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

September 24, 2013
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 workflow
1. Compute and produce data
2. Store data
3. Analyze data
4. Visualize data

Distributed HPC workflow

Nodes/systems dedicated to specific task

More important at Exascale for processing data

Compute nodes with minimal environment

I/O, analysis, visualization libraries only available on remote resources
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 workflow**
  1. Compute and produce data
  2. Store data
  3. Analyze data
  4. Visualize 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 workflow
  1. Compute and produce data
  2. Store data
  3. Analyze data
  4. Visualize data

- Distributed HPC workflow
  - Nodes/systems dedicated to specific task
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 workflow
1. Compute and produce data
2. Store data
3. Analyze data
4. Visualize data

Distributed HPC workflow
- Nodes/systems dedicated to specific task

More important at Exascale for processing data
- Compute nodes with minimal environment
- I/O, analysis, visualization libraries only available on remote resources
Objective: create a layer that can serve as a basis for storage systems, I/O forwarders or analysis frameworks.
- **Objective**: create a layer that can serve as a basis for storage systems, I/O forwarders or analysis frameworks
- Cannot re-use common RPC frameworks *as-is*
  - Do not support large data transfers
  - Mostly built on top of TCP/IP protocols
- **Objective**: create a layer that can serve as a basis for storage systems, I/O forwarders or analysis frameworks
- Cannot re-use common RPC frameworks *as-is*
  - Do not support large data transfers
  - Mostly built on top of TCP/IP protocols
- Use in HPC systems means that it must support
  - Non-blocking transfers
  - Large data arguments
  - Native transport protocols
Objective: create a layer that can serve as a basis for storage systems, I/O forwarders or analysis frameworks

Cannot re-use common RPC frameworks as-is
- Do not support large data transfers
- Mostly built on top of TCP/IP protocols

Use in HPC systems means that it must support
- Non-blocking transfers
- Large data arguments
- Native transport protocols

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

---

Diagram:

- **Client**
- **Server**
- RPC proc

---

*September 24, 2013 Mercury: Enabling Remote Procedure Call for High-Performance Computing*
- Function arguments / metadata transferred with RPC request
  - Two-sided model with unexpected / expected messaging
  - Message size limited to a few kilobytes
- 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

1. Register call and get request id
2. Post unexpected send with request id and serialized parameters + Pre-post receive for server response
3. Post receive for unexpected request
4. Execute call
4. Test completion of send/receive requests

\[
\begin{array}{cccc}
\text{id}_1 & \cdots & \text{id}_N \\
\end{array}
\]

Client

\[
\begin{array}{cccc}
\text{id}_1 & \cdots & \text{id}_N \\
\end{array}
\]

Server
Remote Procedure Call

- Mechanism used to send an RPC request

1. Register call and get request id

Client

\[ \text{id}_1 \quad \ldots \quad \text{id}_N \]

Server

\[ \text{id}_1 \quad \ldots \quad \text{id}_N \]
Remote Procedure Call

- Mechanism used to send an RPC request

1. Register call and get request id
2. Post unexpected send with request id and serialized parameters + Pre-post receive for server response
3. Execute call
4. Test completion of send / receive requests

Client

server

id_1  ...  id_N

2. Post receive for unexpected request
Remote Procedure Call

- Mechanism used to send an RPC request

1. Register call and get request id
2. Post unexpected send with request id and serialized parameters + Pre-post receive for server response
3. Execute call
4. Test completion of send / receive requests

Client

Server

id_1  ...  id_N

id_1  ...  id_N

September 24, 2013 Mercury: Enabling Remote Procedure Call for High-Performance Computing
Remote Procedure Call

- Mechanism used to send an RPC request

1. Register call and get request id
2. Post unexpected send with request id and serialized parameters + Pre-post receive for server response
3. Post receive for unexpected request
4. Execute call
4. Test completion of send / receive requests

\[ id_1 \ldots id_N \]
Remote Procedure Call: Example

- Client snippet:

```c
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):

```c
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):

```c
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
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

   Server

   1. Register local memory segment and get handle
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

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
2. Send serialized memory handle
3. Post put/get operation using local/deserialized remote handles

Client

Server

September 24, 2013 Mercury: Enabling Remote Procedure Call for High-Performance Computing
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
3. Post put/get operation using local/deserialized remote handles
4. Test completion of remote put/get
• Client snippet (contiguous):  

```c
/* 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):**

```c
/* 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...
  
  ```c
  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...
  
  ```c
  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
  ```c
  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
  ```c
  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:**

```c
/* 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:

```c
/* 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

Solution:

Pipelining transfers and overlapping communication/execution.

Transfers can complete while writing/executing the RPC call.
Fine-grained Transfers

- Two issues with previous example
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

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

*Data buffer (nbytes)*
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)*

```
| 0 | 1 | 2 | 3 |
```
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

Data buffer (nbytes)
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

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

*Data buffer (nbytes)*

```
          W
\    / 1  2  3
 /\  \
```

September 24, 2013

Mercury: Enabling Remote Procedure Call for High-Performance Computing
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 (n bytes)
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)
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

Data buffer ( nbytes )

[Diagram of data buffer with sections labeled E, 1, 2, 3]
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)*

```
  E 1 2 3
```

September 24, 2013
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)

---

E 1 2
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)*
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 \((n\text{bytes})\)
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)
- Scalability / aggregate bandwidth of RPC requests to single server with bulk data transfer (QDR 4X Infiniband cluster)
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)
Scalability / aggregate bandwidth of RPC requests to single server with bulk data transfer (Cray XE6)
NULL RPC request execution on Cray XE6
- NULL RPC request execution on Cray XE6
  - With XDR encoding: 23 µs

September 24, 2013 Mercury: Enabling Remote Procedure Call for High-Performance Computing
NULL RPC request execution on Cray XE6
  - With XDR encoding: 23 µs
  - Without XDR encoding: 20 µs
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
- Generate as much boilerplate code as possible for
• Generate as much boilerplate code as possible for
  • Serialization / deserialization of parameters
- 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 client and server
- 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 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
MERCURY_GEN_PROC(
    struct_type_name, fields
)

MERCURY_GEN_PROC(
    open_in_t, ((hg_string_t)(path)) 
    ((int32_t)(flags)) 
    ((uint32_t)(mode))
)
Current and Future Work

- Implement plugins that make use of true RMA capability
- ibverbs
- SSM, etc.
- Checksum parameters for data integrity
- Support cancel operations of ongoing RPC calls
- Integrate Mercury into other projects
- Mercury POSIX: Forward POSIX calls using dynamic linking
- Triton
- IOFSL
- HDF5 virtual object plugins
Current and Future Work

- Implement plugins that makes use of true RMA capability
  - ibverbs
  - SSM
  - etc

Checksum parameters for data integrity

Support cancel operations of ongoing RPC calls

Integrate Mercury into other projects

Mercury POSIX: Forward POSIX calls using dynamic linking

Triton

IOFS

HDF5 virtual object plugins
Current and Future Work

- Implement plugins that makes use of true RMA capability
  - ibverbs
  - SSM
  - etc
- Checksum parameters for data integrity
Current and Future Work

- Implement plugins that makes use of true RMA capability
  - ibverbs
  - SSM
  - etc
- Checksum parameters for data integrity
- Support cancel operations of ongoing RPC calls
Current and Future Work

- Implement plugins that make use of true RMA capability
  - ibverbs
  - SSM
  - etc
- Checksum parameters for data integrity
- Support cancel operations of ongoing RPC calls
- Integrate Mercury into other projects
  - Mercury POSIX: Forward POSIX calls using dynamic linking
  - Triton
  - IOFSL
  - HDF5 virtual object plugins
Questions

- Mercury project page
  - 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