Active Messages for MPI
- would make things easier! -

Torsten Hoefler and Jeremiah Willcock

Open Systems Lab
Indiana University
Bloomington, USA

10th MPI Forum Meeting June'09
Menlo Park, CA, USA
Jun 8–10th, 2009

Motivation—What are Active Messages?

Original Active Message Motivation

- T. von Eicken, et al.: *“Active Messages: a Mechanism for Integrated Communication and Computation”* (1992)
- Overlap communication and computation
- Asynchronous communication with minimal synch.
- Low overhead injection, pipelined transport
- Potentially run in interrupt handler → no buffering

Active Messages have many Faces

- Similar to but not quite like remote procedure invocation
- Perform small non-blocking functions on message data
- Work is often inserted in a queue to be handled by main thread (cf. first-level interrupt handlers)
- Thread (and signal) safe?
- Handlers must halt for every input in each environment!

Motivation—State of the Art?

Many Middlewares Provide Active Messages

- GASNet
- IBM's LAPI and DCMF
- Myrinet's MX
- POOMA/CHEETAH
- Application layers like in PBGL . . .

Some Application Examples

- Parallel graph computations (inserting into remote queues)
- Implementing DSM systems (copying data to remote addresses without full (non-scalable or global) mapping)
- High-level language bindings (manipulating remote data structures)
- Also usable for (irregular) halo exchange (like one-sided)

Can we implement the functionality using MPI-2.2?

One-Sided?

- MPI_Accumulate goes in the direction but is too limited
- cf. D. Bonachea: *"The Inadequacy of the MPI 2.0 One-sided Communication API for Implementing Parallel Global Address-Space Languages"*

Two-Sided?

- The usual suspect—*asynchronous progress* (invocation) ⇒ polling vs. thread
- Polling would be impractical (eliminate all AM advantages)
- Thread requires *THREAD_MULTIPLE* to not limit options
- Performance penalty for threaded execution
- Is workable in practice (we have an implementation) ⇒ performance penalty is clearly visible

Should we have it in the Spec?

Disadvantages (Problems, Architectures)

- A new (but simple) concept in the spec
- Some esoteric architectures could be problematic (learn from GASNet?)
- Same problems as ticket #26 (“Add a callback function if a request completes”)

Advantages (Performance, Semantics)

- It is universal (one could implement one-sided with AM)
- Could serve well as compilation target (UPC, CAF, ...)
- It solves many high-level language binding issues
- All the original AM advantages (overlap, asynch., ...)

Option #1

The Handler Function

- (*MPI_AM_Handler)(void* userdata, const void* recvbuf, MPI_Status* status)
- MPI_AM_Register(int id, MPI_AM_Handler handler, void* userdata, int maxcnt, MPI_Datatype type, MPI_Comm c)
- MPI_AM_Deregister(int id, MPI_Comm comm)

Sending an AM

- MPI_AM_Send(const void* sendbuf, int count, MPI_Datatype type, int dest, int id, MPI_Comm comm)
- ... Isend, Bsend, Ssend?

Comments

- Only one handler per message (id)
- Special send calls
- Datatype hard-coded in handler function

Option #2

The Handler Function

- (*MPI_AM_Handler)(void* userdata, const void* recvbuf, MPI_Status* status)
- MPI_AM_Register(int tag, MPI_AM_Handler handler, void* userdata, int maxcnt, MPI_Datatype type, MPI_Comm c)
- MPI_AM_Deregister(int tag, MPI_Comm comm)

Sending an Active Message

- Just like normal point-to-point (MPI_{Is,Rs,Ss,S}end)

Comments

- Only one handler per message (tag)
- Same tag namespace and matching logic as for P2P
- Integrates well in MPI

More Options

Multiple Handlers per ID/Tag?

- MPI_AM_Register(MPI_AM_Handler handler, ..., void* userdata, MPI_Comm comm, MPI_AM_Func *f);
- MPI_AM_Deregister(MPI_AM_Func *f);

Multiple Datatypes per Handler?

- (*MPI_AM_Handler)(void* userdata, MPI_Status status)
- MPI_AM_Register(MPI_AM_Handler handler, int tag, void* userdata, MPI_Comm comm, MPI_AM_Func *f)
- Handler function has to MPI_Recv the message
- ⇒ would lead to buffering issues

Reintroducing needed Synchronization Semantics?

- MPI_AM_Quiesce(MPI_Comm comm {, int id})
- Effectively a barrier—drains all AMs

Even More Options

Enforce progress?

- MPI_AM_Flush(MPI_Comm comm, int id)
- Flushes the local AM queue (if coalescing is used)

Collective (de)registration?

- (De)register collectively
- Can simplify tag AM tables (is this significant?)
- Initialize hardware support (?)

Reply Messages?

- cf. gasnet_AMReply()
- Can be invoked in handler, would require reply handler
- Can be done on top of MPI but could be more efficient (?)

And Even More Options (credits go to GASNet Spec)

Differentiate between Message Sizes

- cf. GASNet short, medium and large AM sends
- Short—only register transfers (cf. inline)
- Medium—short data in temp buffer (cf. eager)
- Long—long data in buffer specified at sender (cf. rendezvous)
- MPI P2P philosophy is different
- Would enable small message optimizations (inlining etc.)

Separate Progression

- Program logic might depend on progression (no MPI_Wait)
- Back at the old MPI_Progress discussion
- See tickets #25 and #154

Restrictions on Handlers (the hard part)

Local Computation

- Handlers **must** terminate for any input in any env
- Normal mutex locks vs. special MPI locks (like GASNet) ?
- Need to be thread-safe (also signal-safe?)
- Should finish “quickly” (performance)

Messaging Operations

- Are not allowed to call all MPI calls (e.g., MPI_Recv)
- All local MPI calls should be allowed (e.g., MPI_Get_count)
- Restricted set of remote operations
  - Only “immediate” (fire & forget) send operations
  - Generate new AMs!
  - Special send operations?
  - Restrict to reply messages?
  - Very hairy topic

Restrictions on Handlers (the hard part)

Memory Management

- MPI allocates memory to receive messages
- The handler "uses" the data and the runtime deallocates it
- Data can be stored in fast communication memory
- Should the handler be allowed to "keep" the memory?

Progress Semantics

- We do **not** intend to change MPI's progress semantics!
- Handlers can either be asynch or synch
- Handlers should be invoked **last** before leaving an MPI call
- Handlers can also be invoked asynchronously (e.g., in threads)
- Handlers that call MPI functions would be tail-recursive calls (no problem to handle them)
- Called MPI functions follow normal progress rule
- Deadlock/race problems not worse than in threaded environments

Discussion

Thanks for input from Nick Edmonds and Marcin Zalewski!