MPI-INTEROPERABLE GENERALIZED ACTIVE MESSAGES

Xin Zhao, Pavan Balaji, William Gropp, Rajeev Thakur

University of Illinois at Urbana-Champaign
Argonne National Laboratory

ICPADS’13

December 17, 2013
Data-Intensive Applications

- “Traditional” applications
  - Organized around dense vectors or matrices
  - Regular communication, use MPI SEND/RECV or collectives
  - Communication-to-computation ratio is low
  - Example: stencil computation, matrix multiplication, FFT

- Data-intensive applications
  - Organized around graphs, sparse vectors
  - Communication pattern is irregular and data-dependent
  - Communication-to-computation ratio is high
  - Example: bioinformatics, social network analysis
Data-Intensive Applications

- Graph algorithm (BFS)

- DNA sequence assembly
  - remote search
  - local node
  - DNA consensus sequence
  - ACGCGATTTCAG
  - GCGATTCAGTA
  - ACGCGATTTCAGTA

- Common characteristics
  - Organized around sparse structures
  - Communication-to-computation ratio is high
  - Irregular communication pattern
Massage Passing Interface (MPI)

- Industry standard communication runtime for high performance computing

Process 0  Process 1
Send (data)  Receive (data)
Receive (data)  Send (data)

two-sided communication (explicit sends and receives)

Process 0  Process 1
Put (data)  
Get (data)  
Acc (data)  +=

one-sided (RMA) communication (explicit sends, implicit receives, simple remote operations)

remote node
ACGCGATTCAG
GCGATTCAGTA
ACGCGATTCAGTA
remote search

local node
DNA consensus sequence

not sufficient
MPI and Data-Intensive Applications

• Use MPI two-sided with dedicated threads / processes for data-intensive applications
  • Waste cores to wait for incoming messages
  • Examples: SWAP / Kiki (DNA assembly), MADNESS (computational chemistry)

Active Messages Paradigm

- Sender explicitly sends message
- Upon message’s arrival, message handler is triggered, receiver is not explicitly involved
- User-defined operations on remote process
- A suitable paradigm for data-intensive applications
  - Data is sent immediately
  - Communication is asynchronous
Motivation

• MPI is the most widely used parallel programming model and many applications are written by MPI
• Rewriting the entire application requires too much effort
• Enable MPI-interoperable Active Messages
  • Applications can be modified incrementally to use AM only when necessary
  • Support different capabilities in one model, user can flexibly choose which one to use
Generalized Active Messages
Accumulate-style AM

- Leverage MPI RMA interface
- Extend Accumulate-style operations to support user function (originally for MPI_REDUCE)
  - User-defined function
    - MPI_User_function (void *invec, void *inoutvec, int *len, MPI_Datatype *dtype)
  - Operation creation
    - MPI_Op_create (MPI_User_function *user_fn, int commute, MPI_Op *user_op)
  - Operation registration
    - MPIX_Am_op_register (MPI_Op user_op, int id, MPI_Win win)
    - Collective call on the window

Restrictions of Accumulate-style AM

• Target input buffer and output buffer cannot be different count and datatype
• Cannot return arbitrary data from AM handler
• How to manage temporary buffers?
• Streaming active messages is a problem: MPI does not know segmentation granularity
• Memory consistency is not clear
Streaming Active Messages

- User level segmentation — “segment”
  - User defines the minimum number of input / output elements for the execution of AM handler (e.g. query sequence + result sequence)

- System level segmentation — “pipeline unit”
  - Transparent to user
  - Each AM contains N segments, MPI can internally splits AM into multiple pipeline units, each with 1~N segments

- Beneficial when: (1) no enough buffers on target
  (2) pipeline effects
Data Buffering Requirements

- Who to allocate and maintain temporary buffers?
  - User buffers

MPIX_Am_win_buffer_attach (void *buffer, int size, MPI_Win win)
MPIX_Am_win_buffer_detach (void *buffer, MPI_Win win)

- Accommodate at least one AM segment
- Shared by all processes
- Rendezvous protocol, hand-shake operation is required

- Internal buffers (system buffers)
  - Eager protocol
  - May be not enough or even exist
Correctness Semantics
Memory Consistency / Ordering / Concurrency / Atomicity
MPI-3 RMA Memory Model

- MPI-3 provides two memory models: SEPARATE and UNIFIED.
- MPI-2: SEPARATE model
  - Logical public and private copies
  - MPI provides software coherence between window copies
  - Extremely portable to systems that don’t provide hardware coherence
- MPI-3: new UNIFIED model
  - Single copy of the window
  - System must provide coherence
  - Superset of separate semantics
    - e.g. allows concurrent local/remote access
  - Provides access to full performance potential of hardware
Interoperability

- **AM vs. RMA** – RMA operations access “public” window, whereas AM handlers access “private” window.
- In SEPARATE window model, if AM and RMA, or AM and AM, or AM and STORE, update on the same window, even on non-overlapping locations, result data is undefined.

![Diagram showing AM write, PUT, and AM write in public and private copies with STORE and invalid operation combinations.](image)
Maintaining Memory Consistency

- In both SEPARATE and UNIFIED window models, MPI runtime should ensure the consistency of window

SEPARATE window model

UNIFIED window model
Ordering

- By default three orderings are imposed:
  - AMs with different operations
  - AMs with the same operation
  - Segments within one AM

when AMs are from same origin, to same target, update on same window and overlapping memory locations

all orderings can be released using MPI info for better performance!
Concurrenty

• By default, MPI implementation is “as if” AMs are executed in some sequential order
  • If MPI implementation can know that concurrency is inconsequential (e.g. target data is non-overlapping), it can execute AMs concurrently

strict ordering may force MPI implementation to disable concurrency 😞

concurrency can be released using MPI assert for better performance!
Other Considerations

- Atomicity is not provided by MPI
  - It is OK if all accesses in AM are read-only 😊
  - If you don’t need concurrency, atomicity is not a concern
  - User can emulate atomic AM using exclusive window lock
- AM handler is not allowed to call other MPI functions
  - MPI stack needs to be reentrant-safe
  - Require higher thread level of MPI runtime to guarantee thread-safety
Evaluation
Experimental Settings

• BLUES cluster at ANL: 310 nodes, with each consisting 16 cores, connected with QLogic QDR InfiniBand
• Based on mpich-3.0.2
• Micro-benchmarks: two common operations
  • Remote search of string sequences (20 chars per sequence)
  • Remote summation of absolute values in two arrays (100 integers per array)
  • Result data is returned to origin
Streaming Active Messages

overlapping effect is best at unit size of 50

higher throughput is achieved when using more internal buffers
Impact of Internal Buffer

Using and freeing of internal buffers are balanced.

Providing more system buffers brings 1.7 times improvement.
Impact of Ordering and Concurrency

- epoch alternates between large AMs and small AMs
- concurrent AMs is enabled by “local computation”
Conclusion

- Data-intensive applications are increasingly important in many areas
- Their characteristics make them very different with traditional applications
- New parallel programming model is needed to achieve high performance and to avoid too much programming effort
- Enabling Active Messages within MPI programming model and runtime system
THANKS! 😊
BACKUP SLIDES
Active Message Trigger API

- MPIX_Am
  IN  origin_input_addr
  IN  origin_input_segment_count
  IN  origin_input_datatype
  OUT origin_output_addr
  IN  origin_output_segment_count
  IN  origin_output_datatype
  IN  num_segments
  IN  target_rank
  IN  target_input_datatype
  IN  target_persistent Disp
  IN  target_persistent_count
  IN  target_persistent_datatype
  IN  target_output_datatype
  IN  am_op
  IN  win
User-defined Handler API

- **MPIX_Am_user_function**
  - **IN** input_addr
  - **IN** input_segment_count
  - **IN** input_datatype
  - **INOUT** persistent_addr
  - **INOUT** persistent_count
  - **INOUT** persistent_datatype
  - **OUT** output_addr
  - **OUT** output_segment_count
  - **OUT** output_datatype
  - **IN** cur_num_segments
  - **IN** cur_segment_offset
Restrictions of Accumulate-style AM

Restriction #1: target input buffer and output buffer cannot be different count and datatype.
Restrictions of Accumulate-style AM

Restriction #2: cannot return arbitrary data from AM handler
Restrictions of Accumulate-style AM

Restriction #3: who to provide buffers for input and output data?
Restrictions of Accumulate-style AM

Restriction #4: MPI runtime cannot know segmentation granularity.
Restrictions of Accumulate-style AM

Restriction #5: When result data can be seen by other processes? Memory consistency is unclear.

DNA consensus sequence:
ACGCGATTCAG
GCGATTCAATA
ACGCGATTCAGTA

remote search
local node

origin process
AM
AM handler
remote node

AM response
target process