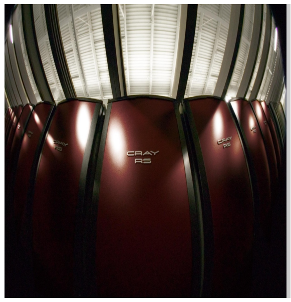


Open MPI Data Transfer

Brian W. Barrett

Scalable System Software

Sandia National Laboratories

bwbarre@sandia.gov

December 3, 2012

SAND Number: 2012-10326P

Exceptional service in the national interest

Outline…

- Introduction
- MPI Communication models
- Open MPI Overview
  - Foundational Components
  - Communication in Open MPI
  - Communication deep-dive
- Future plans
  - BTL relocation
  - Fault tolerance
  - Threads

Thanks!

- Many of these slides / diagrams borrowed from previous Open MPI talks. A number of people deserve thanks:
  - George Bosilca, University of Tennessee
  - Ralph Castain, Greenplum
  - Galen Shipman, Oak Ridge
  - Jeff Squyres, Cisco

Outline…

- Introduction
- MPI Communication models
- Open MPI Overview
  - Foundational Components
  - Communication in Open MPI
  - Communication deep-dive
- Future plans
  - BTL relocation
  - Fault tolerance
  - Threads

Point-to-point Communication

- MPI provides ordered, tagged messaging with wildcards
  - Ordered: messages sent from rank A to rank B on the same communicator (channel) and tag are FIFO ordered
  - Tagged: In addition to channel semantics, messages have tags which are used to select a message during receive operaions
  - Wildcards: Both the source of the message and tag can be wildcarded to enable more flexible receive operaions
- MPI allows unexpected messages
  - Implementation handles buffering (this is hard!)
- Send and receive have blocking and non-blocking variants

Collective communication

- Common collective operations in scientific computing
  - Barrier
  - Broadcast
  - All-to-all
  - Gather/Allgather
  - Scatter
  - Reduce/Allreduce (global sum, product, min, max, etc.)
- Blocking and non-blocking variants

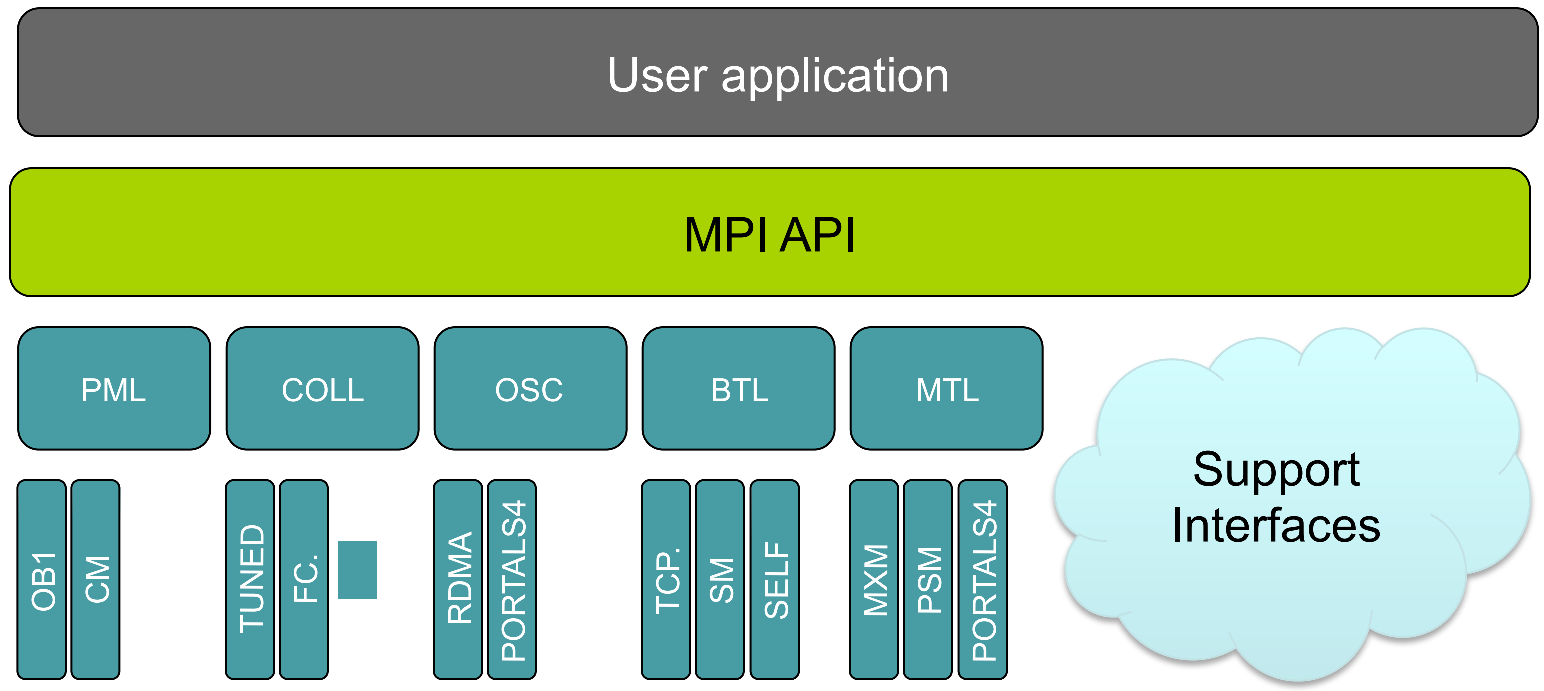
One-sided

- Put, get, atomic, fetch atomic operations
- MPI-2 fixed remote accessible segment
  - Expanded slightly in MPI-3
  - Remote addressing is an offset into the segment
- Explicit synchronization of memory spaces
  - Active or passive target synchronization
- Well defined memory model (which is a talk all its own…)

Outline…

- Introduction
- MPI Communication models
- Open MPI Overview
  - Foundational Components
  - Communication in Open MPI
  - Communication deep-dive
- Future plans
  - BTL relocation
  - Fault tolerance
  - Threads

High-level overview



Process Naming

- ORTE process name (orte_process_name_t)
  - Structure which can be compared
  - Values important, not the structure itself
- OMPI process (ompi_proc_t)
  - MPI information for every connected process
  - Never copied; always passed by pointer (pointers comparable!)
  - Multiple useful pieces of information
    - Pointers to opaque communication structures
    - Architecture information for remote process
    - Datatype convertor
    - Processor Affinity flags
- ORTE interfaces use ORTE process name, most OMPI functions use OMPI process

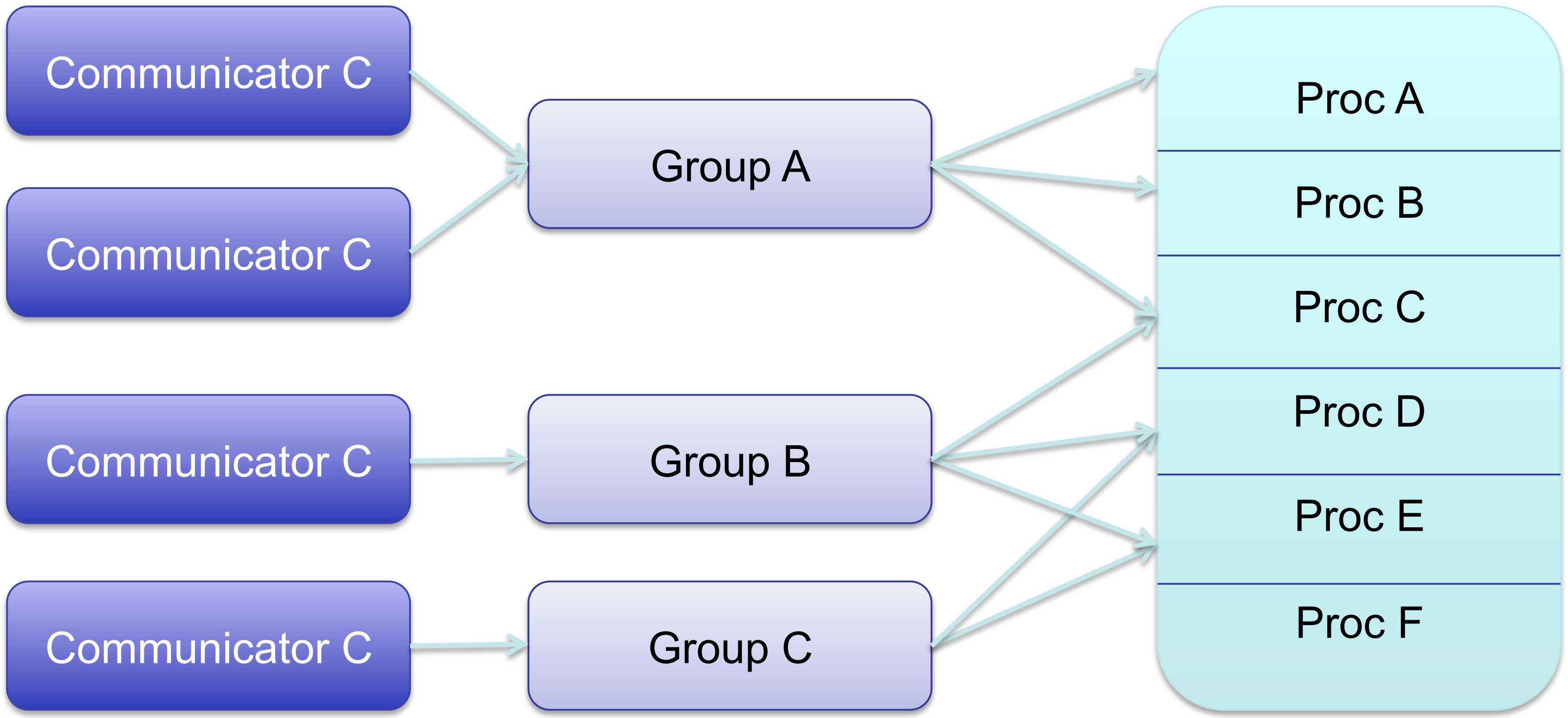
MPI Communicator

- Virtual communication channel
- Communicator structure encapsulates channel
- Useful MPI structures:
  - cid: channel identifier (system makes sure a given communicator has a single, unique cid)
  - Local and remote groups: set of ranks (endpoints) in the communicator
  - Collective modules and data
  - PML communicator opaque pointer

MPI Groups

- A set of nodes indexed by their rank (pointers to their process structure).
- All the functions required to create sub-groups from other groups:
  - Union, inclusion, exclusion, intersection, difference
  - Range inclusion and range exclusion.
- Holds the (ordered) list of OMPI proc structures related to the group

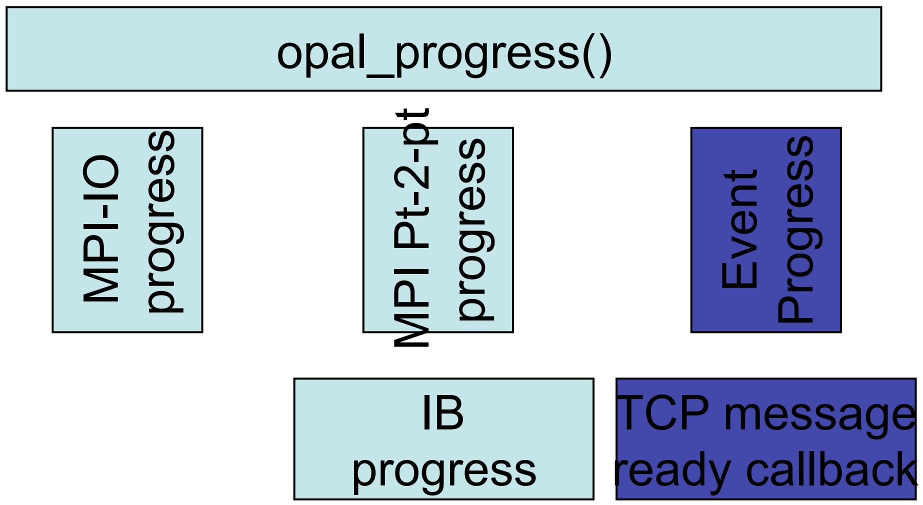
To review…



Each OMPI process has an associated ORTE process name

Progress Engine

- opal_progress() triggers callbacks to registered functions
- Event library for complicated progression
  - triggers for file descriptors (like select, but with callbacks)
  - Timer callbacks
  - Signal callbacks (not in signal handler context!)
  - Event library can run in own thread



MPI Datatypes

- MPI provides datatype engine for describing communication buffers for all communication options
  - Heterogeneous datatype conversion
  - Build transformation engines (matrix transpose)
  - Pack/unpack strided data
- Central data type engine provides DDT processing semantics
- Two layers:
  - Datatype engine for users to describe buffer layout
  - Convertor engine for communication buffer packing
- Unfortunately, can’t ignore convertors entirely…

Datatypes: Predefined

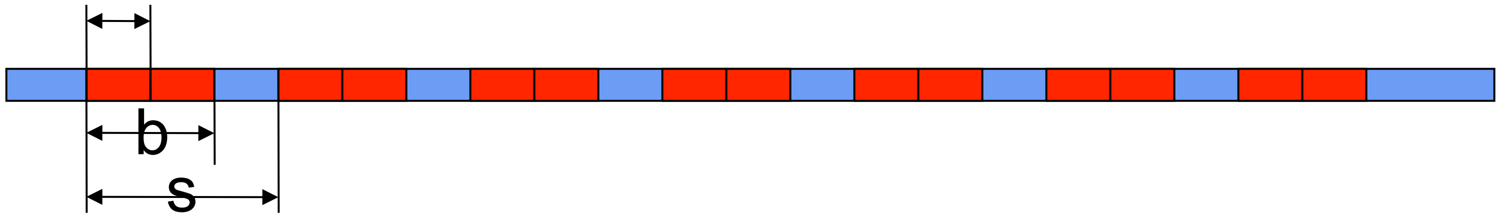
MPI_Datatype	C datatype	Fortran datatype
MPI_CHAR	signed char	CHARACTER
MPI_SHORT	signed short int	INTEGER*2
MPI_INT	signed int	INTEGER
MPI_LONG	signed long int	
MPI_UNSIGNED_CHAR	unsigned char	
MPI_UNSIGNED_SHORT	unsigned short	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long int	
MPI_FLOAT	float	REAL
MPI_DOUBLE	double	DOUBLE PRECISION
MPI_LONG_DOUBLE	long double	DOUBLE PRECISION*8

Datatypes: User-defined

- Applications can define unique datatypes
  - Composition of other datatypes
  - MPI functions provided for common patterns
    - Contiguous
    - Vector
    - Indexed
    - …
- Always reduces to a type map of pre-defined datatypes

Datatypes: Vectors

- Replication of a datatype into locations that consist of equally spaced blocks
  - MPI_Type_vector(7, 2, 3, oldtype, newtype)



Datatypes: Convertor

- Created based on 2 architectures: local and remote.
- Once the data-type is attached is can compute the local and remote size
- Can convert the data segment by segment: iovec conversion
  - For performance reasons there is no room for recursivity

Datatypes: Convertor

- Creating a convertor is a costly operation
  - Should be avoided in the critical path
  - Master convertor
  - Then clone it or copy it (!)
  - Once we have a initialized convertor we can prepare it by attaching the data and count
    - Specialized preparation: pack and unpack
- Position in the data: another costly operation
  - Problem with the data boundaries …

Datatypes: Convertor

- Once correctly setup
  - Pack
  - Unpack
- Checksum computation
- CRC
- Predefined data-type boundaries problem
- Convertor personalization
  - Memory allocation function
  - Using NULL pointers

Datatypes: Convertor

- Sender
  - Create the convertor and set it to position 0
  - Until the end call ompi_convertor_pack in a loop
  - Release the convertor

- Receiver
  - Create the convertor and set it to position 0
  - Until the end call ompi_convertor_unpack in a loop
  - Release the convertor

Easy isn't it ?!

Datatypes: Convertor

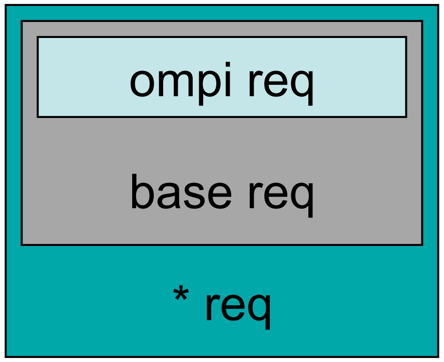
- In fact the receive is more difficult
  - Additional constraints
    - Fragments not received in the expected order
    - Fragments not received (dropped packets)
    - Fragments corrupted
    - Fragments stop in the middle of a predefined data-type …
  - Do we look for performance ?

MPI Requests

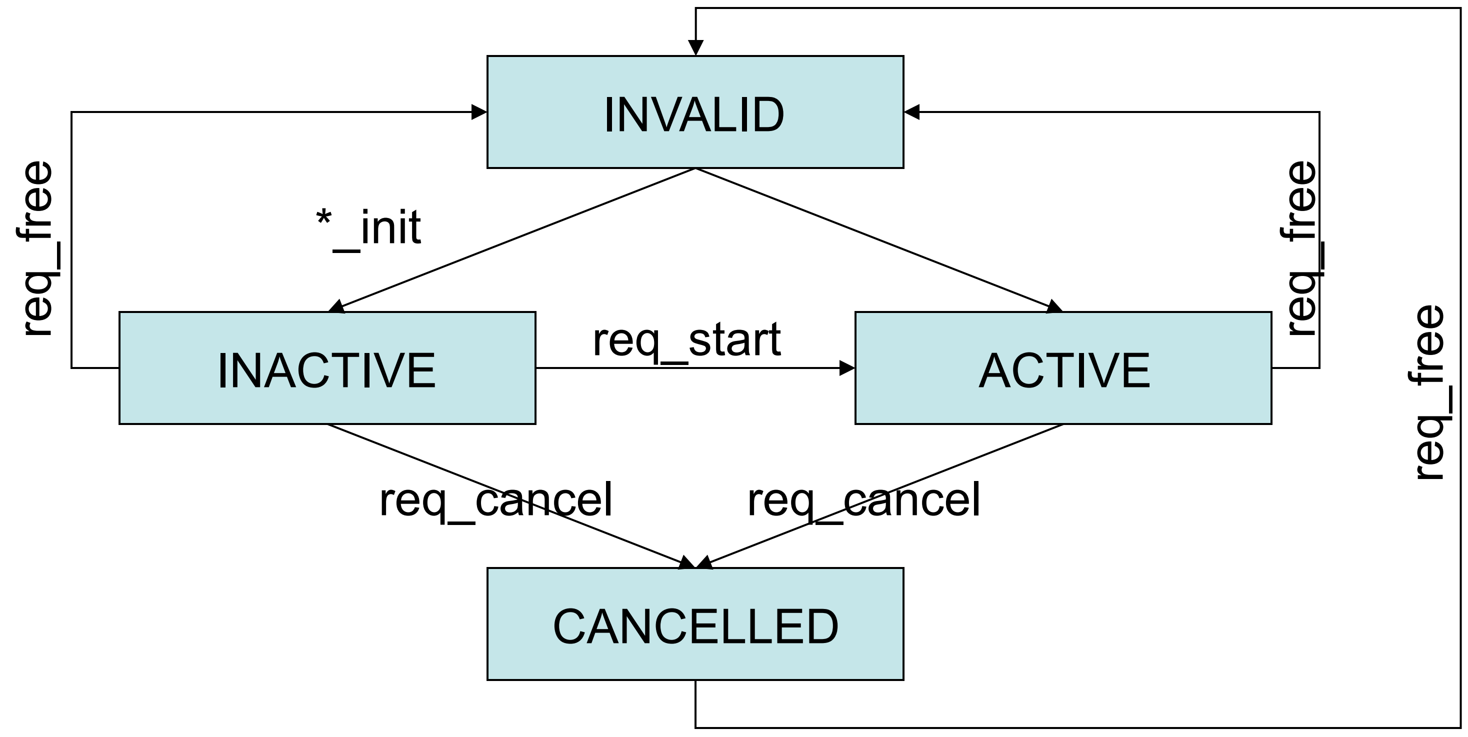
- Used for:
  - Communications (point-to-point, one sided)
  - MPI I/O
  - Generalized requests
- Fortran indexes
  - Created only if required
  - Removed when the request is freed internally

Requests: Overview

- Inheritance
  - Share the data between layers
- OMPI store general information
- The PML base request store the user level information
- The specific PML request store all other information required by this specific implementation of the PML.



Requests: Life cycle



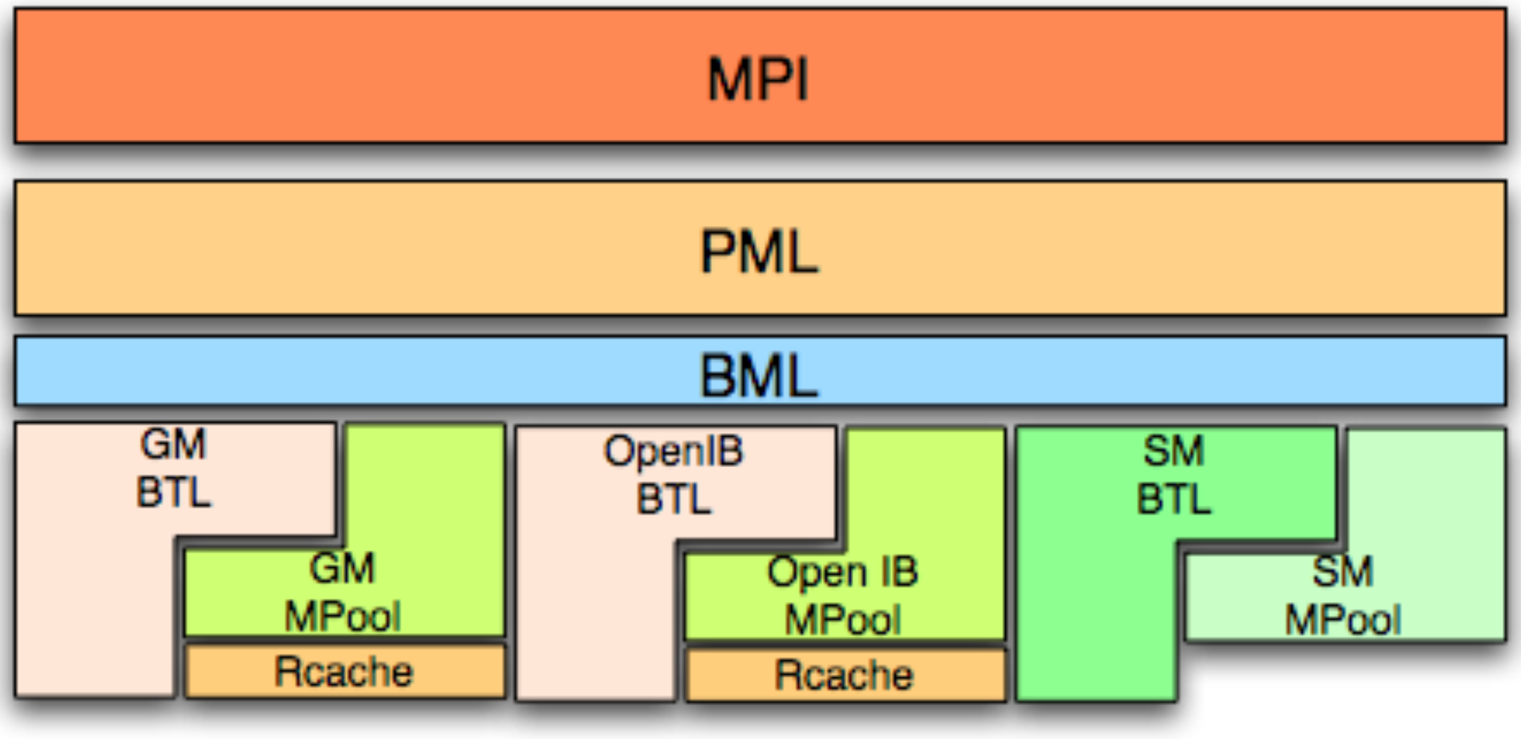
Outline…

- Introduction
- MPI Communication models
- Open MPI Overview
  - Foundational Components
  - Communication in Open MPI
  - Communication deep-dive
- Future plans
  - BTL relocation
  - Fault tolerance
  - Threads

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

P2P Component Frameworks



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_enable - initialize communication structures
- 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
    - Software MPI matching
    - Multi-device (striping, device-per-endpoint, etc.)
    - Utilize BTLs for device-level communication
  - CM
    - Offload MPI matching
    - Single device
    - Utilize MTLs for device-level communication
- OB1 found in <topdir>/ompi/pml/ob1

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

BTL

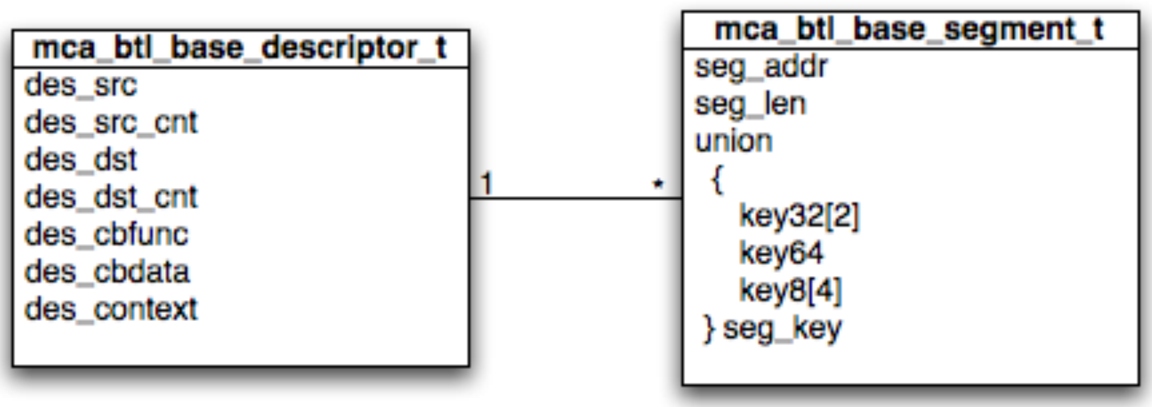
- Byte Transfer Layer
- Provides abstraction over the underlying interconnect
- A simple tag based interface for communication similar to active messaging
- Provides facilities for RDMA operations including preparing memory registrations
- Supports both Put and Get RDMA operations
- Provides completion 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: Devices

- Supported
  - openib – Open Fabrics (reliable connection)
  - self – send to self semantics
  - sm – shared memory
  - smcuda – shared memory with CUDA support
  - tcp
  - ugni – Cray Gemini/Aries
  - vader – shared memory using XPMEM/Cross-mapping
- Unsupported
  - mx – Myrinet Express
  - sctp
  - wv – Windows VERBS

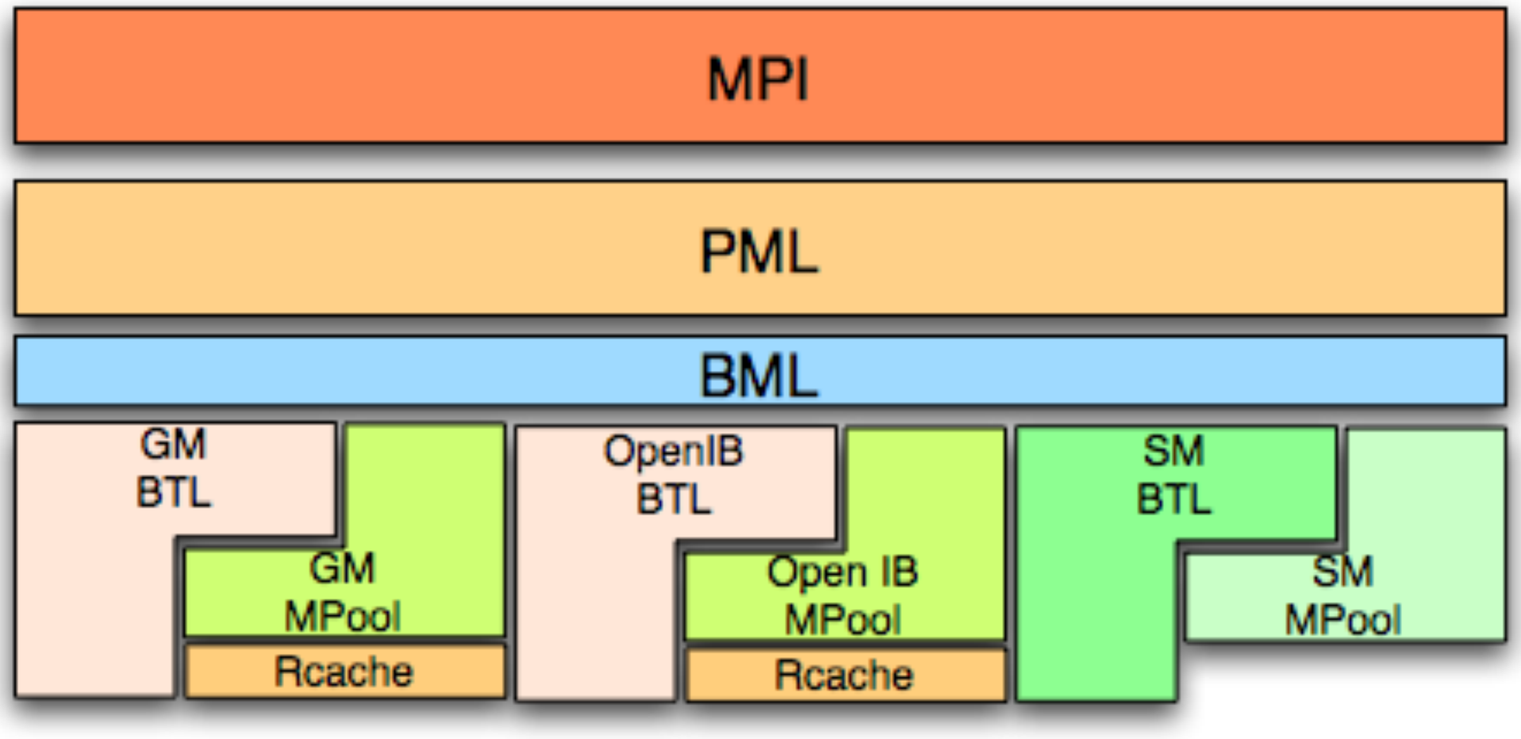
Mpool

- 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 Component Frameworks



Outline…

- Introduction
- MPI Communication models
- Open MPI Overview
  - Foundational Components
  - Communication in Open MPI
  - Communication deep-dive
- Future plans
  - BTL relocation
  - Fault tolerance
  - Threads

Example: OB1 PML / Open IB BTL

- Open IB:
  - Provides support for Infiniband HCAs and many RDMA over Ethernet devices
  - Uses RC based communication
  - Send/Recv including inline data
  - SRQ support
  - RDMA support (read/write)
  - Small message RDMA

Startup

- Decide which PML being used
- Initialize BTLs/MTLs as needed
  - Resource discovery
  - Create BTL modules for each endpoint
  - Initialize channel creation mechanism, if needed
- Modex: scalable data share
- Initialize peer informaiton
  - MPI_Init builds OMPI process structure array
  - 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

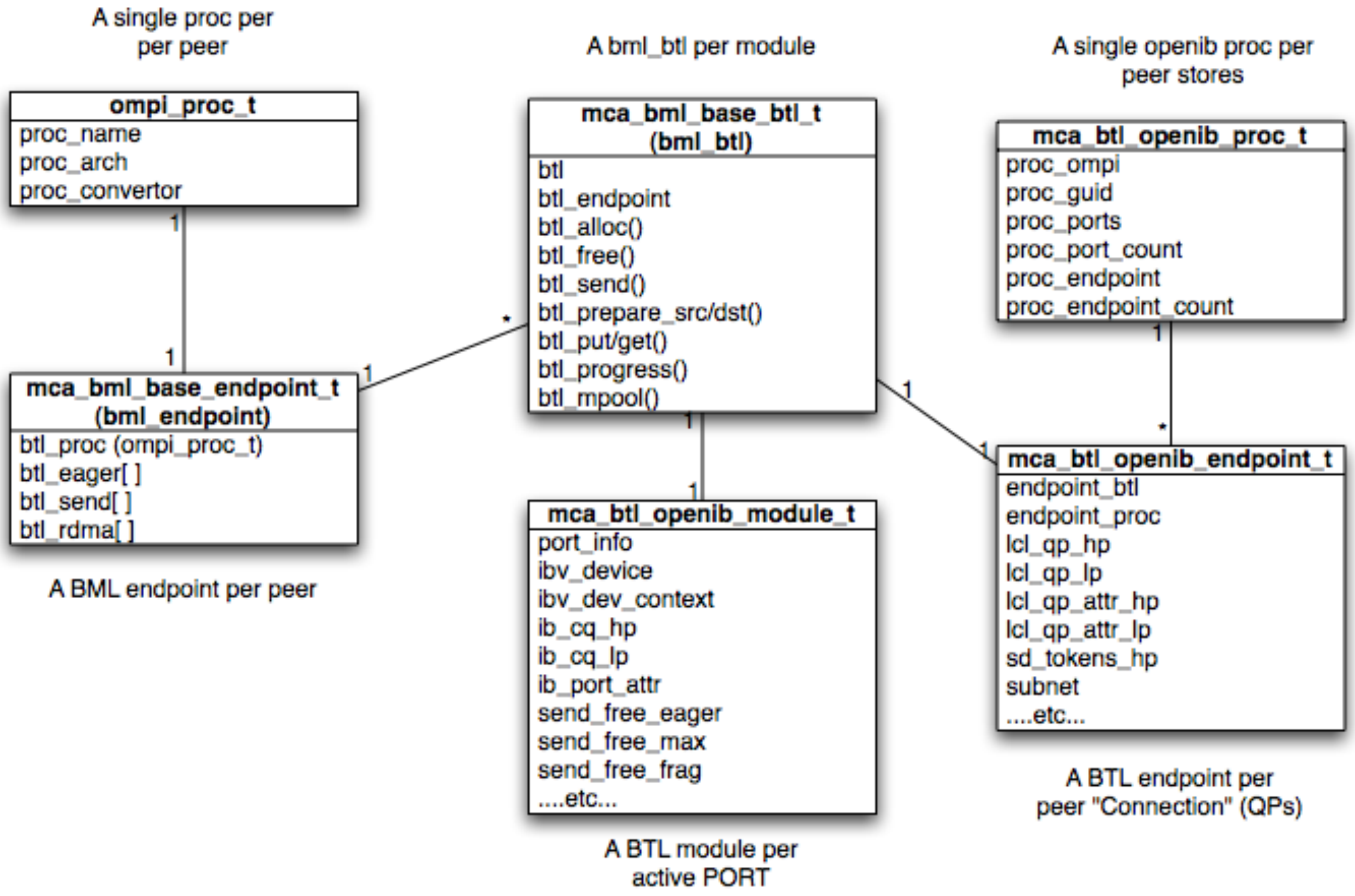
Startup: Peer Reachability

- 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
- 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

Startup: OpenIB Discovery

- Prior to add procs the modex distributes all the peers addressing information to every process
- BTL will query peer resources (Number of ports/lids from cached modex 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
- 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

- MPI call
  - Does any parameter validation (if enabled)
  - Calls the PML interface
- PML interface includes blocking, non-blocking, and persistent interfaces
  - Essentially MPI call, but without parameter validation
  - Don't reduce to the base case (persistent, non-blocking) for performance reasons
- We'll start with a blocking send call, entering OB1.

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

- Find a BTL to use
  - BML endpoint cached on ompi_proc_t structure
  - BML endpoint contains list of available BTLs to that peer
  - Select next available (round-robin)
- BML_BTL structure returned; interface to BTL is through thin shim layer in BML.
- Small messages are scheduled via mca_pml_ob1_send_request_start_copy

Send: Eager (short)

- 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)

Send: Eager (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

Send: BTL

- 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
- BTL sends are un-ordered but reliable

Receive: Posting

- MPI_Recv 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
- 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 opal_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
- 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
- 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

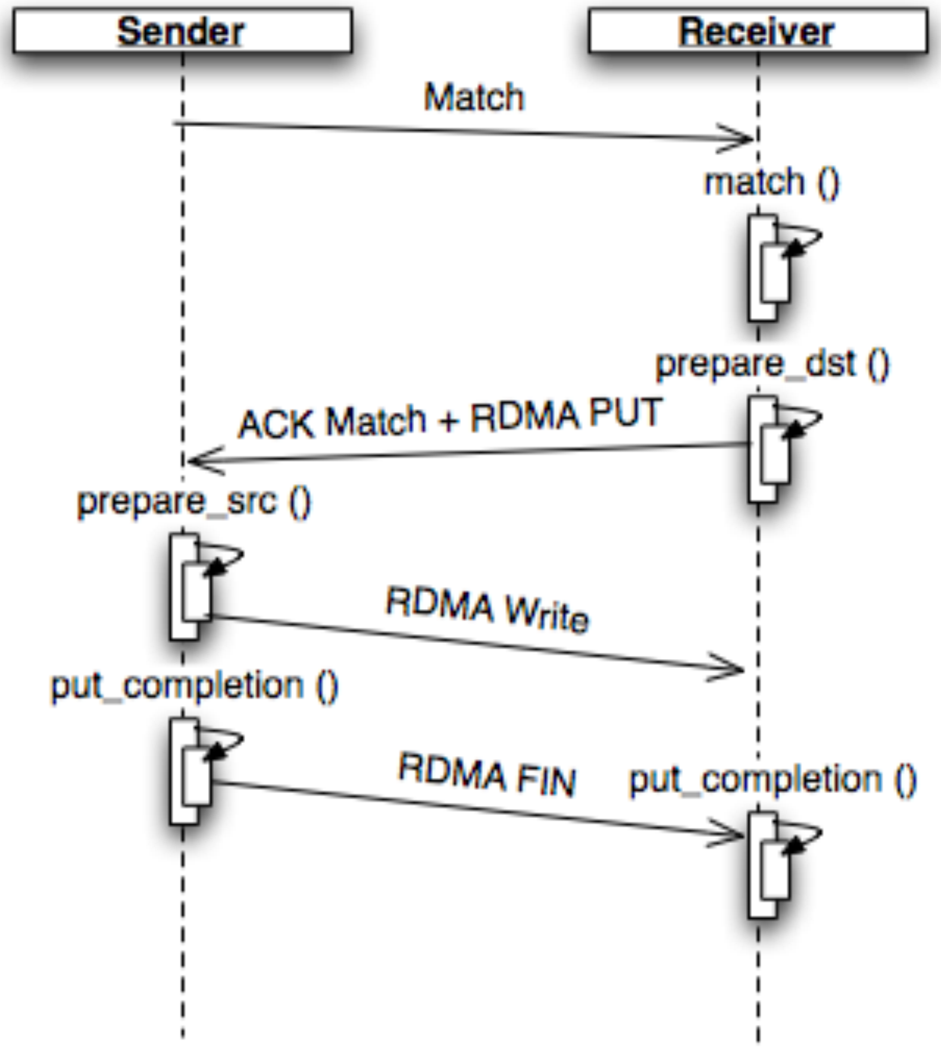
Receive: Unpack

- 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

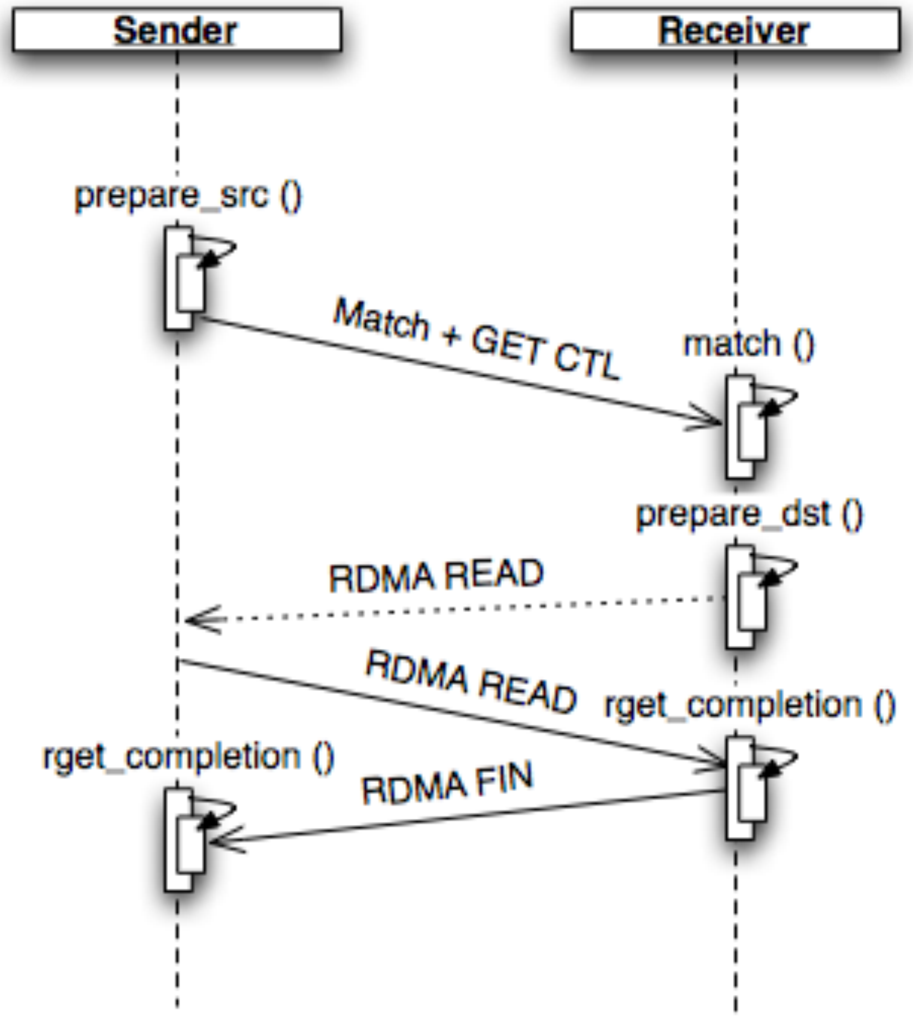
Long Messages

- Unexpected messages make long transfer tricky
- Generally use a RTS/CTS protocol
- Many options for the payload transfer:
  - CTS can actually be an RDMA get
  - CTS can cause sender to do one or more RDMA put
  - CTS can cause the sender to do a number of active message sends
- Choice depends on many factors:
  - Capabilities of BTL (no RDMA get in TCP)
  - Re-use of buffer (pinning can be expensive)

Long: RDMA Put (pinned)

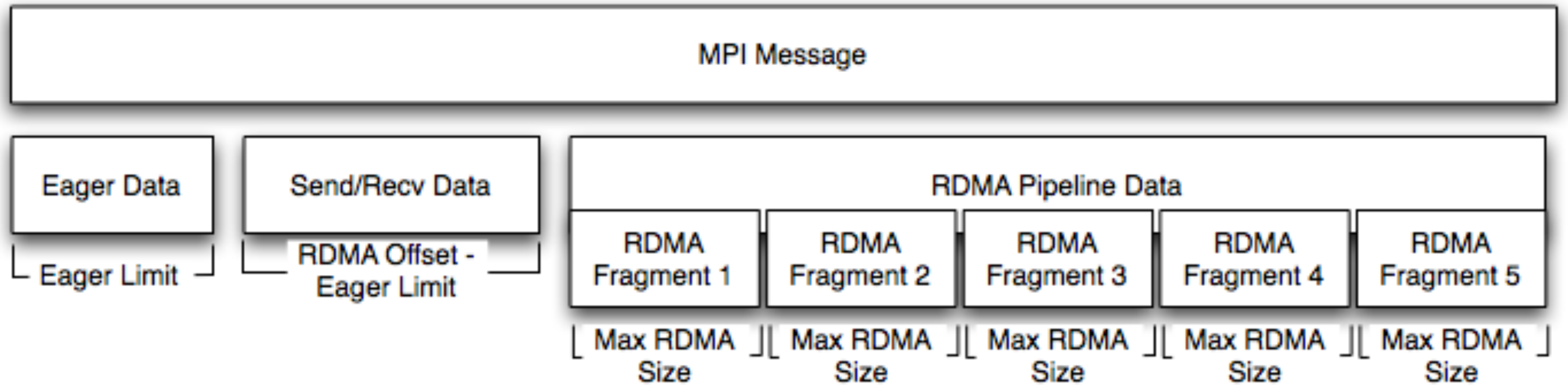


Long: RDMA Get (pinned)



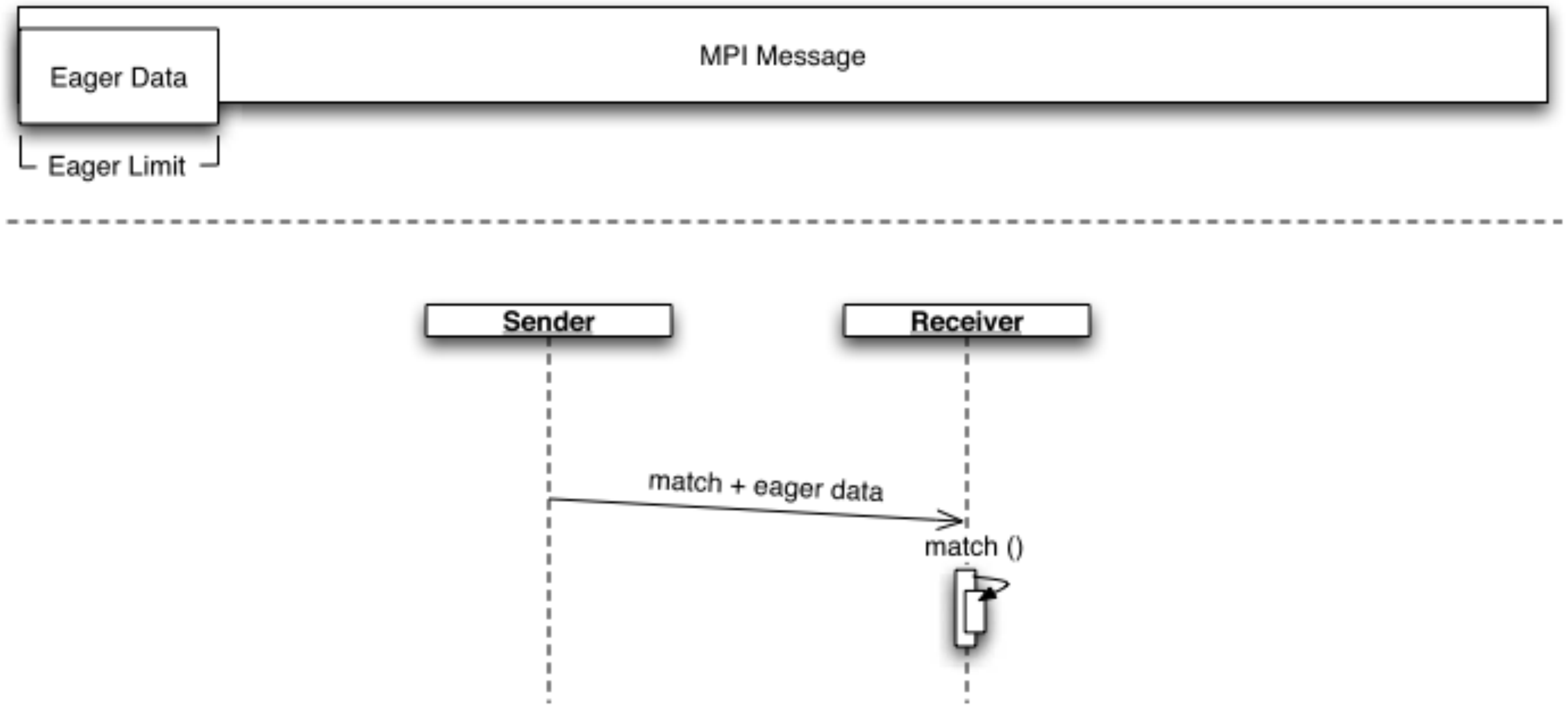
Long: Pipeline Protocol

- 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)



Long: Pipeline Start

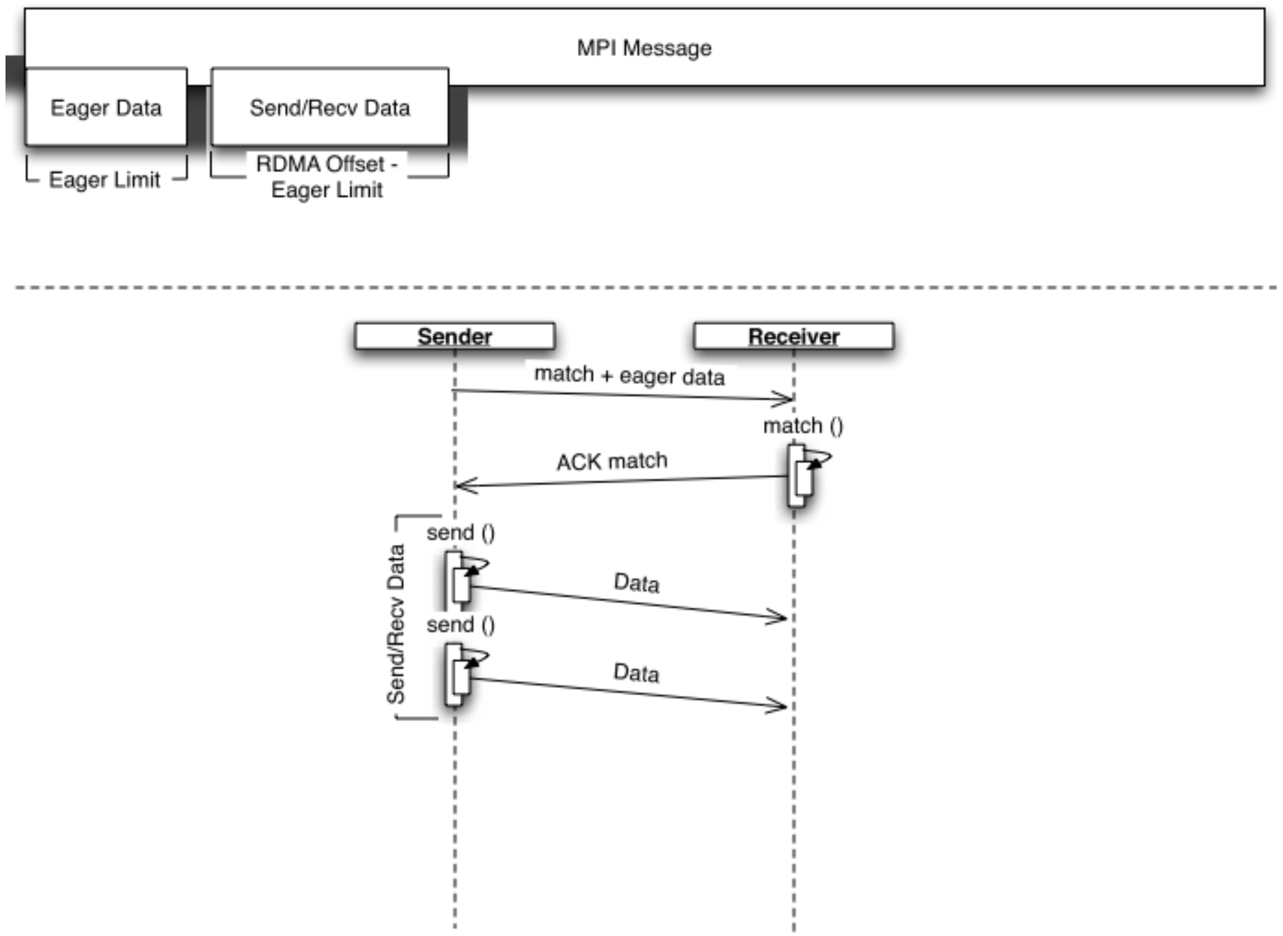
- Start just like a short eager message



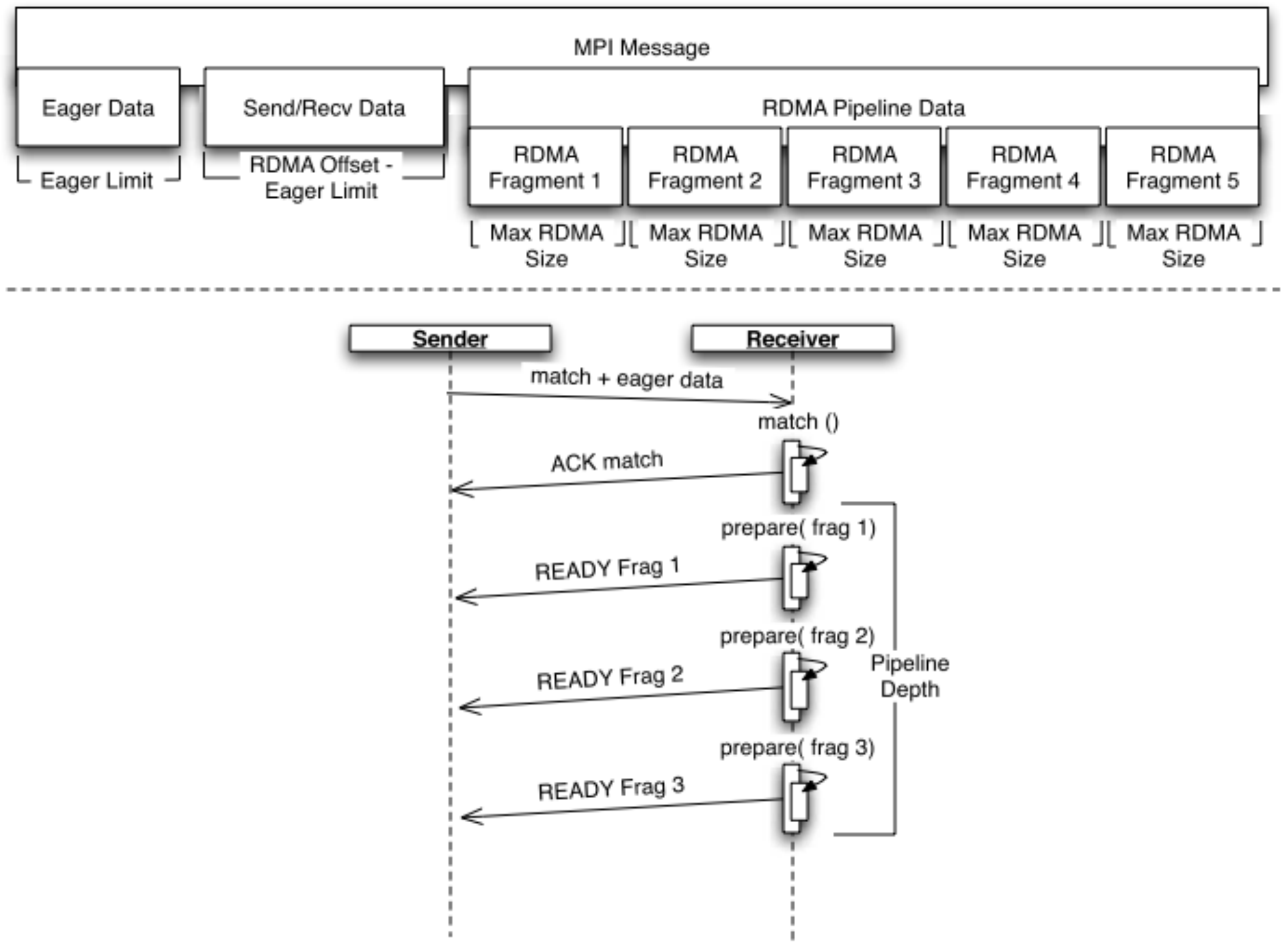
Long: Pipeline Match

- 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

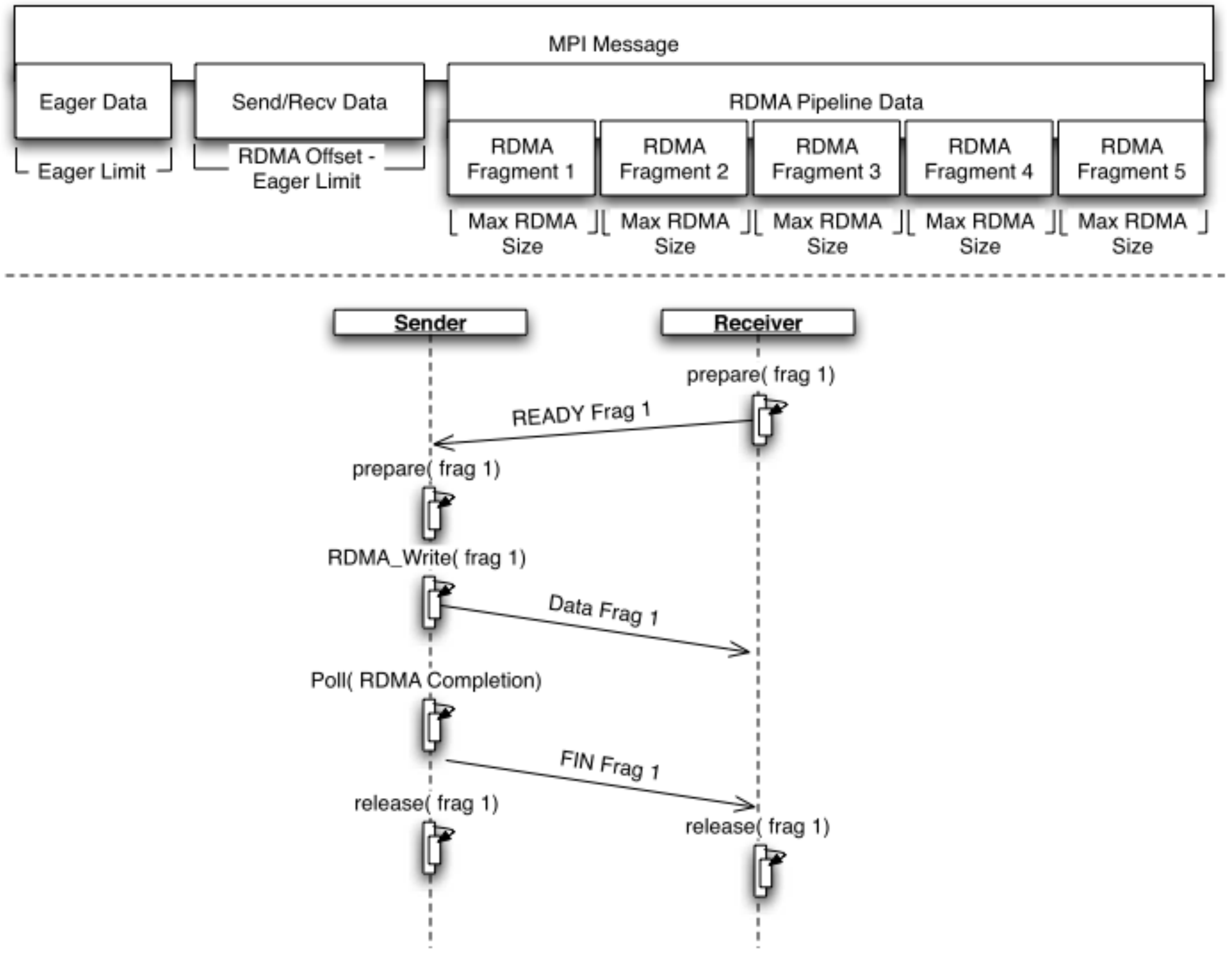
Long: Pipeline “Priming” Send



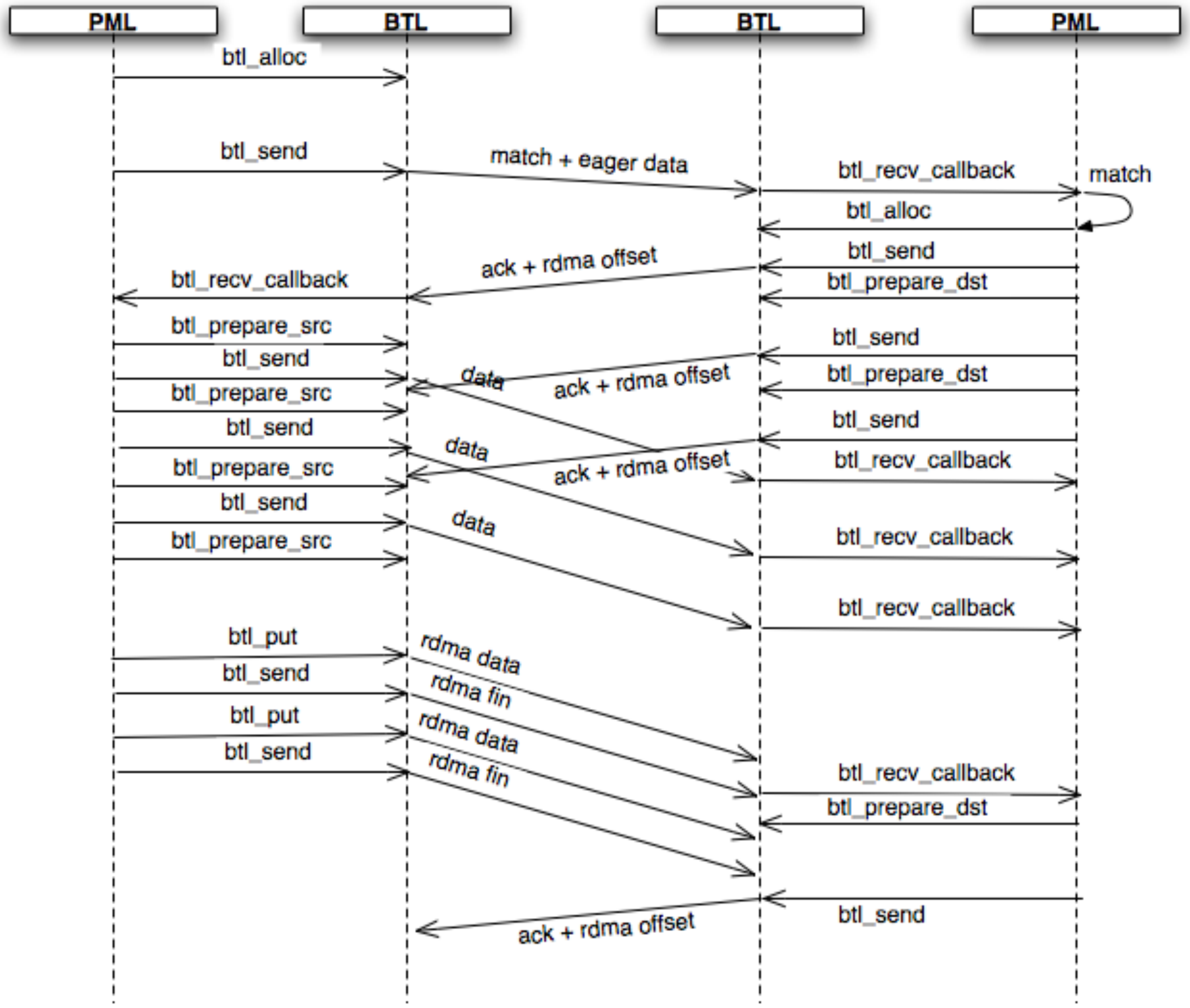
Long: Pipeline RDMA Schedule



Long: Pipelined RDMA transfer



Long: Pipelined (all of it…)



Outline…

- Introduction
- MPI Communication models
- Open MPI Overview
  - Foundational Components
  - Communication in Open MPI
  - Communication deep-dive
- Future plans
  - BTL relocation
  - Fault tolerance
  - Threads

BTL Migration

- Desire to move BTLs from OMPI to OPAL
  - Increase ability to reuse BTLs
  - Requires moving mpool and rcache as well
  - Possibly need to move BML?
- Challenges
  - Modex needs to be replaced
    - Keep scalability for large job launch
    - Provide ability to bootstrap through launch
  - How to handle end endpoint caching?
  - Rational way to handle callback indexing?
- Number of institutions looking at this

Outline…

- Introduction
- MPI Communication models
- Open MPI Overview
  - Foundational Components
  - Communication in Open MPI
  - Communication deep-dive
- Future plans
  - BTL relocation
  - Fault tolerance
  - Threads

Fault Tolerance

- Big area of research for HPC
- Areas of research in Open MPI
  - Handle BTL failure in multi-path situations (BFO) (unmaintained)
  - Checkpoint / restart (System level & app assisted)
  - Run-through failure (U. Tennessee)
- For the most part, BTLs not impacted by FT research
  - Need to return errors properly on failure
  - Careful what you print!
  - No aborting!
  - May need timeouts if network doesn't return failures

Outline…

- Introduction
- MPI Communication models
- Open MPI Overview
    - Foundational Components
    - Communication in Open MPI
    - Communication deep-dive
- Future plans
    - BTL relocation
    - Fault tolerance
    - Threads

Thread Status

- MPI_THREAD_MULTIPLE: getting better (OpenIB problems)
- Asynchronous progress: Let’s talk on Friday ☺
- Let’s walk through some of the thread issues

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

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 Thread model

- BTLs provide asynchronous callbacks
  - PML, OSC, etc. respond to asynchronous events
  - Theoretically, no difference between callbacks from progress thread and multiple MPI threads
- Problems:
  - Latency, Gap, and Message Rate ☺
    - Progress thread blocking increases latency
    - Progress thread polling drastically reduces effectiveness
  - MPICH model would loosely translate to having two BTL channels:
    - One which provides no asynch progress (for short messages)
    - One which provides asynch progress (long Pt-2-pt, one-sided)

Sandia National Laboratories