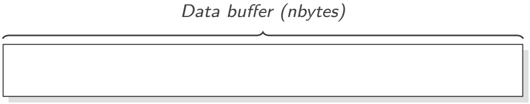
Fine-grained Transfers

- 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



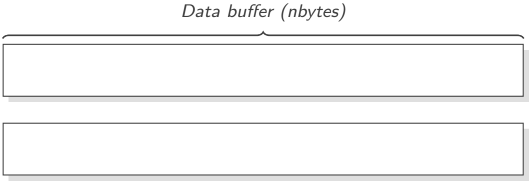
Fine-grained Transfers

- 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



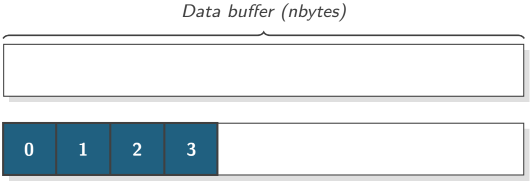
Fine-grained Transfers

- 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



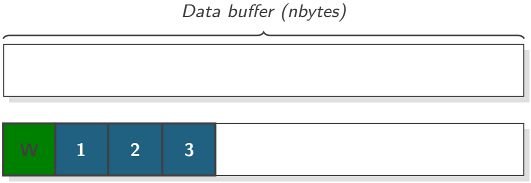
Fine-grained Transfers

- 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



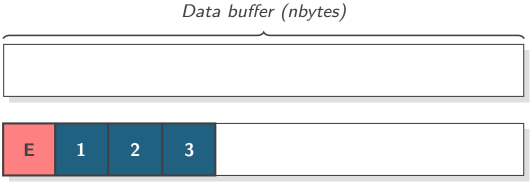
Fine-grained Transfers

- 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



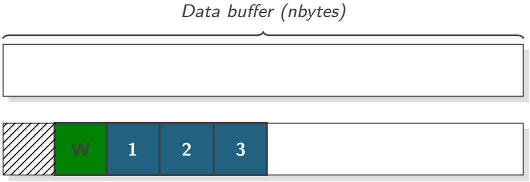
Fine-grained Transfers

- 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



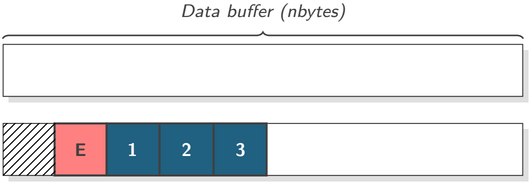
Fine-grained Transfers

- 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



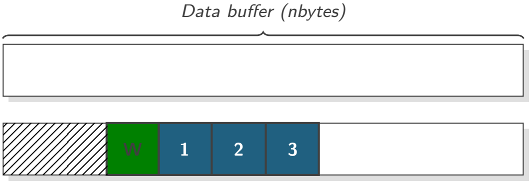
Fine-grained Transfers

- 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



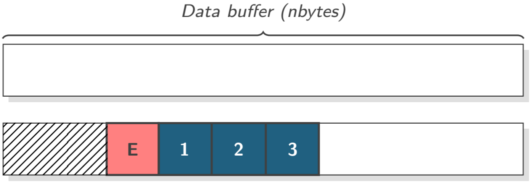
Fine-grained Transfers

- 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



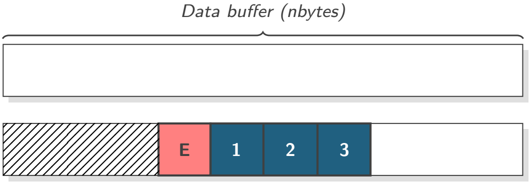
Fine-grained Transfers

- 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



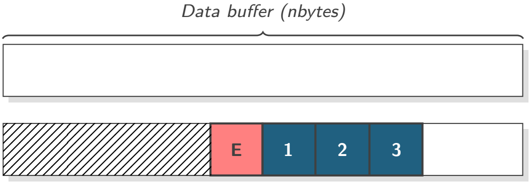
Fine-grained Transfers

- 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



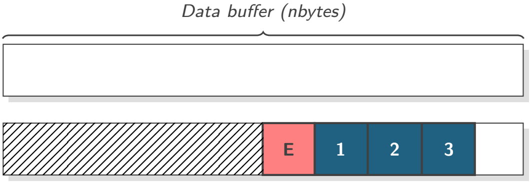
Fine-grained Transfers

- 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



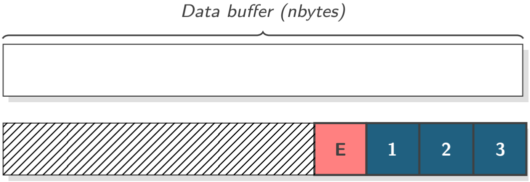
Fine-grained Transfers

- 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



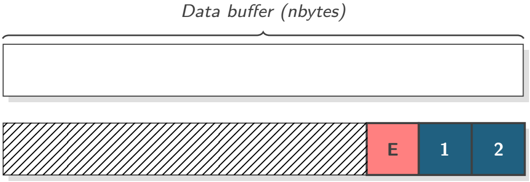
Fine-grained Transfers

- 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



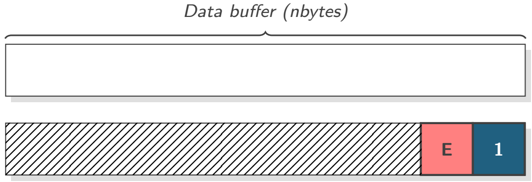
Fine-grained Transfers

- 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



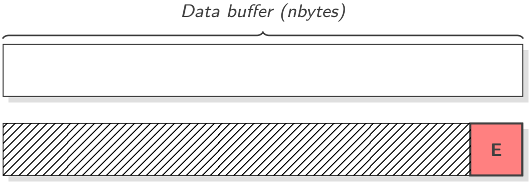
Fine-grained Transfers

- 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



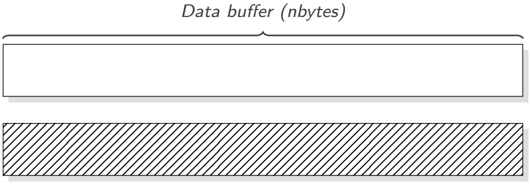
Fine-grained Transfers

- 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



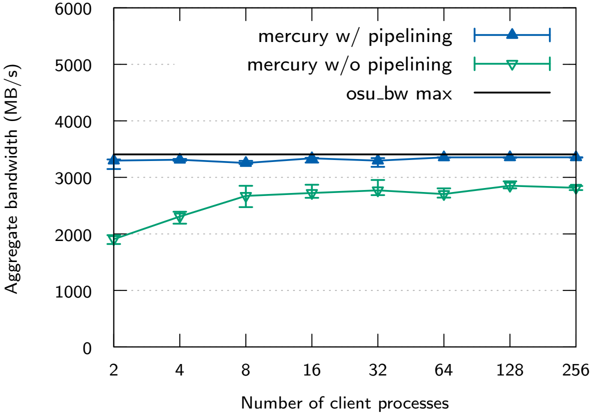
Fine-grained Transfers

- 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



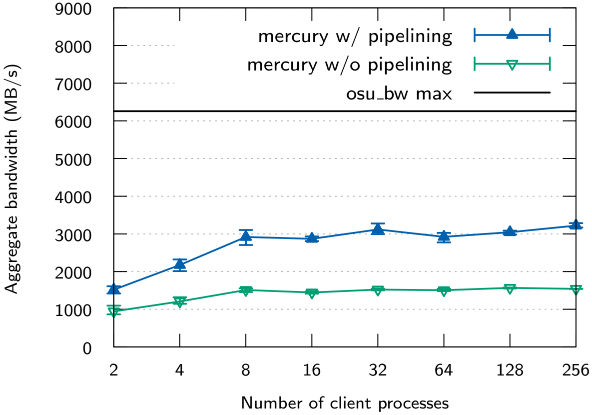
Performance Evaluation

- Scalability / aggregate bandwidth of RPC requests to single server with bulk data transfer (QDR 4X Infiniband cluster)



Performance Evaluation

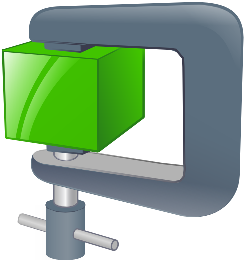
- Scalability / aggregate bandwidth of RPC requests to single server with bulk data transfer (Cray XE6)



Performance Evaluation

- NULL RPC request execution on Cray XE6
  - With XDR encoding: 23 μs
  - Without XDR encoding: 20 μs
- About 50 000 calls /s
- Still working on improving that result
- Can depend on server side CPU affinity etc

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: 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;
}
```

Current and Future Work

- Add true RMA capability NA plugins
  (ibverbs, DMAPP, SSM, NNTI)
- Checksum parameters for data integrity (done)
- Support cancel operations of ongoing RPC calls (ongoing)
- Change progress model to callback and trigger (done)
  (both Mercury and NA)
- Optimizations: batches and eager bulk data
- Integrate Mercury into other projects
  - Mercury POSIX: Forward POSIX calls using dynamic linking
  - Triton (done)
  - IOFSL
  - HDF5 virtual object plugins

Where to go next

Mercury project page

- http://www.mcs.anl.gov/projects/mercury
- Download / Documentation / Source / Mailing-lists

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
- This research was supported by the United States Department of Defense