Open MPI P2P Architecture

George Bosilca - UTK
Galen Shipman - LANL
Tim Woodall - LANL

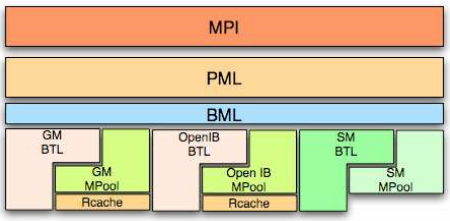
Introduction to P2P

- Based on a component architecture
  - Flexible run-time tuning
  - "Plug-ins" for different capabilities (e.g., different networks)
- Natively support commodity networks
  - Infiniband
  - Myrinet GM / MX
  - LAPI
  - Portals
  - Shared memory
  - TCP

P2P

Component Frameworks

P2P Component Frameworks



Example

- Framework
  - BTL - Byte Transfer Layer (interconnect abstraction)
  - The framework defines the interface that all components and modules export
- Component
  - BTL OpenIB - The Infiniband BTL Component provides initialization (discovery of HCA adapters, etc)
- Module
  - For the BTL OpenIB a module is created for each Active Port on HCA

MPI Layer

- Not a component
  - Located in topdir/ompi/mpi
    - C, F77, F90 and C++ specific support/bindings are located in corresponding subdirectories
    - Example source file: topdir/ompi/mpi/c/isend.c
      - MPI_Isend - calls PML level through a "helper" macro
      - PML provides support for the asynchronous send
      - In general PML level provides all messaging semantics required for MPI point-to-point

PML

- Provides MPI Point-to-point semantics
  - Standard
  - Buffered
  - Ready
  - Synchronous
- Message Progression
- Request Completion and Notification (via request objects)

PML

- Internal MPI messaging protocols
  - Eager send
  - Rendezvous
- Support for various types of interconnect
  - Send/Recv
  - RDMA
  - Hybrids

PML Interfaces

- pml_add_procs - peer resource discovery (via BML)
- pml_del_procs - clean up peer resources (via BML)
- pml_progress - progress BTLS (via BML)
- pml_add_comm - add PML data structures to the communicator
- pml_del_comm - remove PML data structures from communicator
- pml_irecv_init - Initialize persistent receive request
- pml_irecv - Asynchronous receive
- pml_isend_init - Initialize persistent send request
- pml_isend - Asynchronous send
- pml_iprobe - Probe receive queues for match
- …. Mirrors MPI interfaces

PML

- Framework located in topdir/ompi/pml
  - Interfaces defined in topdir/ompi/pml/pml.h
- 2 Components currently available in this framework
  - OB1 -Default
  - DR - Data reliability (under development)
- OB1 found in topdir/ompi/pml/ob1

PML

- OB1 Component
  - Defined in topdir/ompi/mca/pml/ob1/pml_ob1.h
  - Each function prototype defined by the framework is implemented by the component and modules
- OB1 instantiates a single module
  - Think of this as a singleton in object speak

BTL

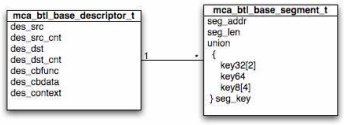
- BTL - Byte Transfer Layer
  - Provides abstraction over the underlying interconnect
  - A simple tag based interface for Send/Recv similar to active messaging
  - Provides facilities for RDMA operations including preparing memory registrations
  - Supports both Put and Get RDMA operations
  - Provides completion callback functions

BTL Interfaces

- btl_add_procs - discover peer resources and setup endpoints to the peer
- btl_del_procs - remove resources allocated to remote peer
- btl_register - register a active message callback
- btl_alloc - allocate a descriptor
- btl_free - free a descriptor
- btl_prepare_src - prepare a source descriptor
- btl_prepare_dst - prepare a destination descriptor
- btl_send - send a descriptor to an endpoint
- btl_put - put a descriptor to an endpoint (RDMA write)
- btl_get - get a descriptor from an endpoint (RDMA read)

BTL descriptor

- The BTL descriptor contains a list of source/destination segments, completion callback function and callback data



BTL support

- Infiniband - OpenIB/MVAPI
- Myrinet - GM/MX
- Portals
- Process Loopback - Self
- Quadrics - ELAN 4 - under development
- Shared Memory - SM
- TCP
- uDAPL - under development

Open IB BTL

- OpenIB BTL Component
  - Provides support for Infiniband HCAs through the OpenIB Driver and Libs
  - Uses RC based communication
  - Send/Recv including inline data
  - SRQ support
  - RDMA support (read/write)
  - Small message RDMA (ala Gleb)

GM BTL

- GM BTL Component
  - Provides support for Myrinet GM API
  - Send/Recv
  - RDMA support (put/get)
  - GM mpool abstracts memory registration
  - Supports PML pipeline protocol

BML

- BML - BTL Management Layer
  - Provides a thin multiplexing layer over the BTL's (inline functions)
  - Manages peer resource discovery, allowing multiple upper layers to use the BTLs
  - Allows for simple round robin scheduling across the available BTL's

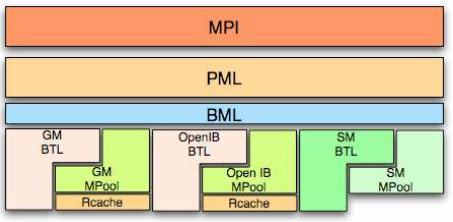
Mpool

- Mpool - Memory Pool
  - Provides memory management functions
    - Allocate
    - Deallocate
    - Register
    - Deregister
  - May be used by various other components
    - BTL - on demand registration and pre-allocated fragments
    - PML - allows pml to make protocol decisions based whether the user's buffer is registered with an mpool
    - MPI - provides a simple sollution for MPI_Alloc_mem

Rcache

- Rcache - Registration Cache
  - Provides memory registration caching functions
    - Find
    - Insert
    - Delete
  - Currently used by memory pools to cache memory registrations for RDMA capable interconnects
  - Implemented as a Red Black Tree in the RB Component although a variety of caching techniques could be used by simply adding another Rcache Component.

P2P Components Frameworks



P2P

Component Initialization

MPI_Init

- Opens PML Components (OB1 and DR, defaulting to OB1)
  - PML opens the BML Component (R2)
  - BML opens all BTL Components
- Init PML Component
  - PML Inits the BML Component
    - BML Init returns a singleton module
  - BML Inits the BTL Components
    - BTL Init returns a list of modules
  - PML registers active message callbacks with each BTL module

Resource Discovery

- Most of the work is at the BTL level during open/init
  - Local resource discovery, for OpenIB this includes:
  - opening devices
  - checking for active ports
  - creating a module for each active port
  - Publish port information to the GPR
  - Set recv call back for OOB messages (dynamic connection establishment)

Determining Peer Reachability

- MPI_Init creates a list of process structures via ORTE for each peer in MPI_COMM_WORLD
- Calls add procs on the PML passing in the process list
- PML calls add procs on the BML
  - Call add procs on each BTL passing the list of process structures
  - For each peer the BTL creates an endpoint data structure which will represent a potential connection to the peer and caches the peers addressing information

Determining Peer Reachability

- Add procs continued
  - After the BTL endpoint is created the BML creates a data structure to cache the BTL endpoint and module used to reach a peer
  - The BML caches an array of these data structures grouping them by BTL functionality
    - btl_eager - used for eager frags (low latency)
    - btl_send - send/receive capable
    - btl_rdma - RDMA capable

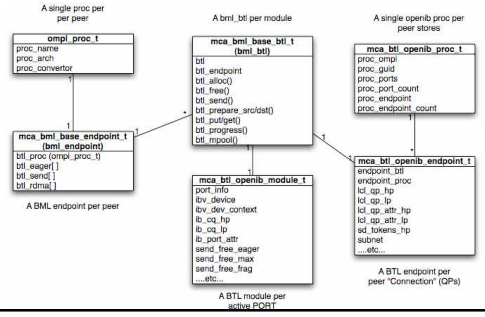
OpenIB Peer Resource Discovery

- Open IB BTL example
  - Prior to add procs the GPR distributes all the peers addressing information to every process
  - BTL will query peer resources (Number of ports/lids from cached GPR data)
  - BTL endpoint is created, matching the BTL module's port/subnet with the peer's
  - Note that a connection is not yet established and will be wired up on the first message sent to the peer via the endpoint

OpenIB Peer Resource Discovery - Continued

- OpenIB BTL example continued
  - A mirror of the ompi_proc_t structure is created at the BTL level
  - Caches information from the OMPI proc
  - Stores port information
  - Stores an array of endpoints to the peer used in establishing connections based on port/subnet

Data Structures



Send / Receive

Send (MPI_Send)

- Does any parameter validation (if enabled)
- Calls the PML interface
  - mca_pml.pml_send()

Send Request Init (mca_pml_ob1_send)

- Allocate a send request (from PML free list)
- Initialize the send request
  - Lookup ompi_proc_t associated with the dest
- Create (copy) and initialize the converter for this request
  - Note that a converter is cached for the peer on the pml proc structure based on peer architecture and user datatype Start the send request

Send Request Start

- Cached on the ompi_proc_t is a BML endpoint
  - contains a list of BML_BTL data structures (combines the BTL module and the BTL endpoint for the peer)
- The next BML_BTL is selected round robin for this request
- Small messages are scheduled via mca_pml_ob1_send_request_start_copy

Eager Send (mca_pml_ob1_send_request_start_copy)

- The PML will allocate a send descriptor by calling mca_bml_base_alloc
  - specifying the amount of the message to send (up to eager limit) plus reserve for headers
  - The send descriptor is allocated by the BTL from a free list
  - An Mpool associated with the BTL is used to grow the free list if necessary (may use pre-registered memory)
- The converter is then used to pack the user data into the send descriptor
- Header information is populated including the tag value (for active message callback)

Eager Send - Continued

- A callback is set on the descriptor and the send request is set as callback data
- The descriptor is ready for sending mca_bml_base_send is called
- On sender side completion, the descriptor's callback function is called along with the callback data (send request)
- The callback is a PML function which returns the send request and frees the descriptor

Open IB BTL Send Example

- mca_bml_base_send calls the BTL level send, passing in the endpoint and module
- If this is the first descriptor to the peer
  - Queue the descriptor at the BTL
  - Initialize the QP locally
  - Send the QP information to the peer via the OOB (triggers the recv callback registered with the OOB)
  - On receipt of the peers QP information finish establishing the QP Connection
  - Send any queued fragments to the peer

Receive - Posting

- MPI_Recv simply calls the PML recv (mca_pml_ob1_recv)
- Allocate a recv request (from global free list)
- Initialize the recv request
  - Lookup ompi_proc_t associated with the dest
- Unlike the send request, the recv request does not initialize a converter for the request until the recv is matched (saves resources)
- Start the recv request
  - Check the unexpected recv list for the match
  - If not found post it to the right list for matching later

Receive - Fragments

- Messages are received via the progress engine
- For polling progress mca_bml_r2_progress is registered as a progress function and is called via orte_progress
- mca_bml_r2_progress loops through the BTL's and calls a component level progress function
- Receiving data is BTL specific
- After receipt of the data BTL progress will lookup and invoke the active message callback based on the tag value specified in the message header passing in the descriptor

Receive - Active Message Callback

- Recall the active message callback was registered earlier with the BTL
- PML OB1 uses a single active message callback mca_pml_ob1_recv_frag_callback
- The callback is specific to the type of send that was initiated
  - for small eager messages the receiver will attempt to find a match by calling mca_pml_ob1_recv_frag_match

Receive - Matching

- If the message is matched
  - Copy and initialize a converter for the request
  - Note that a converter is cached for the peer on the pml proc structure based on peer architecture and user datatype
  - mca_pml_ob1_recv_request_match is called
- Otherwise the data is buffered and the match is posted to the unexpected list

Receive - Unpack

- Assuming the receive is matched
- With the converter now initialized the data is unpacked into the user's buffer
- A small message (less than eager limit) is now complete and the receive request is signaled complete at the MPI level
- The PML level resources are then released and the request returned to the global free list
  - For non-blocking receives the request is not freed until MPI_Test or MPI_Wait
- Note that the BTL descriptor is only valid for the life of the active message callback so the descriptor must be unpacked into the user's buffer or buffered at the PML level

"Leave Pinned"

"Leave Pinned"

- For contiguous data
- Uses the Mpool to register the entire buffer up front and cache the registration via the Rcache
- Initiate a single RDMA read or write (per available RDMA BTL)
- Subsequent send/receive from the same user buffer can avoid registration by searching for the registration in the Rcache

"Leave Pinned"

- Only for messages over the eager limit
- Requires a Rendezvous
  - For RDMA Write the receiver must be notified of the request (match) and responds with the target address, the sender then "puts" the data
  - For RDMA Read the receiver is notified of the request and given the senders address, the receiver then "gets" the data
- For registration cache coherency memory hooks are used to detect changes in virtual/physical mappings or sbrk is disabled

"Leave Pinned"

- MPI_Send --> mca_pml_ob1_send
- Check user buffer contiguous = true
- Find a cached registration (mca_pml_ob1_rdma_btls)
  - Each BTL module is cached in the bml_endpoint, loop through each BTL
    - Check if the user buffer is registered mpool_find(…)
    - If no registration is found, register the user buffer and cache it using mpool_register(….. , MCA_MPOOL_FLAGS_CACHE)
    - Mpools use the Rcache to cache/search the registration

Send Rendezvous (mca_pml_ob1_send_request_start_rdma)

- mca_pml_ob1_send_request_start_rdma is called passing in the registration found/registered previously
- If the BTL supports RDMA Read and there is only one RDMA BTL available
  - Prepare a source descriptor for the entire message
  - Send an RDMA GET control message including the descriptor segments
- Otherwise send a rendezvous header (no eager data is sent)

Rendezvous Received mca_pml_ob1_recv_frag_callback

- On the receipt of the fragment the active message callback mca_pml_ob1_recv_frag_callback is called
- Matching logic is called for a rendezvous header (mca_pml_ob1_recv_frag_match)
  - If the receive is not posted, insert the fragment on the unexpected list
  - When the receive is matched generate an ack and schedule the receive request

Generate ACK mca_pml_ob1_recv_request_ack

- Using the converter check if the user buffer is contiguous (ompi_convertor_need_buffers)
- Find a cached registration (or register and cache if not found)
- Cache the available registrations and corresponding BTLs on the receive request
- Generate an ack to the peer setting RDMA offset to zero
  - The RDMA offset tells the sender how much data to schedule via send/receive
  - An offset of zero indicates the receiver will schedule the entire message

Receiver Side Schedule mca_pml_ob1_recv_request_schedule

- Schedule the message over the available registrations/BTLs.
- Fragment the message based on a static BTL weighting factor (currently percentage of configured bandwidth)
- For each BTL, use the BML/BTL mca_bml_base_prepare_dst interface to create a descriptor as the destination of the RDMA

Receiver Side Schedule - Continued

- For each BTL, send a control message (PUT) with the RDMA target address to the source
  - Use the BTL on which the RDMA should be scheduled
  - Include a reference to the RDMA descriptor at the destination (sent back on RDMA completion)

Sender - PUT Control Message Received

- On receipt of the "PUT" control message
  - mca_pml_ob1_send_request_put is called
  - The sender side registration is obtained from the mpool/rcache for this BTL
  - A descriptor corresponding to the user buffer is obtained via
    - mca_bml_base_prepare_src

Sender Initiates RDMA Write mca_pml_ob1_send_request_put

- The descriptor is updated to point to the destination segments returned in the PUT control message
- mca_pml_ob1_put is called on the descriptor
- On local completion the descriptor callback is invoked
  - mca_pml_ob1_put_completion

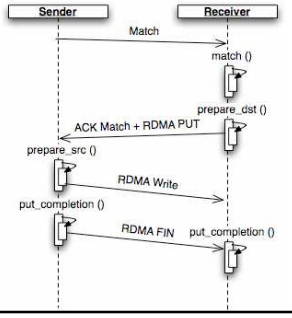
Sender Side RDMA Completion mca_pml_ob1_put_completion

- The send request is updated w/ bytes transferred
- If bytes transferred == message size, MPI completion is signalled
- A descriptor is allocated
- An RDMA FIN control message is sent to the peer to indicate remote completion
  - included in this message is the RDMA write descriptor reference (for the receiver)
- Free the RDMA descriptor btl_free
- On local completion the descriptor used for the control message is returned via btl_free

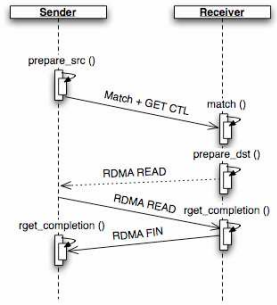
Receiver - RDMA FIN received

- A reference to the receiver's RDMA descriptor is included in the RDMA control message
  - The local completion callback associated with this descriptor is called (simulates remote completion)
    - mca_pml_ob1_put_completion
    - Different function.. Same name :-(
- The RDMA descriptor is freed (mca_bml_base_free)
- The MPI request is signalled complete

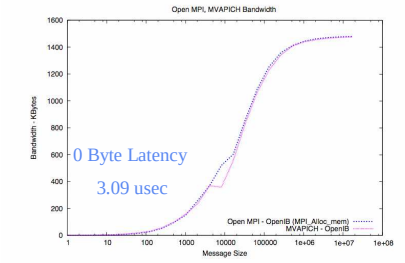
"Leave Pinned" - PUT



"Leave Pinned" - GET



Results

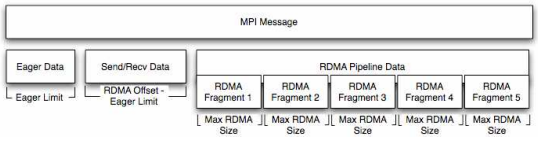


Pipeline Protocol

Pipeline Protocol

- For contiguous data
- Messages larger than BTL max send size
- Overlaps memory registration with RDMA operations
- Uses Mpool to register memory in chunks (BTL max RDMA size)
- Initiate multiple RDMA operations at once (up to BTL pipeline depth)

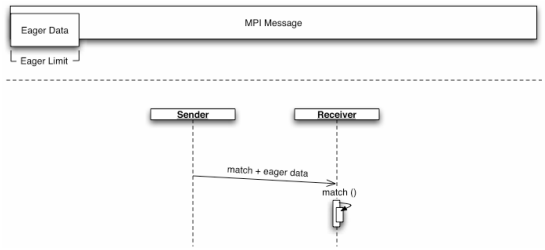
Pipeline Protocol - Message Layout



Pipeline Protocol

- Starts like Send/Recv Rendezvous protocol
  - Select next BTL from BML Endpoint's eager BTL list
  - Allocate a descriptor using the BML/BTL
  - Build a RNDV match header and pack eager data using converter
  - Send RNDV match+Eager

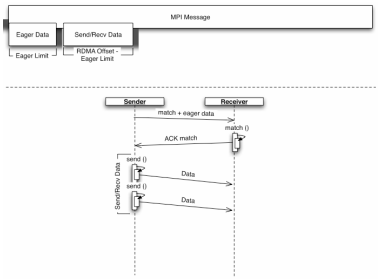
Pipeline Protocol - Eager+Match



Receive RNDV Header

- On match of RNDV header
  - Generate a RNDV ACK to the peer with the RDMA offset
    - RDMA offset is the minimum of the MIN_RDMA_SIZE of the RDMA devices available on the receiver
- On receipt of the RNDV ACK the source:
  - The source schedules up to the RDMA offset using send/recv semantics
  - This helps cover the cost of initializing the pipeline on the receive side

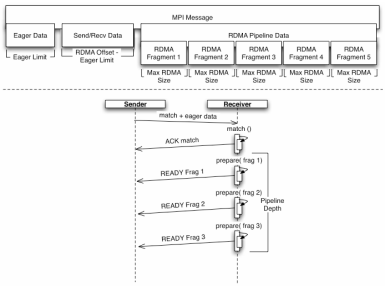
Pipeline Protocol - Send up to RDMA Offset



Receiver Schedules RDMA Fragments

- After generating the RNDV ACK the receiver immediately begins scheduling RDMA fragments
  - The next BTL is selected from the BML endpoint's RDMA BTL array
  - Prepare dest is used to register a segment of the user buffer
  - Each segments size is constrained by the BTL MAX_RDMA_SIZE
  - Send a PUT control message (including the segment list of the RDMA descriptor) to the source using the same BTL

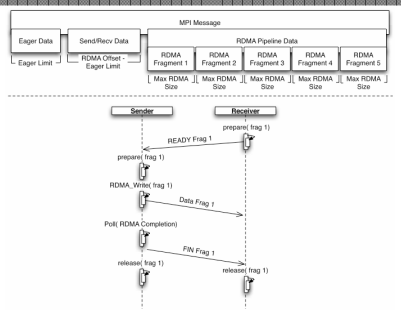
Pipeline Protocol - Register Receiver Side "chunks"



Source receives PUT control message

- Similar to the “Leave Pinned” protocol:
  - Calls prepare source to build a descriptor for the RDMA operation
    - The PUT control message specifies offset and size
    - BTL Registers the user’s buffer if required via the MPool
  - Setup the descriptor to point to the destination segments as specified in the PUT control message
  - Initiate the RDMA Write
  - On local completion of the RDMA Write send a FIN control message to the peer

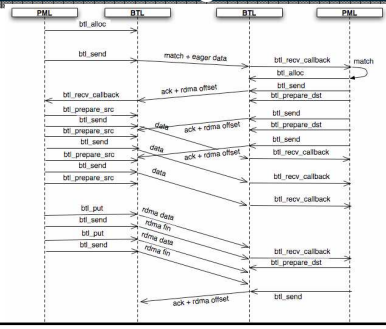
Pipeline Protocol - Register and RDMA



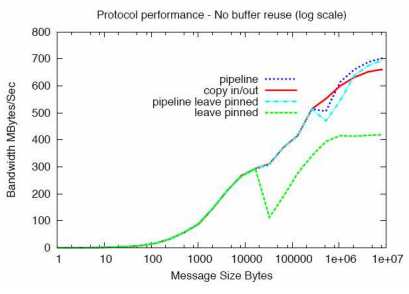
FIN message received

- Included in the FIN message is the pointer to the receiver’s RDMA descriptor
  - The RDMA descriptor’s callback is invoked ☐(simulating remote completion)
  - Update receive request status
    - may signal MPI completion
  - Free the RDMA descriptor
    - may unregister the user’s buffer

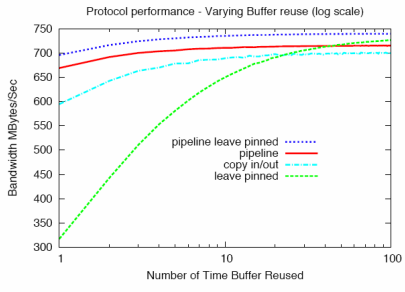
Pipeline Protocol - Timing Diagram



Pipeline Performance



Pipeline Performance



Threading Strategies

Enabling Thread Support

- Compile Time Decision
  - --enable-mpi-threads
    - Supports multiple user threads in the library
    - Thread locks disabled unless initialized as MPI_THREAD_MULTIPLE
  - --enable-progress-threads
    - Threads used internally by the library to enable asynchronous progress
    - Thread locks always enabled

Coding Standards

- Critical sections protected by atomic operations and/or mutexes
- Macros optionally enable locking based on configure options

```
OPAL_THREAD_LOCK(&lock)
/* critical section */
OPAL_THREAD_UNLOCK(&lock)
```

- By default these are compiled out

Coding Standards (Cont)

- Abstractions for mutexes and condition variables provided in:
  - opal/threads/mutex.h condition.h
- Atomic operations provided in opal/sys/atomic.h
  - opal_atomic_add32(volatile int32_t*, int)
- Wrappers for conditional atomics provided in opal/threads/mutex.h
  - OPAL_THREAD_ADD32(volatile int32_t*, int)

PML Request Completion

- Global mutex (ompi_request_lock) protects changes to request state
- Global condition variable (ompi_request_cond) used to wait on request completion
- Condition variables provide an abstraction that supports progression with multiple threading models

Single Threaded

- Multiple implementations of opal_condition_t
- Single Threaded
  - opal_condition_wait() spins calling opal_progress() until condition variable is signalled.

Progress Threads

- User level thread polls via opal_progress() for completion
- Blocks on posix condition variable after configurable number of cycles
- BTL threads progress requests asynchronously
- Upcalls from BTL->PML will signal condition variable on request completion.

MPI_Request_Wait

```
int ompi_request_wait(ompi_request_t ** req_ptr, ompi_status_public_t * status)
{
    ompi_request_t *req = *req_ptr;
    if(req->req_complete == false) {
#if OMPI_ENABLE_PROGRESS_THREADS
        /* poll for completion */
        if(opal_progress_spin(&req->req_complete))
            goto finished;
#endif
        /* give up and sleep until completion */
        OPAL_THREAD_LOCK(&ompi_request_lock);
        ompi_request_waiting++;
        while (req->req_complete == false) {
            opal_condition_wait(&ompi_request_cond, &ompi_request_lock);
        }
        ompi_request_waiting--;
        OPAL_THREAD_UNLOCK(&ompi_request_lock);
    }
#if OMPI_ENABLE_PROGRESS_THREADS
finished:
#endif
    ...
```

PML Locking

- Multiple threads could be attempting to progress a pending request
  - Utilize a per request counter and atomic operations to prevent multiple threads from entering scheduling logic
- MPI queues (unexpected,posted receives) are stored on a PML datastructure associated with the communicator
  - Utilize a single per communicator lock that is held during matching

BTL Locking

- Per BTL mutex acquired when accessing BTL specific queues/state
- Atomic operations used to manage token counts (e.g. number of send tokens available)
- Free lists (ompi_free_list_t) each maintain a mutex to protect against concurrent access

Progress Threads (--with-progress-threads)

- BTL is responsible for progressing all requests asynchronously
- Depending on the interconnect, this may require a thread per BTL
- Myrinet/GM - we create a thread that blocks in the driver
  - gm_blocking_receive_no_spin()

Progress Threads

- To improve latency, we can allow user thread to progress events for a configurable number of cycles
- If requests do not complete, allow BTL thread to complete them asynchronously
- Need to define a standard mechanism to signal/wake BTL thread in this case