

Gemini description, MPI

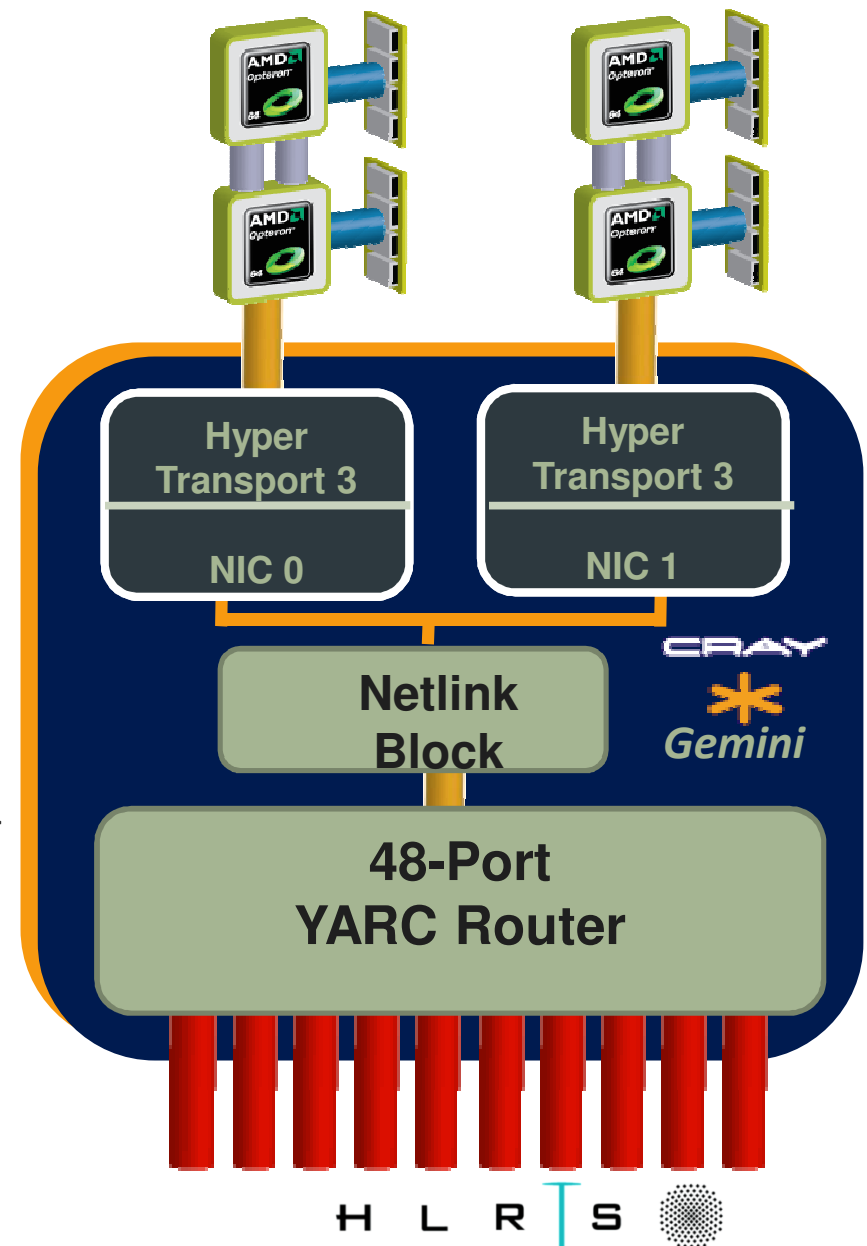
Jason Beech-Brandt
jason@cray.com



Cray Gemini ASIC

- Supports 2 Nodes per ASIC
- 3D Torus network
 - XT5/XT6 systems field upgradable
- Scales to over 100,000 network endpoints
 - Link Level Reliability and Adaptive Routing
 - Advanced Resiliency Features
- Advanced features
 - MPI – millions of messages / second
 - One-sided MPI
 - UPC, Coarray FORTRAN, Shmem, Global Arrays
 - Atomic memory operations

LO
Processor

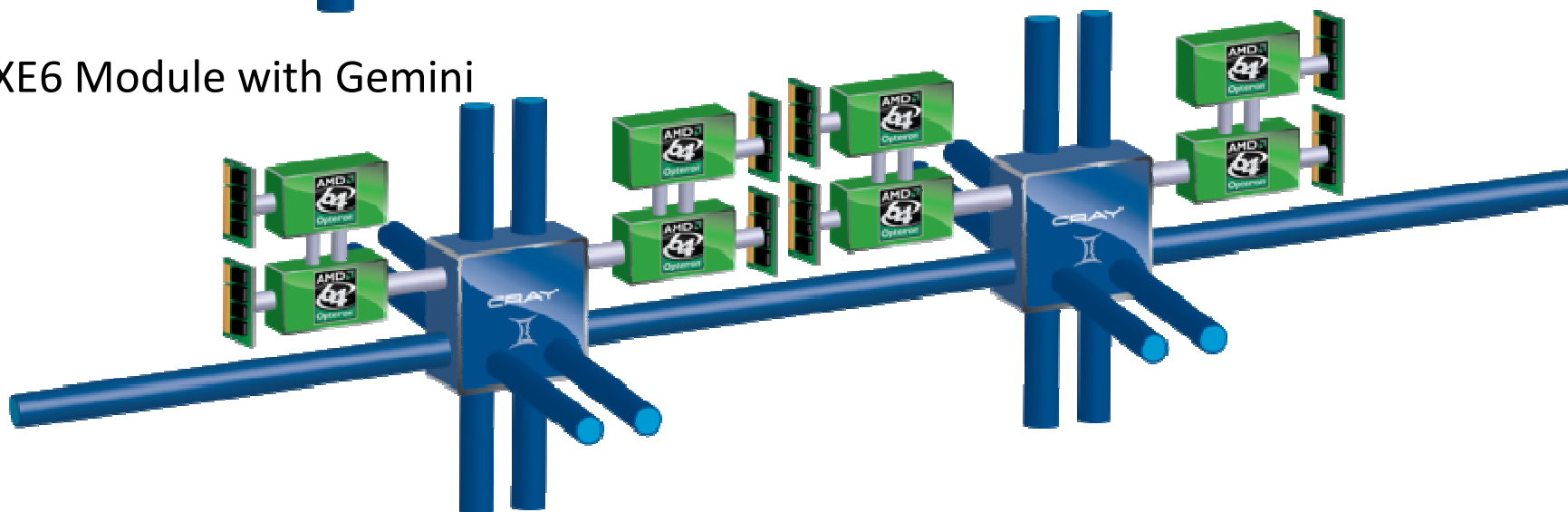


Gemini vs SeaStar – Topology

XT6 Module with SeaStar



XE6 Module with Gemini



Gemini MPI Features



- FMA (Fast Memory Access)
 - Mechanism for most MPI transfers
 - Supports tens of millions of MPI requests per second
- BTE (Block Transfer Engine)
 - DMA offload engine , supports *asynchronous* block transfers between local and remote memory, in either direction
- Gemini provides low-overhead OS-bypass features for short transfers
 - MPI latency around 1.4us in current release
 - NIC provides for many millions of MPI messages per second 20 times better than Seastar
- Much improved injection bandwidth – 6 GB/s user data injection with HT3 link

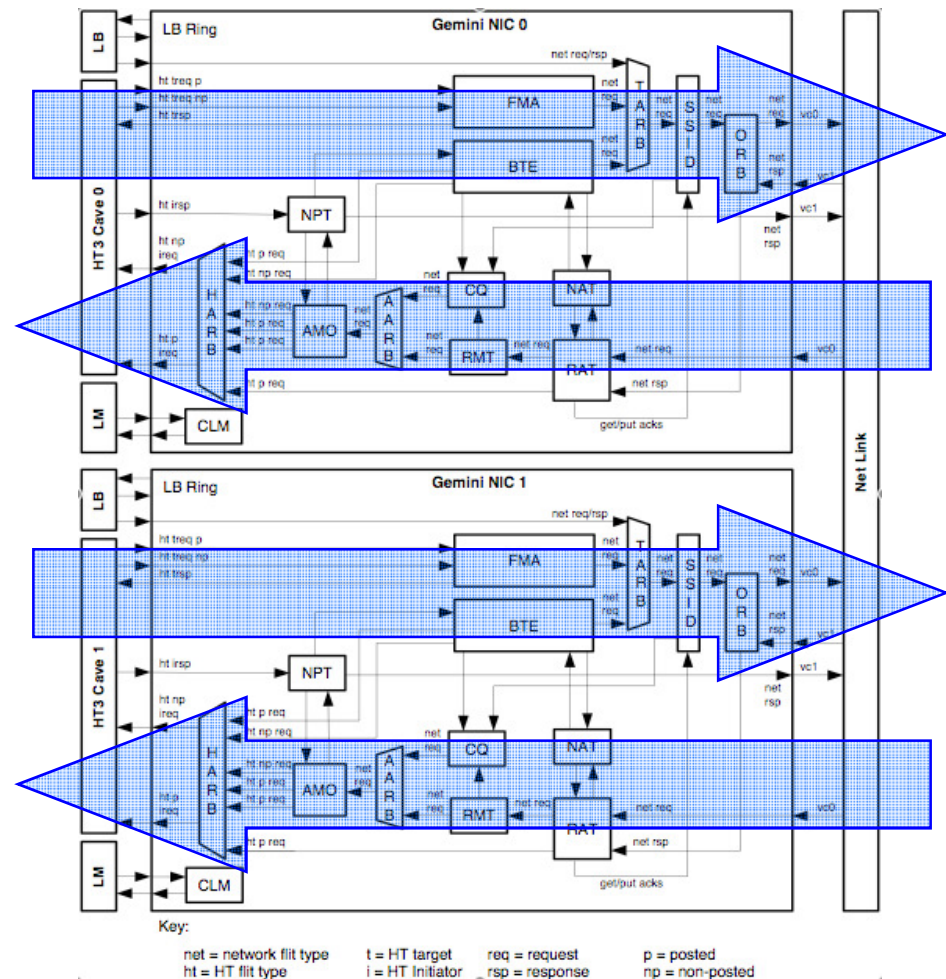
Gemini Advanced Features



- Globally addressable memory provides efficient support for UPC, FORTRAN 2008 with Coarrays, Shmem and Global Arrays
 - Much improved one-sided communication mechanism with hardware support
 - Cray compiler targets this capability directly
- Pipelined global loads and stores
 - Allows for fast irregular communication patterns
- Atomic memory operations
 - AMOs provide a faster synchronization method for barriers
 - Provides fast synchronization needed for one-sided communication models

Gemini NIC Design

- Hardware pipeline maximizes issue rate
- HyperTransport 3 host interface
- **Fast memory access (FMA)**
 - Low latency and high issue rate for small transfers
- **Block transfer engine (BTE)**
 - Gemini does the transfer on behalf of the rank/image
- Hardware translation of user ranks and addresses
- **Global AMOs**
- **Network bandwidth dynamically shared between NICs**



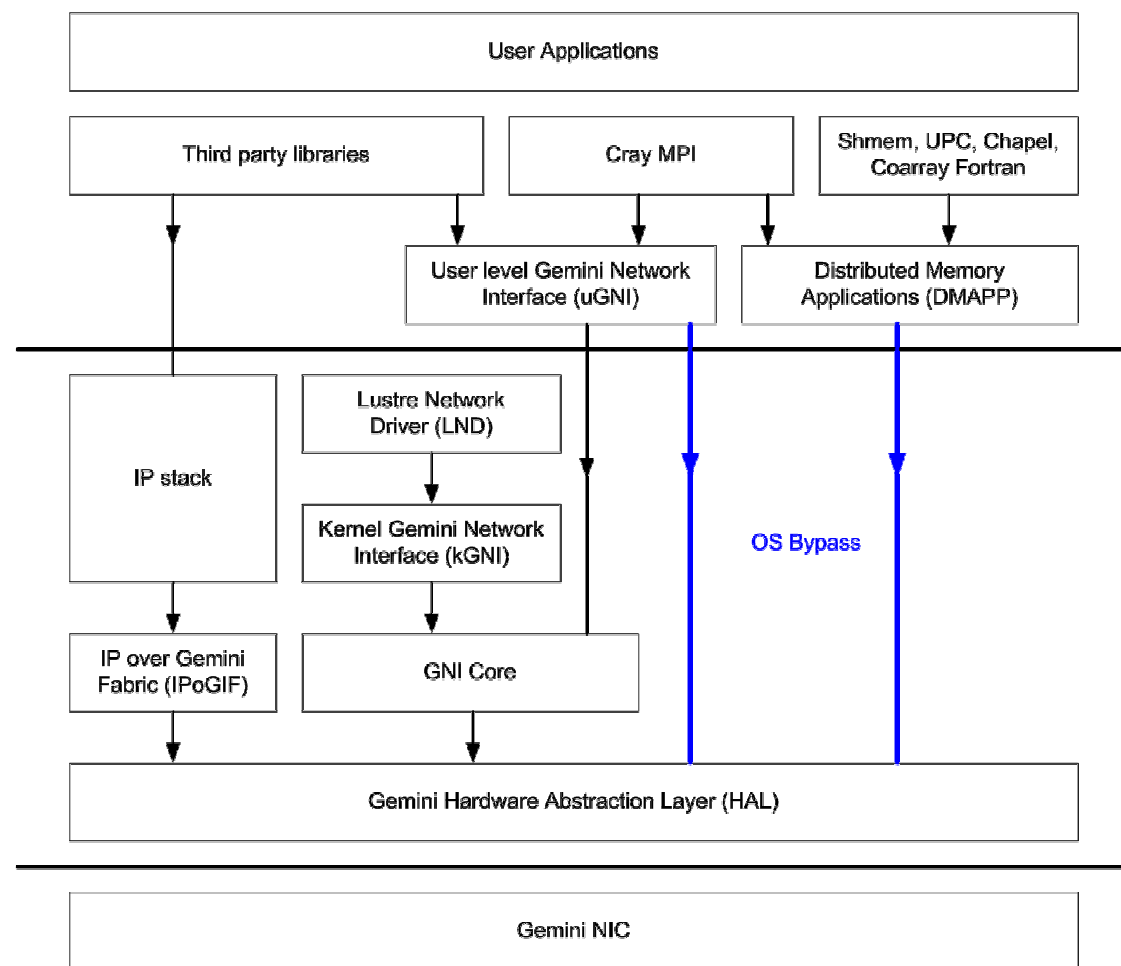
BTE vs FMA

- FMA PROS
 - Lowest latency (~1.2 usecc)
 - All data has been read by the time dmapp return
 - More than one transfer active at the same time
- FMA CONS
 - CPU involved in the transfer
 - Performance can vary depending on die used
- BTE PROS
 - Transfer done by gemini, asynchronous with CPU
 - Transfers are queued if Gemini is busy
 - Seems to get better P2P bandwidth in more cases
- BTE CONS
 - Higher latency : ~2 usec if queue is empty
 - Transfers are queued if Gemini is busy
 - Only one BTE active at a time

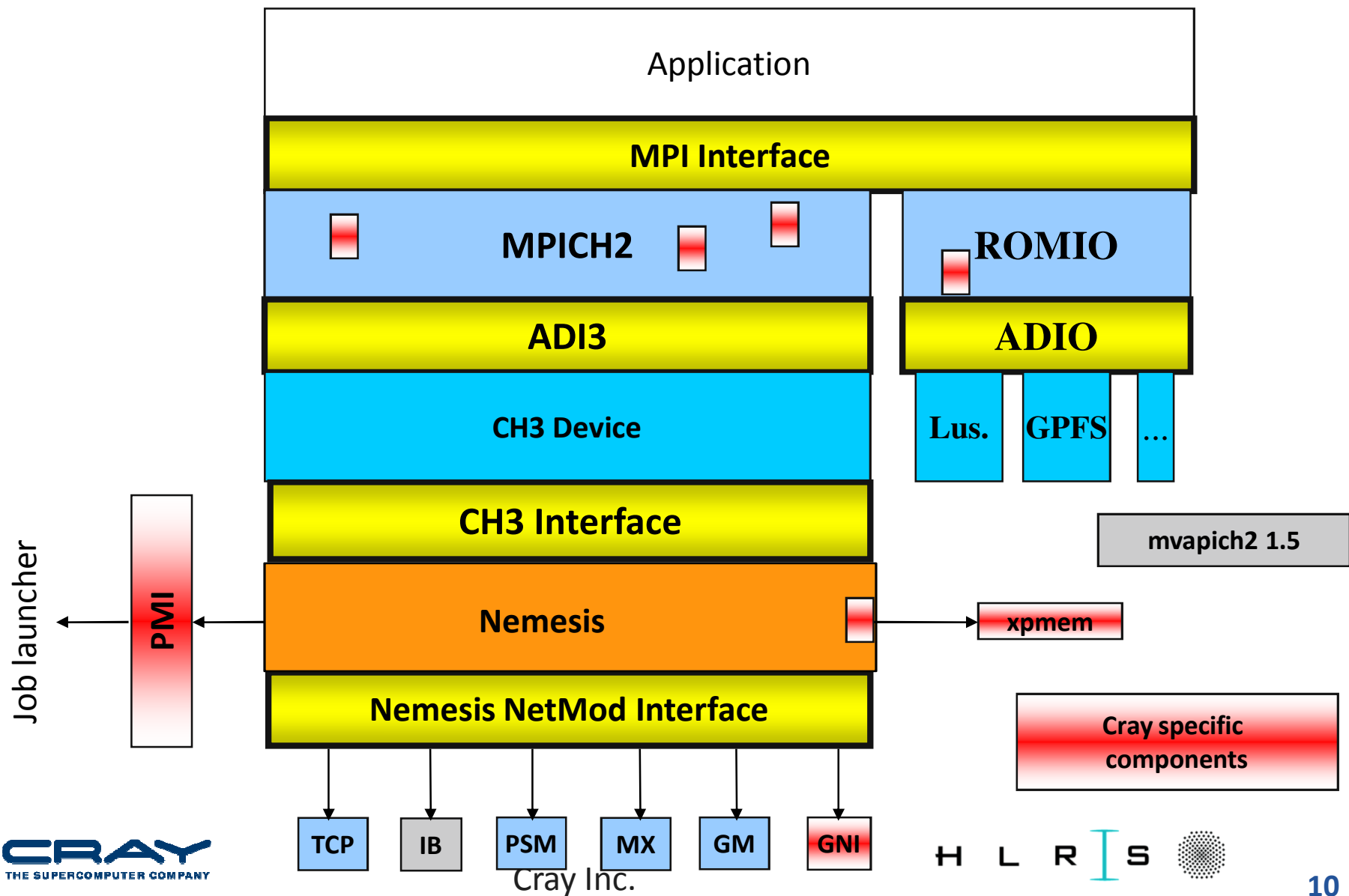
Gemini Reliability Features

- Error protection and detection
 - Link level error detection with hardware retry
 - Packet CRCs provide end-to-end error protection
 - NIC/Router memories protected by ECC
- Diagnosis, error handling and reconfigurability
 - Auto-degrading network channels tolerate single-lane hard failures
 - Adaptive routing reduces packet loss in the event of hard link failures
 - Can warm-swap blades and reconfigure network without rebooting
 - HT link failures will not backpressure/deadlock network
- Capabilities
 - System can ride through network failures and reconfigure network while live
 - Cray MPI implementation is resilient to network failures; provides end-to-end reliable delivery

Gemini Software Stack

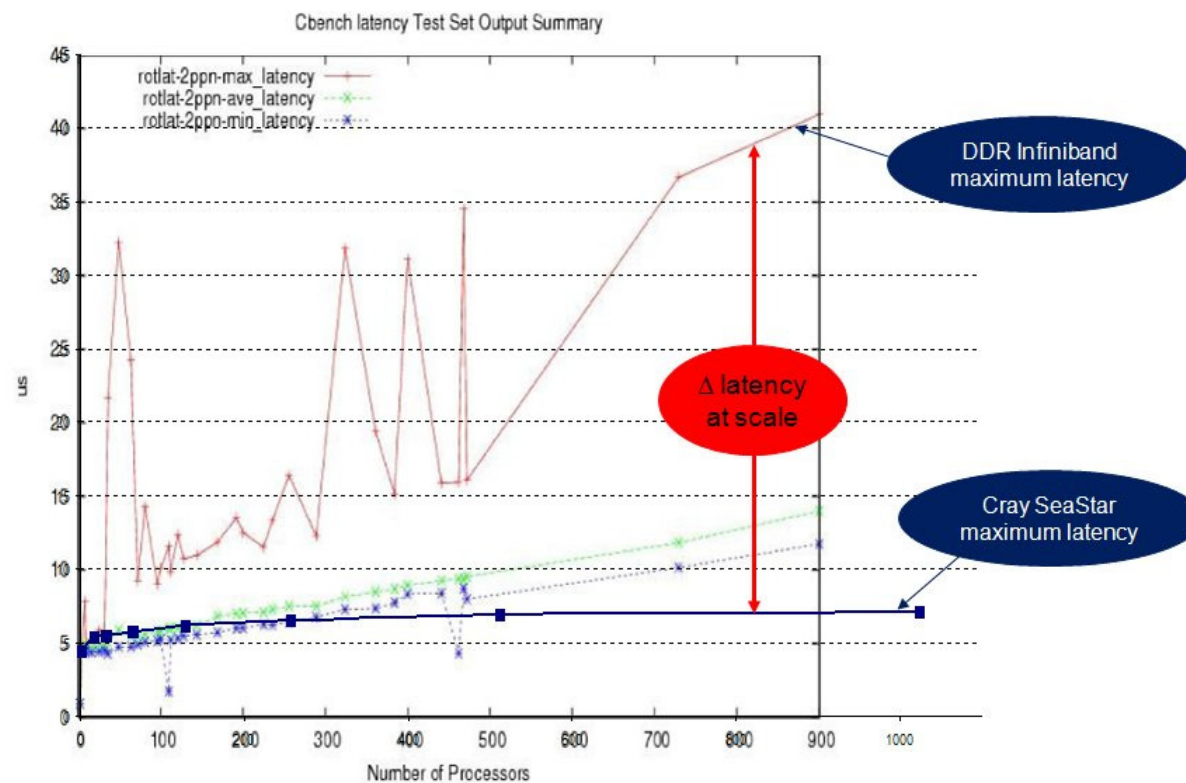


MPICH2 on CRAY XE (aka Gem/Ari)



Latency comparison at scale

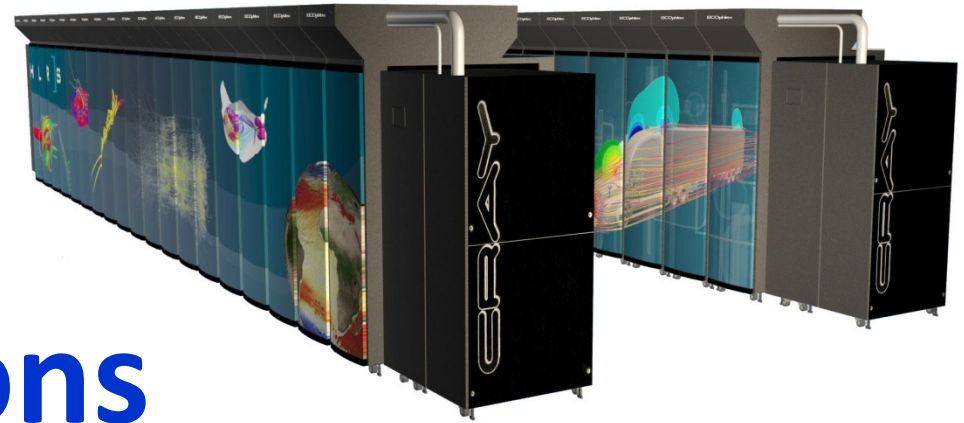
- Low Infiniband latencies seen in micro-benchmarks are not sustained across a large system. Data from LLNL and Sandia shows Infiniband latency and performance variability increasing with system size



Source: Presentation by Matt Leininger & Mark Seager, OpenFabrics Developers Workshop, Sonoma, CA, April 30th, 2007

XT systems shows constant latency at scale.

MPI Communications on the Cray XE6



Cray MPI basics

- Our MPI is based on MPICH-2 from Argonne National Laboratory
- Implements the MPI-2.2 standard except for
 - Cancelling of MPI send requests
 - Dynamic process management
 - external32 data representation
 - MPI_LONG_DOUBLE datatype is not supported
- Currently MPI is distributed in the “xt-mpt” module
 - Contains both MPI and SHMEM libraries
 - For dynamic linking users must swap to xt-mpich2 or xt-shmem
 - As of CLE3.1UP02, xt-mpich2 will be the default MPI module

How MPICH2 Uses GNI – Key Concepts

- Due to lack of messaging hardware on Gem/Ari, a connection oriented approach is used (GNI SMSG mailboxes)
- The relatively limited memory registration resources have a major impact on the MPICH2 GNI Netmod design. Using large pages generally helps alleviate problems associated with these limited memory registration resources.
- All network transactions are tracked at some level. No fire-and-forget. Helps with dealing with transient network errors.

How MPICH2 Uses GNI – SMSG Mailboxes

- Uses put-with-notification hardware on Gemini. This allows implementing a one-way-through network messaging scheme.
- Can recover from transient network errors
- Flow control
- MPICH2 and GNILND (Lustre, DVS, etc.) share same mailbox code
- By default, connections (mailboxes) are established dynamically. Note mailboxes are actually allocated in blocks due to limited memory registration resources on NIC.
- Process private and shared SMSG mailboxes available. Current MPICH2 only uses private ones.

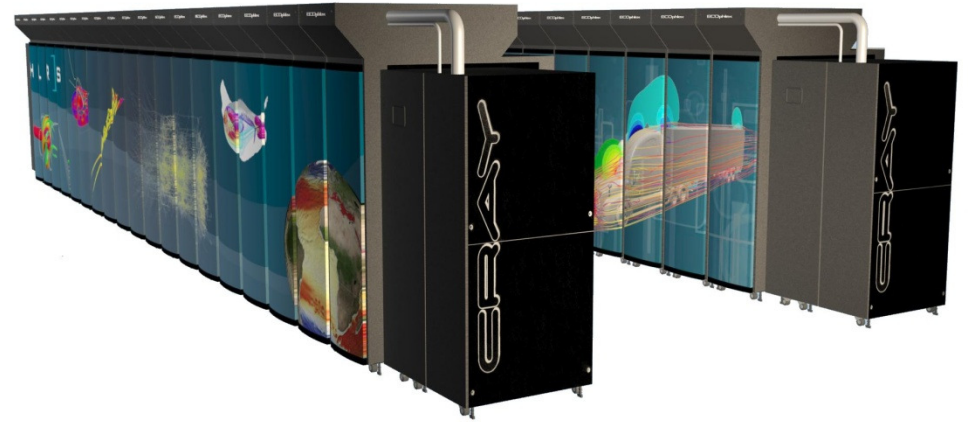
MPICH2 GNI Netmod Message Protocols

- Eager Protocol
 - For a message that can fit in a GNI SMSG mailbox (E0)
 - For a message that can't fit into a mailbox but is less than MPICH_GNI_MAX_EAGER_MSG_SIZE in length (E1)
- Rendezvous protocol (LMT)
 - RDMA Get protocol – up to 512 KB size messages by default
 - RDMA Put protocol – above 512 KB

Maximum message size for E0 varies with Job Size

- Protocol for messages that can fit into a GNI SMSG mailbox
- The default varies with job size, although this can be tuned by the user to some extent

ranks in job	maximum bytes of user data
≤ 1024	984
$>1024 \ \&\& \leq 16384$	472
> 16384	216



MPI

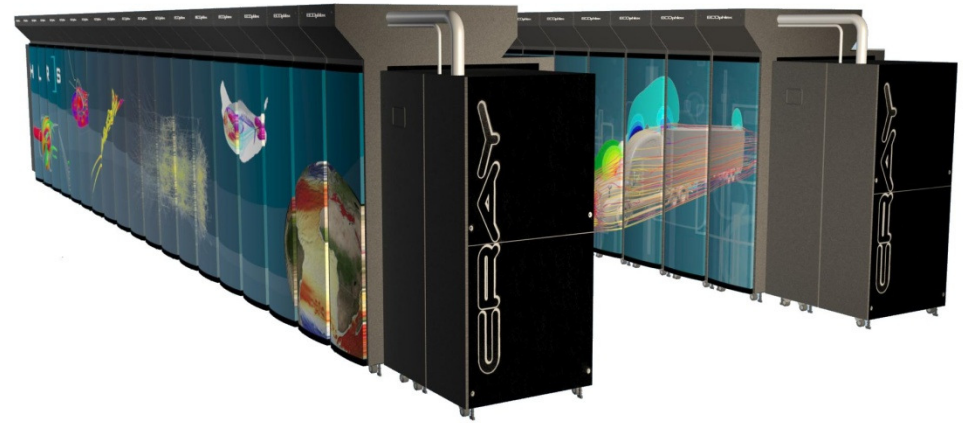
Environment Variables for
Inter-node Point-to-Point Messaging

MPICH2 environment variables

- Several environment variables are available to control MPI features (man mpi or intro_mpi)
- MPICH_ENV_DISPLAY
 - If set, causes rank 0 to display all MPICH environment variables
- MPICH_CPUMASK_DISPLAY
 - If set, causes each MPI rank in the job to display its CPU affinity bitmask
- MPICH_MAX_THREAD_SAFETY
 - Specifies thread-safety level
 - MPI_THREAD_MULTIPLE requires a specific library:
link to -lmpich_threadm

MPICH2 environment variables

- MPICH_ABORT_ON_ERROR
 - If set, causes MPICH-2 to abort and produce a core dump when MPICH-2 detects an internal error. Note the shell coredumpsize must be set appropriately to enable coredumps.
- MPICH_VERSION_DISPLAY
 - If set, causes MPICH2 to display the CRAY MPICH2 version number as well as build date information.
- MPICH_MEMCPY_MEM_CHECK
 - If set, enables a check of the memcpy() source and destination areas. If they overlap, the application asserts with an error message. If this error is found, correct it either by changing the memory ranges or possibly by using MPI_IN_PLACE.



MPI

Gemini specific env. variables

MPICH_GNI_MAX_VSHORT_MSG_SIZE

- Can be used to control the maximum size message that can go through the private SMSG mailbox protocol (E0 *eager* path).
- Default varies with job size.
- Maximum size is 1024 bytes. Minimum is 80 bytes.
- If you are trying to demonstrate an MPI_Alltoall at very high count, with smallest possible memory usage, may be good to set this as low as possible.
- If you know your app has a scalable communication pattern, and the performance drops at one of the edges shown on the table (page 21), you may want to set this environment variable.
- Pre-posting receives for this protocol avoids a potential extra memcpy at the receiver.

MPICH_GNI_MAX_EAGER_MSG_SIZE

- Default is 8192 bytes
- Maximum size message that go through the *eager* (E1) protocol
- May help for applications sending medium size messages
- Maximum allowable setting is 131072 bytes
- Pre-posting receives can avoid potential double memcpy at the receiver.
- Note that a 40-byte Nemesis header is included in account for the message size.

MPICH_GNI_RDMA_THRESHOLD

- Default is now 1024 bytes
- Controls the threshold at which the GNI netmod switches from using FMA for RDMA read/write operations to using the BTE.
- Since BTE is managed in the kernel, BTE initiated RDMA requests can progress even if the applications isn't in MPI.
- But using the BTE may lead to more interrupts being generated

MPICH_GNI_NDREG_LAZYMEM

- Default is enabled. To disable
export MPICH_GNI_NDREG_LAZYMEM=disabled
- Controls whether or not to use a lazy memory deregistration policy inside UDREG. Memory registration is expensive so this is usually a good idea.
- Only important for those applications using the LMT (large message transfer) path, i.e. messages greater than MPICH_GNI_MAX_EAGER_MSG_SIZE.
- Disabling may be a workaround for some UDREG issues
- However, disabling results in a significant drop in measured bandwidth for large transfers ~40-50 %.

MPICH_GNI_DMAPP_INTEROP

- Only relevant for mixed MPI/SHMEM/UPC/CAF codes
- For Danube systems, want to leave enabled so MPICH2 and DMAPP can share the same memory registration cache, reducing pressure on memory registration resources on the NIC
- May have to disable for SHMEM codes that call *shmem_init* after *MPI_Init*.
- May want to disable if trying to add SHMEM/CAF to an MPI code and notice a big performance drop.
- Syntax:

```
export MPICH_GNI_DMAPP_INTEROP=disabled
```

MPICH_GNI_NUM_BUFS

- Default is 64
- Controls the number of 32KB DMA buffers available for each rank to use in the GET-based Eager protocol (E1).
- May help to modestly increase. But other resources constrain the usability of a large number of buffers, so don't go berserk with this one.
- Syntax:

```
export MPICH_GNI_NUM_BUFS=X
```

MPICH_GNI_MBOX_PLACEMENT

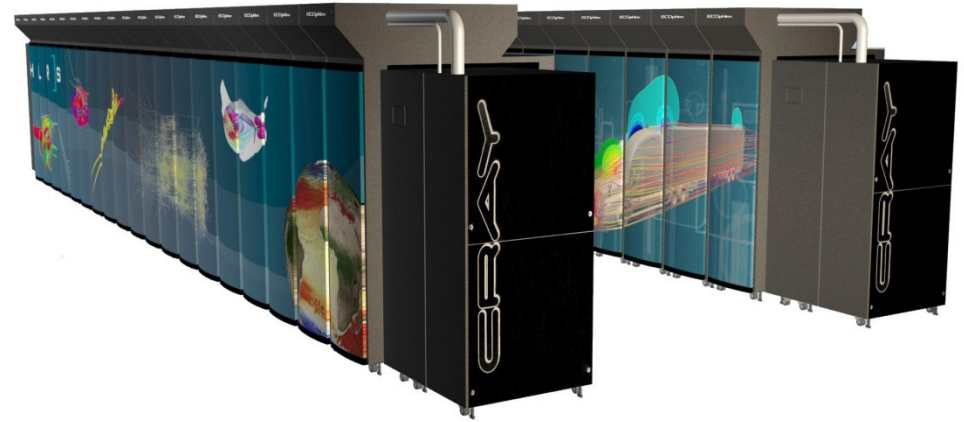
- Provides a means for controlling which memories on a node are used for some SMSG mailboxes (private).
- Default is to place the mailboxes on the memory where the process is running when the memory for the mailboxes is faulted in.
- For optimal MPI message rates, better to place mailboxes on memory of die0 (where Gemini is attached).
- Only applies to first 4096 mailboxes of each rank on the node.
- Feature only available in very recent CLE versions (not all UP01s?) and very most recent MPT build (actually none yet due to build issues as of today).
- Syntax for enabling placement of mailboxes near the Gemini:
`export MPICH_GNI_MBOX_PLACEMENT=nic`

MPICH_GNI_DYNAMIC_CONN

- Enabled by default
- Normally want to leave enabled so mailbox resources (memory, NIC resources) are allocated only when the application needs them
- If application does all-to-all or many-to-one/few, may as well disable dynamic connections. This will result in significant startup/shutdown costs though.
- Recent bugs have been worked around by disabling dynamic connections.
- Syntax for disabling:
export MPICH_GNI_DYNAMIC_CONN=disabled

MPICH_GNI_FORK_MODE

- This environment variable controls the behaviour of registered memory segments when a process invokes a fork or related system call. There are three options:
 - NOCOPY
 - FULLCOPY
 - PARTCOPY
- Be aware this exists, but most apps don't do this.



MPI

Environment Variables for Collective communication

MPICH_COLL_OPT_OFF

- Collectives use, by default, architecture specific algorithms for some MPI collective operations.
- Generally a good thing!
- However they can be disabled for debugging purposes, or for bitwise reproducibility requirements
- Setting this variable to 1 disables all collective optimizations
- Setting to a comma-delimited list of collective operations will disable these selectively
 - For example `MPICH_COLL_OPT_OFF=mpi_allgather`.
 - The following collective names are recognized
`MPI_Allgather`, `MPI_Allgatherv`, and `MPI_Alltoall`.

MPICH_COLL_SYNC

- If set, a Barrier is performed at the beginning of each specified MPI collective function. This forces all processes participating in that collective to sync up before the collective can begin.
- To enable this feature for all MPI collectives, set the value to 1.
- To enable this feature for selected MPI collectives, set the value to a comma-separated list of the desired collective names. Names are not case-sensitive. Any unrecognizable name is flagged with a warning message and ignored. The following collective names are recognized: MPI_Allgather, MPI_Allgatherv, MPI_Allreduce, MPI_Alltoall, MPI_Alltoallv, MPI_Alltoallw, MPI_Bcast, MPI_Exscan, MPI_Gather, MPI_Gatherv, MPI_Reduce, MPI_Reduce_scatter, MPI_Scan, MPI_Scatter, and MPI_Scatterv.

MPI_Allgather

- With MPT 5.1 switched to using Seastar-style algorithm where for short transfers/rank: use MPI_Gather/MPI_Bcast rather than ANL algorithm
- Switchover from Cray algorithm to ANL algorithm can be controlled by the MPICH_ALLGATHER_VSHORT_MSG environment variable. By default enabled for transfers/rank of 1024 bytes or less
- The Cray algorithm can be deactivated by setting

```
export MPICH_COLL_OPT_OFF=mpi_allgather      (bash)
setenv MPICH_COLL_OPT_OFF mpi_allgather      (tcsh)
```

MPI_Allgatherv

- With MPT 5.1 switched to using Seastar-style algorithm where for short transfers/rank: use a specialized MPI_Gatherv/MPI_Bcast rather than ANL algorithm
- Switchover from Cray algorithm to ANL algorithm can be controlled by the MPICH_ALLGATHERV_VSHORT_MSG environment variable. By default enabled for transfers/rank of 1024 bytes or less.
- The Cray algorithm can be deactivated by setting

```
export MPICH_COLL_OPT_OFF=mpi_allgatherv      (bash)
setenv MPICH_COLL_OPT_OFF mpi_allgatherv      (tcsh)
```

MPI_Alltoall

- Optimizations added in MPT 5.1
- Switchover from ANL's implementation of Bruck algorithm (IEEE TPDS, Nov. 1997) is controllable via the `MPICH_ALLTOALL_SHORT_MSG` environment variable. Defaults are

ranks in communicator	Limit (in bytes) for using Bruck
≤ 512	2048
$>512 \ \&\& \ \leq 1024$	1024
> 1024	128

- Larger transfers use an optimized pair-wise exchange algorithm
- New algorithm can be disabled by
`export MPICH_COLL_OPT_OFF=mpi_alltoall`

MPI_Allreduce/MPI_Reduce

- The ANL smp-aware MPI_Allreduce/MPI_Reduce algorithms can cause issues with bitwise reproducibility. To address this Cray MPICH2 has two new environment variables starting with MPT 5.1 -
- MPICH_ALLREDUCE_NO_SMP
disables use of smp-aware MPI_Allreduce
- MPICH_REDUCE_NO_SMP
disables use of smp-aware MPI_Reduce

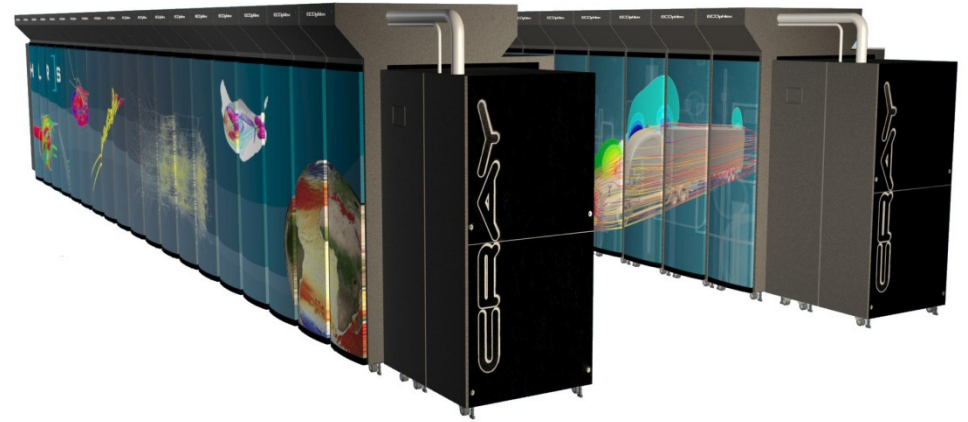
MPI_Bcast

- Starting with MPT 5.1, all ANL algorithms except for binomial tree are disabled since the others perform poorly for communicators with 512 or more ranks
- To disable this tree algorithm-only behavior, set the MPICH_BCAST_ONLY_TREE environment variable to 0, i.e.

```
export MPICH_BCAST_ONLY_TREE=0
```

MPICH_SCATTERV_SYNCHRONOUS

- MPI_Scatterv uses asynchronous sends by default
- Setting this variable forces the use of synchronous sends
- Can be beneficial in some cases
 - Large data sizes
 - High process counts



MPI

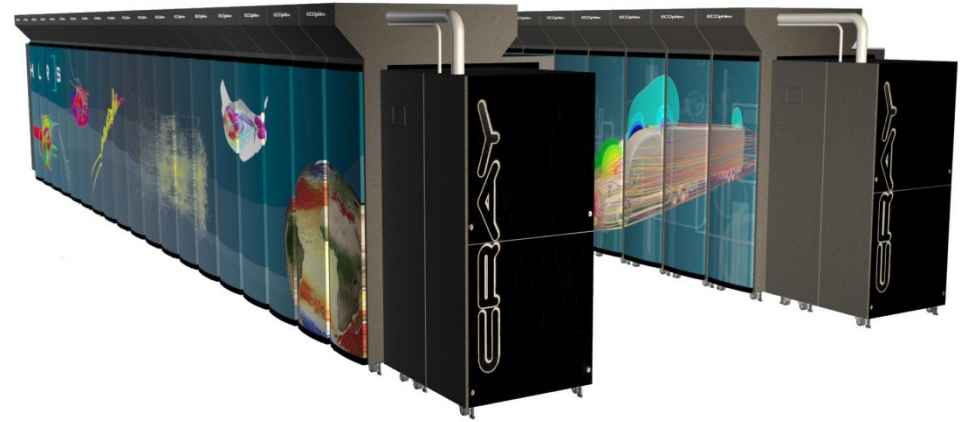
Environment Variables for
Intra-node Point-to-Point Messaging

MPICH_SMP_SINGLE_COPY_SIZE

- Default is 8192 bytes
- Specifies threshold at which the Nemesis shared memory channel switches to a single-copy, XPMEM based protocol for intra-node messages

MPICH_SMP_SINGLE_COPY_OFF

- In MPT 5.1 the default is enabled
- Specifies whether or not to use a XPMEM-based single-copy protocol for intra-node messages of size MPICH_SMP_SINGLE_COPY_SIZE bytes or larger
- May need to set this environment variable if
 - Finding XPMEM is kernel OOPses (check the console on the SMW)
 - Sometimes helps if hitting UDREG problems. XPMEM goes kind of crazy with Linux mmu notifiers and causes lots of UDREG invalidations (at least the way MPICH2 uses XPMEM).



MPI

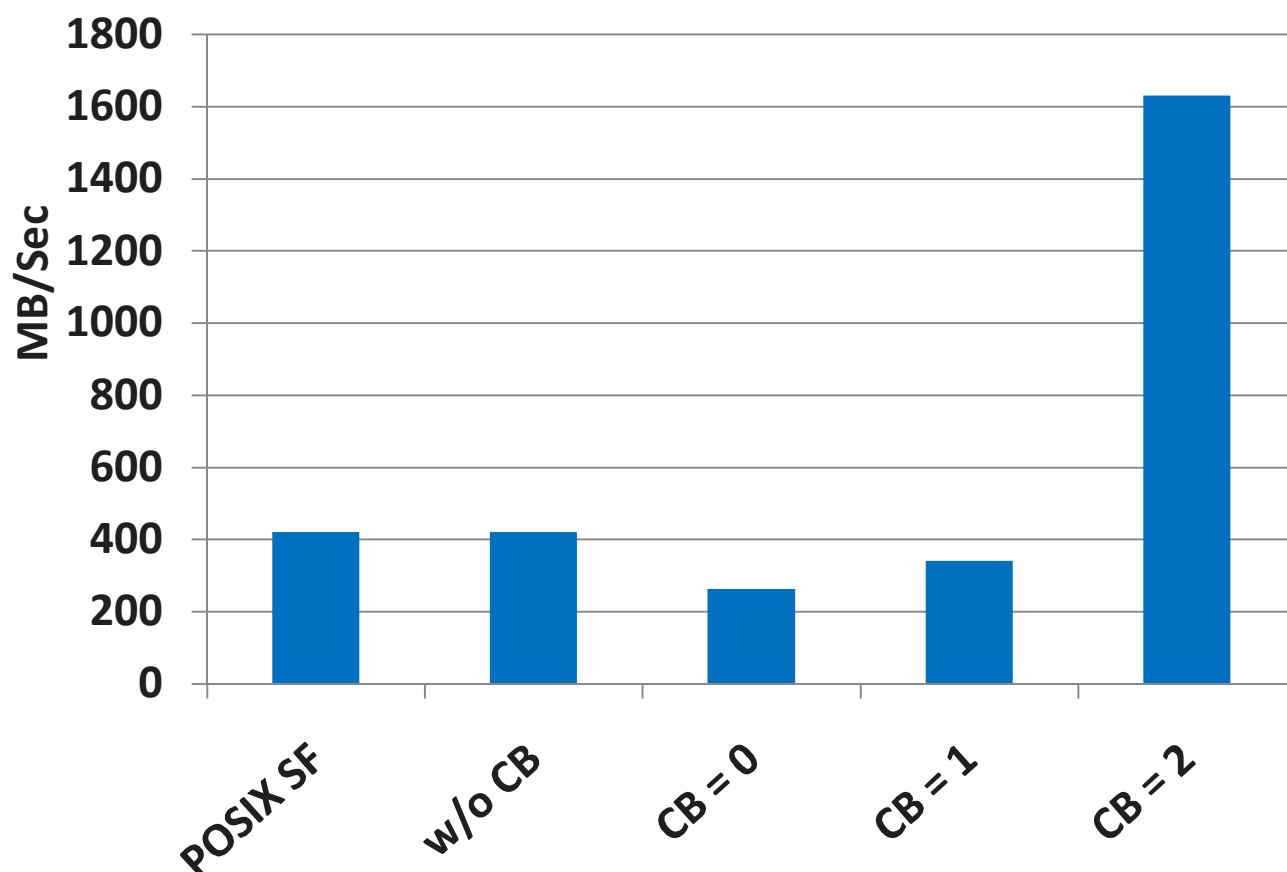
Environment variables for MPI-IO

MPI-IO

- Wildcard matching for filenames in MPICH_MPIIO_HINTS
- MPI-IO collective buffering alignment(MPT 3.1 and MPT 3.2)
 - This feature improves MPI-IO by aligning collective buffering file domains on Lustre boundaries.
 - The new algorithms take into account physical I/O boundaries and the size of the I/O requests. The intent is to improve performance by having the I/O requests of each collective buffering node (aggregator) start and end on physical I/O boundaries and to not have more than one aggregator reference for any given stripe on a single collective I/O call.
 - The new algorithms are enabled by setting the MPICH_MPIIO_CB_ALIGN env variable.

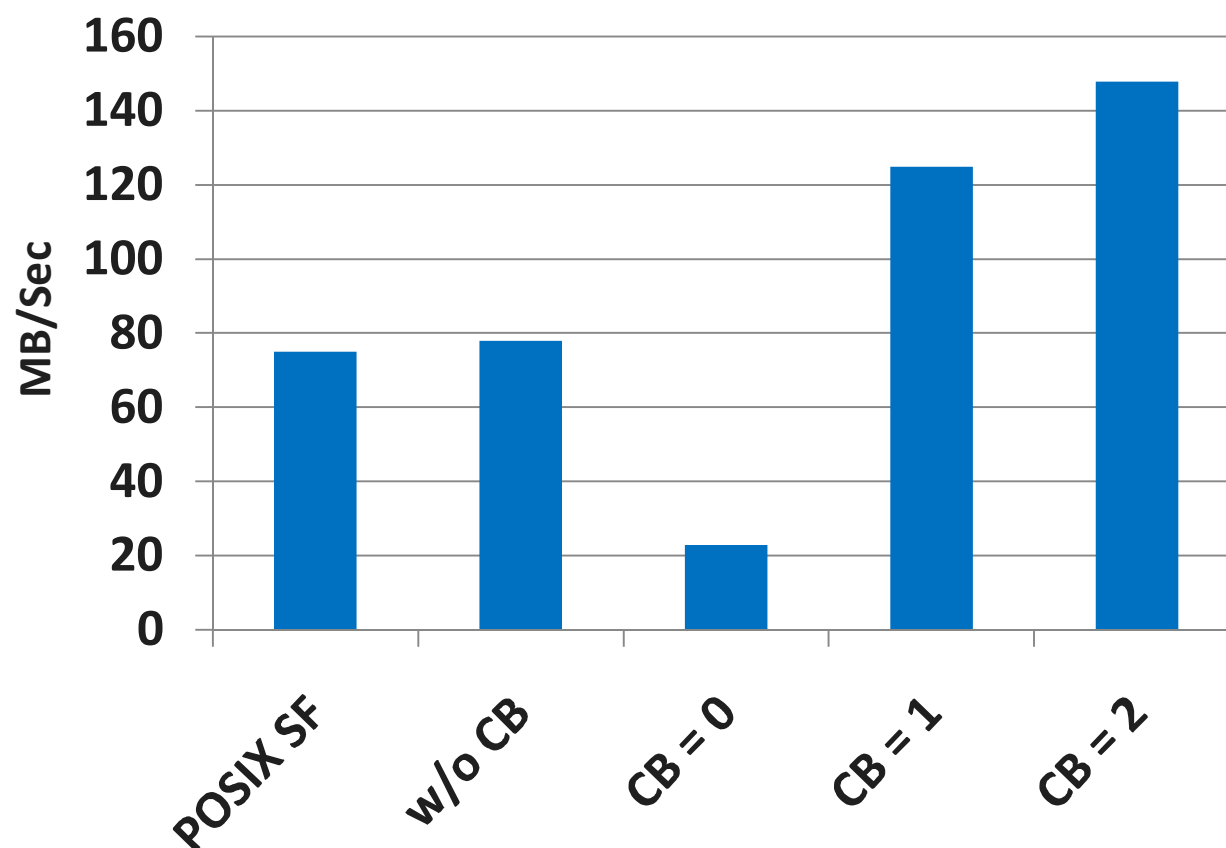
IOR benchmark 1,000,000 bytes

MPI-IO API , non-power-of-2 blocks and transfers, in this case blocks and transfers both of 1M bytes and a strided access pattern. Tested on an XT5 with 32 PEs, 8 cores/node, 16 stripes, 16 aggregators, 3220 segments, 96 GB file



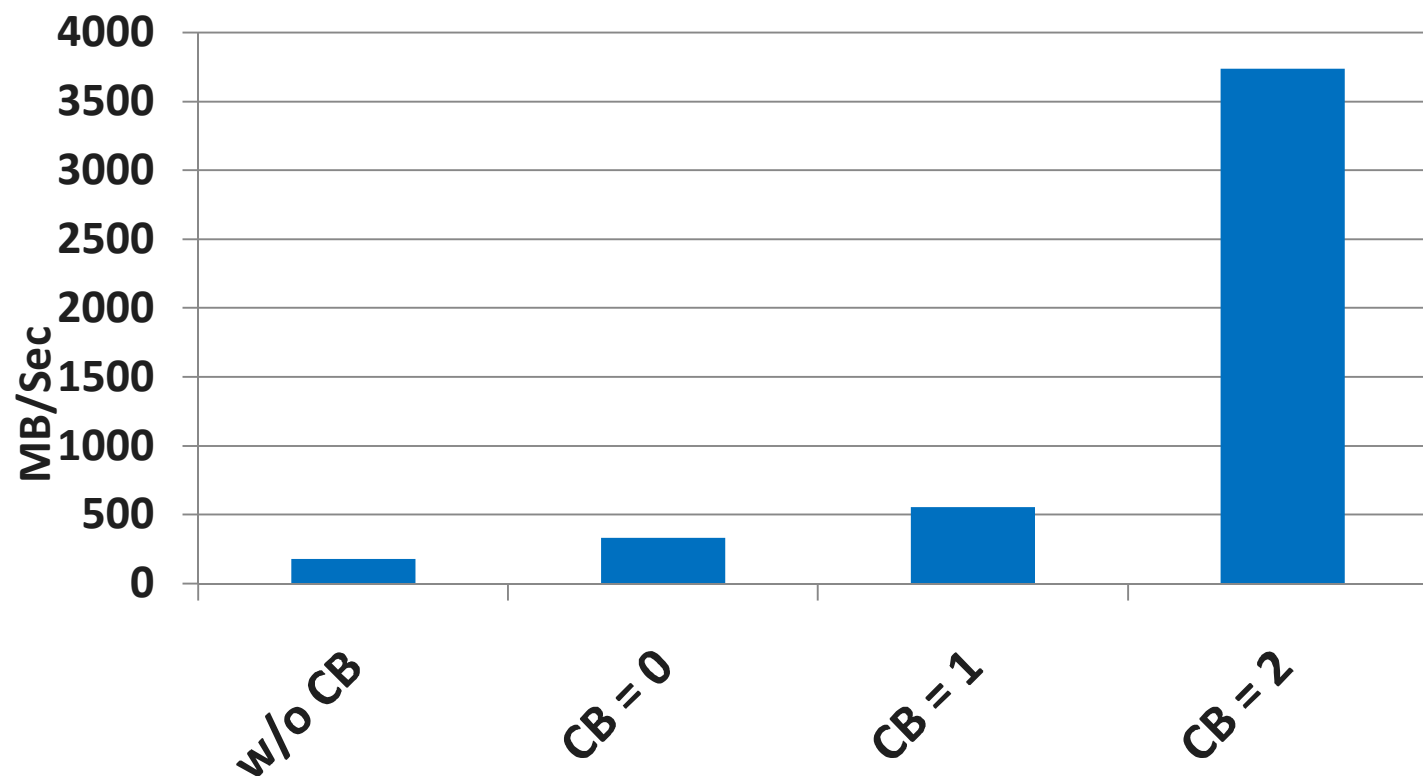
IOR benchmark 10,000 bytes

MPI-IO API , non-power-of-2 blocks and transfers, in this case blocks and transfers both of 10K bytes and a strided access pattern. Tested on an XT5 with 32 PEs, 8 cores/node, 16 stripes, 16 aggregators, 3220 segments, 96 GB file



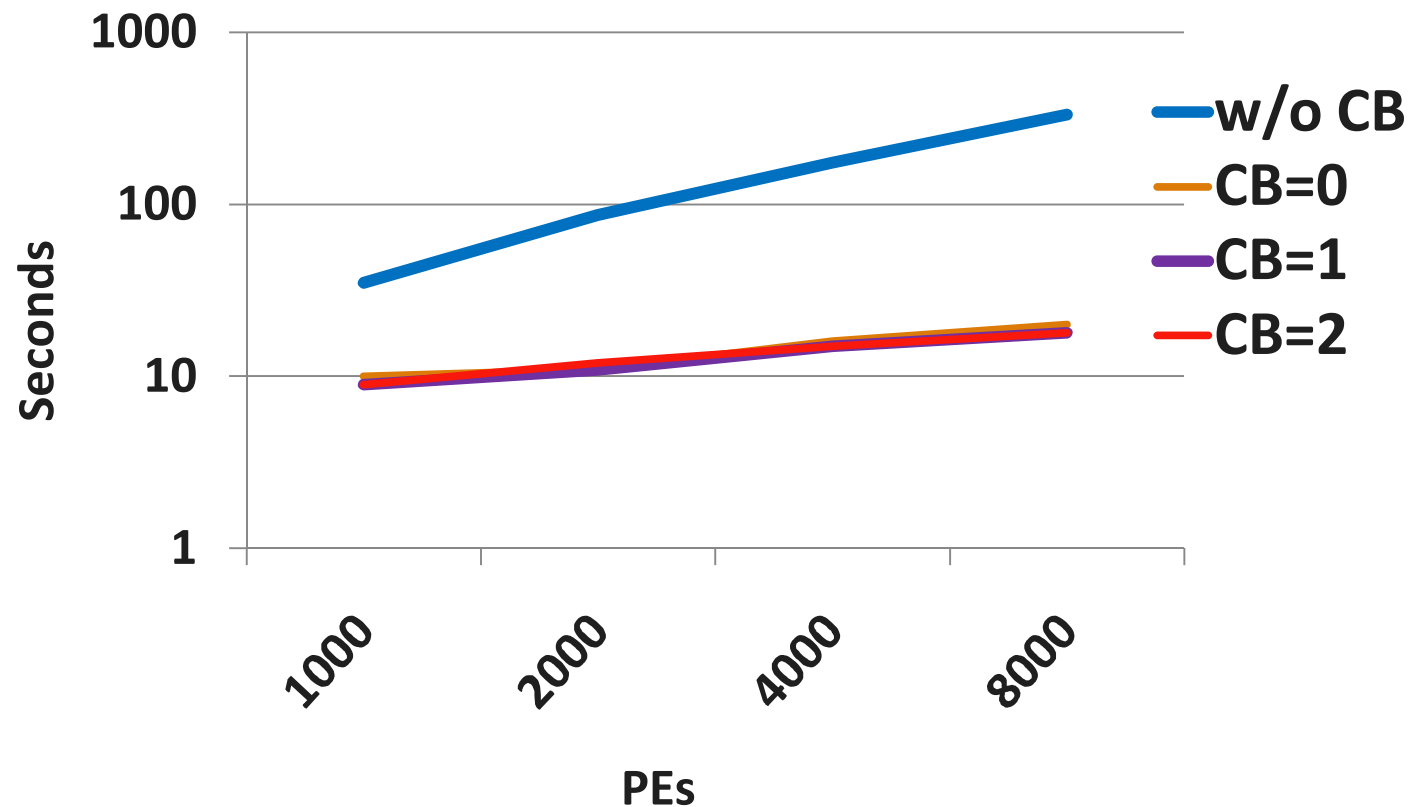
HYCOM MPI-2 I/O

On 5107 PEs, and by application design, a subset of the Pes(88), do the writes. With collective buffering, this is further reduced to 22 aggregators (cb_nodes) writing to 22 stripes. Tested on an XT5 with 5107 Pes, 8 cores/node



HDF5 format dump file from all PEs

Total file size 6.4 GiB. Mesh of 64M bytes 32M elements, with work divided amongst all PEs. Original problem was very poor scaling. For example, without collective buffering, 8000 PEs take over 5 minutes to dump. Note that disabling data sieving was necessary. Tested on an XT5, 8 stripes, 8 cb_nodes

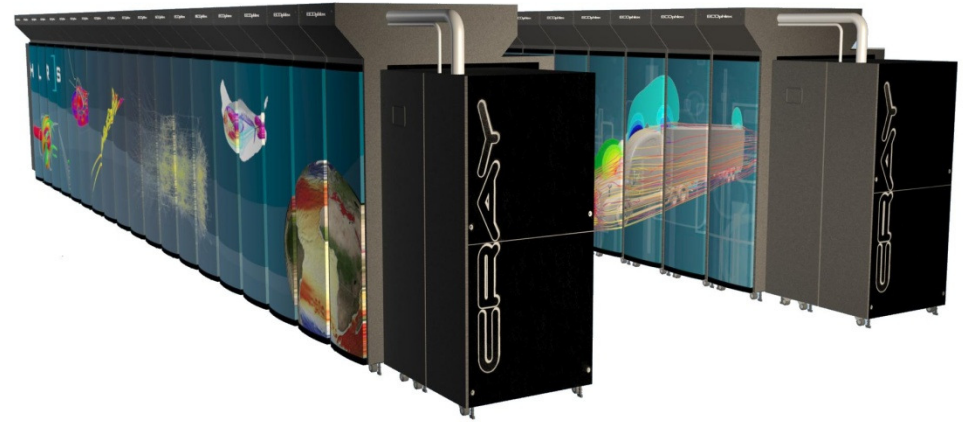


MPICH_MPIIO_CB_ALIGN

- Determines the algorithm to use for dividing IO when using MPI collective IO.
- Default value of 2 divides IO workload into Lustre stripe sized pieces and assigns them to collective buffering nodes (aggregators). Ensures that aggregators always access the same set of stripes – minimizes Lustre lock contention
- A value of 1 uses physical IO boundaries to divide IO load – like method 2 but no fixed association between file stripe and aggregator from call to call.
- Value of 0 divides IO workload amongst aggregators evenly without regard to physical IO boundaries or Lustre stripes. Inefficient if there are only a small number of stripes

MPICH_MPIIO_HINTS

- If set, override the default value of one or more MPI I/O hints. This also overrides any values that were set by using calls to `MPI_Info_set` in the application code. The new values apply to the file the next time it is opened using an `MPI_File_open()` call.
- Wildcard matching for filenames supported.
- Supported hints – `striping_factor`, `striping_unit`, `direct_io`, `romio_cb_read`, `romio_cb_write`, `cb_buffer_size`, `cb_nodes`, `cb_config_list`, `romio_no_indep_rw`, `romio_ds_read`, `romio_ds_write`, `ind_rd_buffer_size`, `ind_wr_buffer_size`
- See “Getting Started on MPI I/O” from docs.cray.com
- `MPICH_MPIIO_DISPLAY_HINTS` – rank 0 in the participating communicator displays hints



MPI

Environment variables to control rank placement

MPICH_RANK_REORDER_METHOD

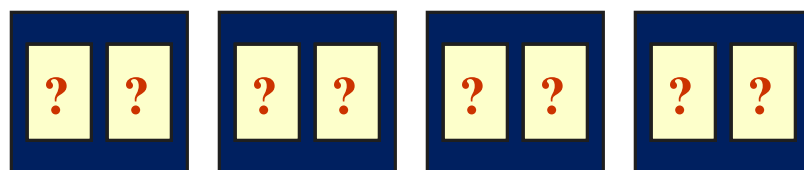
- Overrides the default MPI rank placement scheme. If this variable is not set, the default aprun launcher placement policy is used. The default policy for aprun is SMP-style placement.
- Value of 0 specifies round-robin placement. Sequential MPI ranks are placed on the next node in the list. When every node has been used, the rank placement starts over again with the first node.
- Value of 1 specifies SMP-style placement. This is the default aprun placement. For a multi-node core, sequential MPI ranks are placed on the same node.

MPICH_RANK_REORDER_METHOD, cont

- Value of 2 specifies folded-rank placement. Sequential MPI ranks are placed on the next node in the list. When every node has been used, instead of starting over with the first node again, the rank placement starts at the last node, going back to the first.
- Value of 3 specifies a custom rank placement defined in the file named MPICH_RANK_ORDER. The MPICH_RANK_ORDER file must be readable by the first rank of the program, and reside in the current running directory. The order in which the ranks are listed in the file determines which ranks are placed closest to each other, starting with the first node in the list.
- MPICH_RANK_REORDER_DISPLAY – If set causes rank 0 to display which node each MPI rank resides in.

MPI Rank Reorder

- MPI rank placement with environment variable



- Distributed placement
 - SMP style placement
 - Folded rank placement
 - User provided rank file
- How do you know which to try? And even more importantly, how do I create a custom rank file?
 - Craypat will do this for you!

MPI Rank Placement Suggestions

- When to use?
 - Point-to-point communication consumes significant fraction of the program time and have a significant imbalance
 - When there seems to be a load imbalance of another type
 - Can get a suggested rank order file based on user time
 - Can have a different metric for load balance
 - Also shown to help for collectives (alltoall) on subcommunicators (GYRO)
 - Spread out IO across nodes (POP)
- Information in resulting report
 - Custom placement files automatically generated
 - Table notes in report has instructions on how to use

Rank Order and CrayPAT

- One can also use the CrayPat performance measurement tools to generate a suggested custom ordering.
 - Available if MPI functions traced (-g mpi or -O apa)
 - `pat_build -O apa my_program`
 - see Examples section of `pat_build` man page
- `pat_report` options:
 - `mpi_sm_rank_order`
 - Uses message data from tracing MPI to generate suggested MPI rank order. Requires the program to be instrumented using the `pat_build -g mpi` option.
 - `mpi_rank_order`
 - Uses time in user functions, or alternatively, any other metric specified by using the `-s mro_metric` options, to generate suggested MPI rank order.

Reordering with CrayPAT Workflow

- module load perftools
- Rebuild your code
- `pat_build -O apa a.out`
- Run `a.out+pat`
- `pat_report -Ompi_sm_rank_order a.out+pat+...sdt/ > pat.report`
- Creates `MPICH_RANK_REORDER_METHOD.x` file
- Then set env var `MPICH_RANK_REORDER_METHOD=3` AND
- Link the file `MPICH_RANK_ORDER.x` to `MPICH_RANK_ORDER`
- Rerun code

CrayPAT example

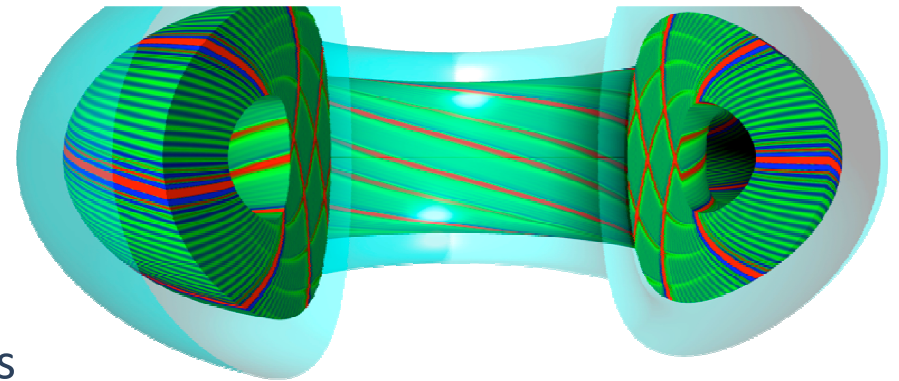
Table 1: Suggested MPI Rank Order

Eight cores per node: USER Samp per node						
Rank	Max	Max/	Avg	Avg/	Max Node	
Order	USER Samp	SMP	USER Samp	SMP	Ranks	
d	17062	97.6%	16907	100.0%	832,328,820,797,113,478,898,600	
2	17213	98.4%	16907	100.0%	53,202,309,458,565,714,821,970	
0	17282	98.8%	16907	100.0%	53,181,309,437,565,693,821,949	
1	17489	100.0%	16907	100.0%	0,1,2,3,4,5,6,7	

- This suggests that
 1. the custom ordering “d” might be the best
 2. Folded-rank next best
 3. Round-robin 3rd best
 4. Default ordering last

Reordering example

GYRO



- GYRO 8.0
 - B3-GTC problem with 1024 processes
- Run with alternate MPI orderings
 - Custom: profiled with with `-O apa` and used reordering file `MPICH_RANK_REORDER.d`

Reorder method	Comm. time
Default	11.26s
0 – round-robin	6.94s
2 – folded-rank	6.68s
d-custom from apa	8.03s

CrayPAT
suggestion
almost right!

Reordering example

TGYRO

- TGYRO 1.0
 - Steady state turbulent transport code using GYRO, NEO, TGLF components
- ASTRA test case
 - Tested MPI orderings at large scale
 - Originally testing weak-scaling, but found reordering very useful

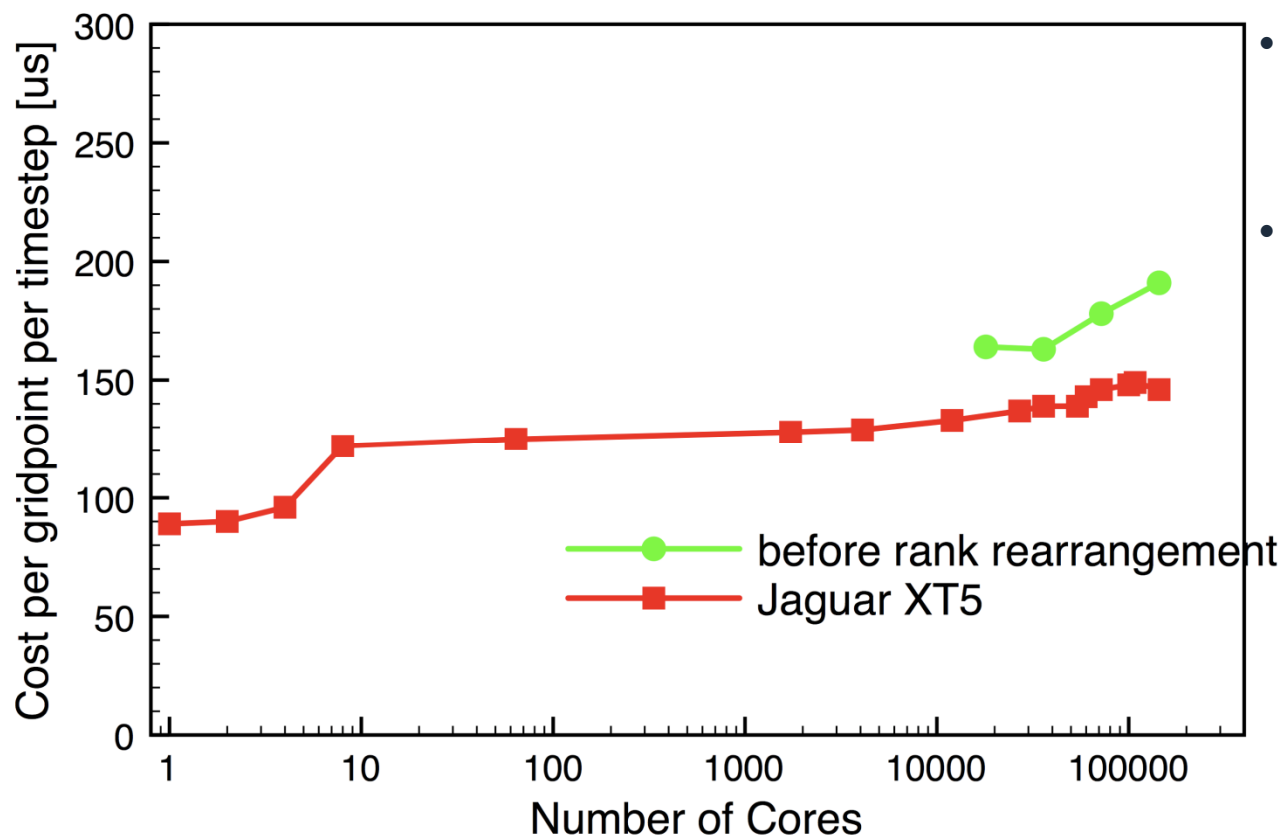
Reorder method	TGYRO wall time (min)		
	20480	40960	81920
Default	99m	104m	105m
Round-robin	66m	63m	72m

Huge win!



Rank Reordering Case Study

- Application data is in a 3D space, $X \times Y \times Z$.
- Communication is nearest-neighbor.
- Default ordering results in $12 \times 1 \times 1$ block on each node.
- A custom reordering is now generated: $3 \times 2 \times 2$ blocks per node, resulting in more on-node communication



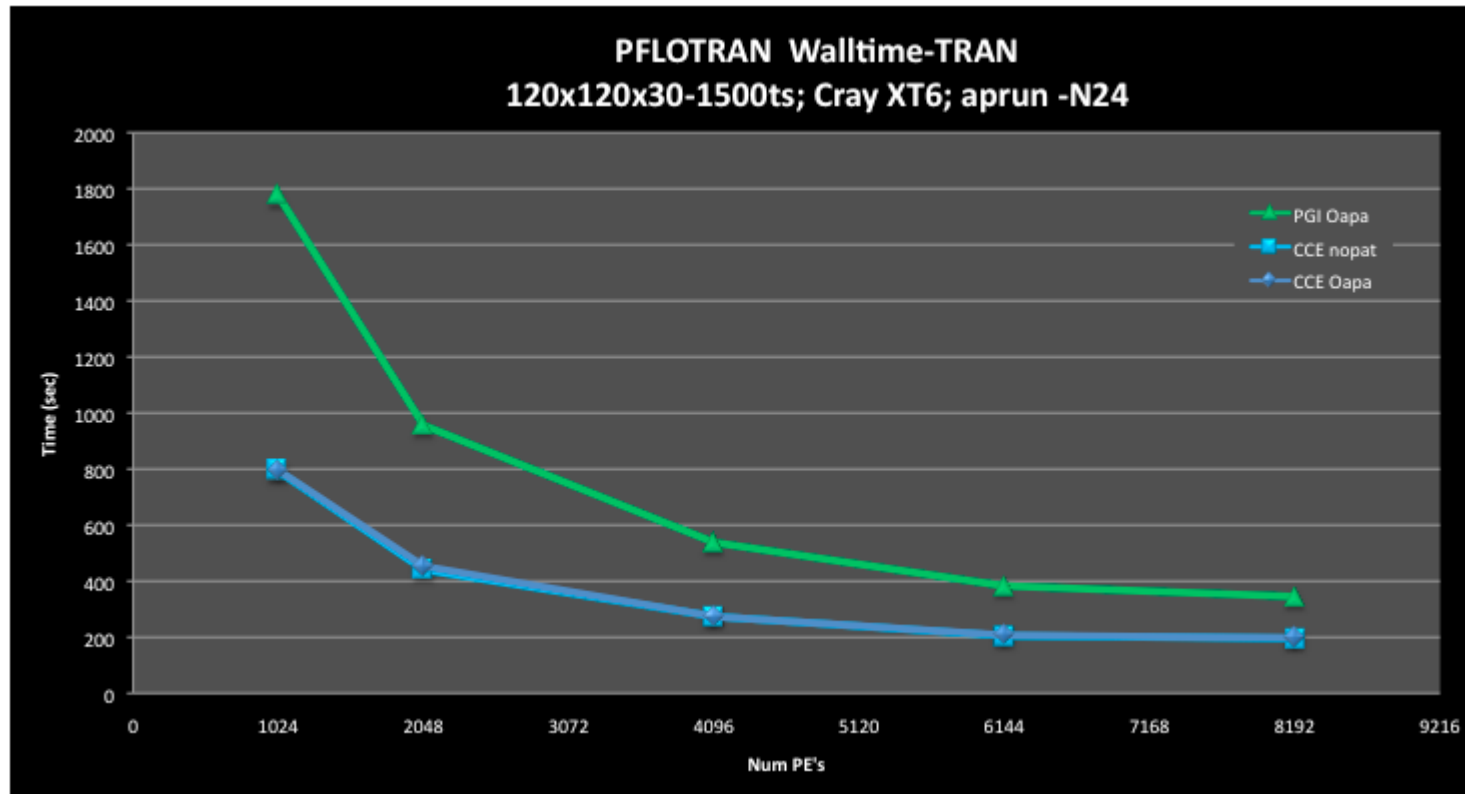
Rank Reorder Case Study: PFLOTRAN

- Instrument binary
 - % module load perftools
 - % make pflotran
 - % pat_build -Oapa pflotran
 - Results in executable pflotran+pat
- Run PFLOTRAN
 - Results in raw data file
- Process collected data
 - Results in text report
 - Customized instrumentation template file for next experiment
 - file for input into Cray Apprentice2
 - csv format available for scalability graphs

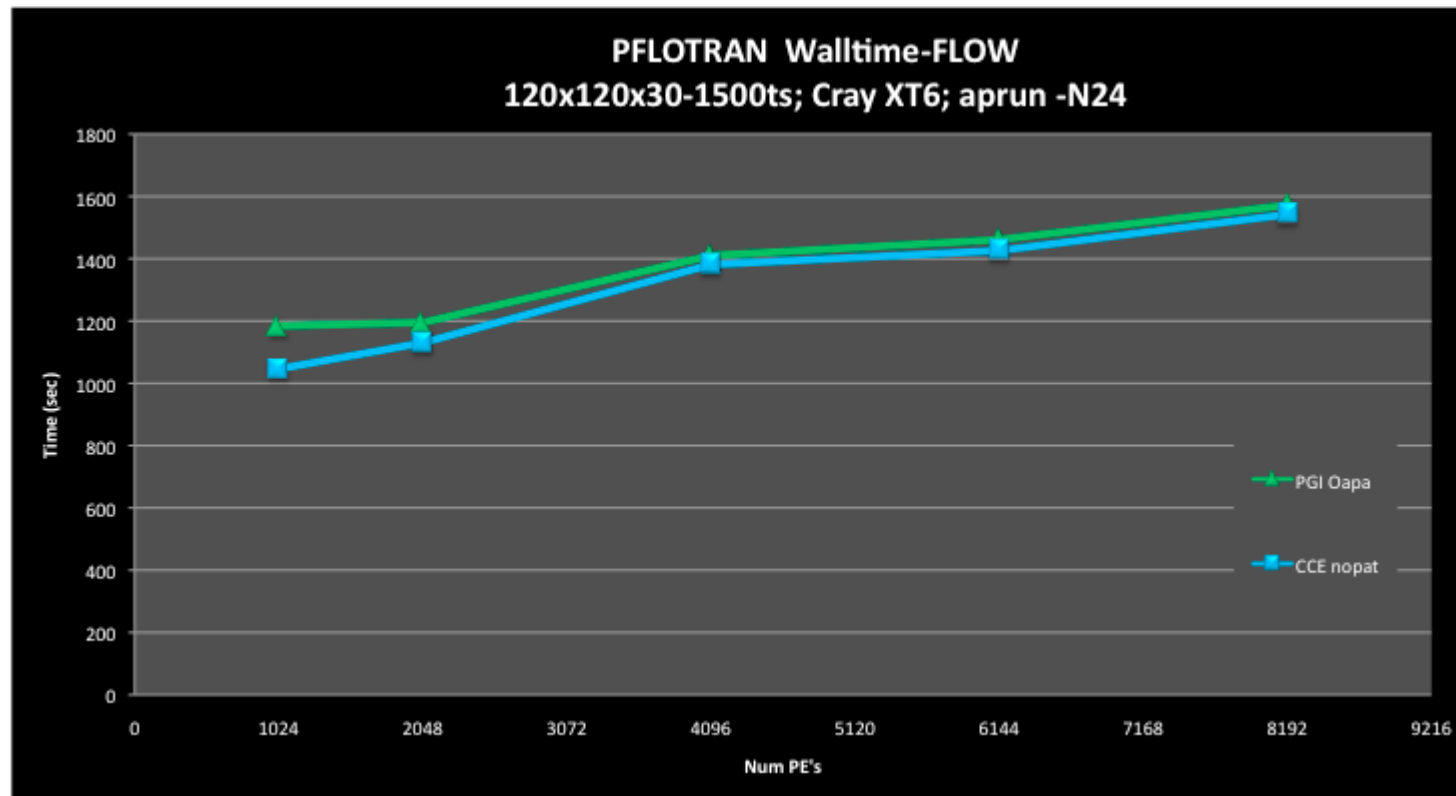
Performance Data Analysis

- Used csv format to plot time vs MPI ranks for top routines to look at scaling
- Found good scaling with computation (TRAN) phase
- Found poor scaling with communication (FLOW) phase
 - Candidate for MPI rank placement suggestions
 - Candidate for load imbalance analysis
- Generated MPI rank placement suggestions
 - `% pat_report -O mpi_sm_rank_order pflotran+apa.ap2`
- Used visualization tool to look for load imbalance

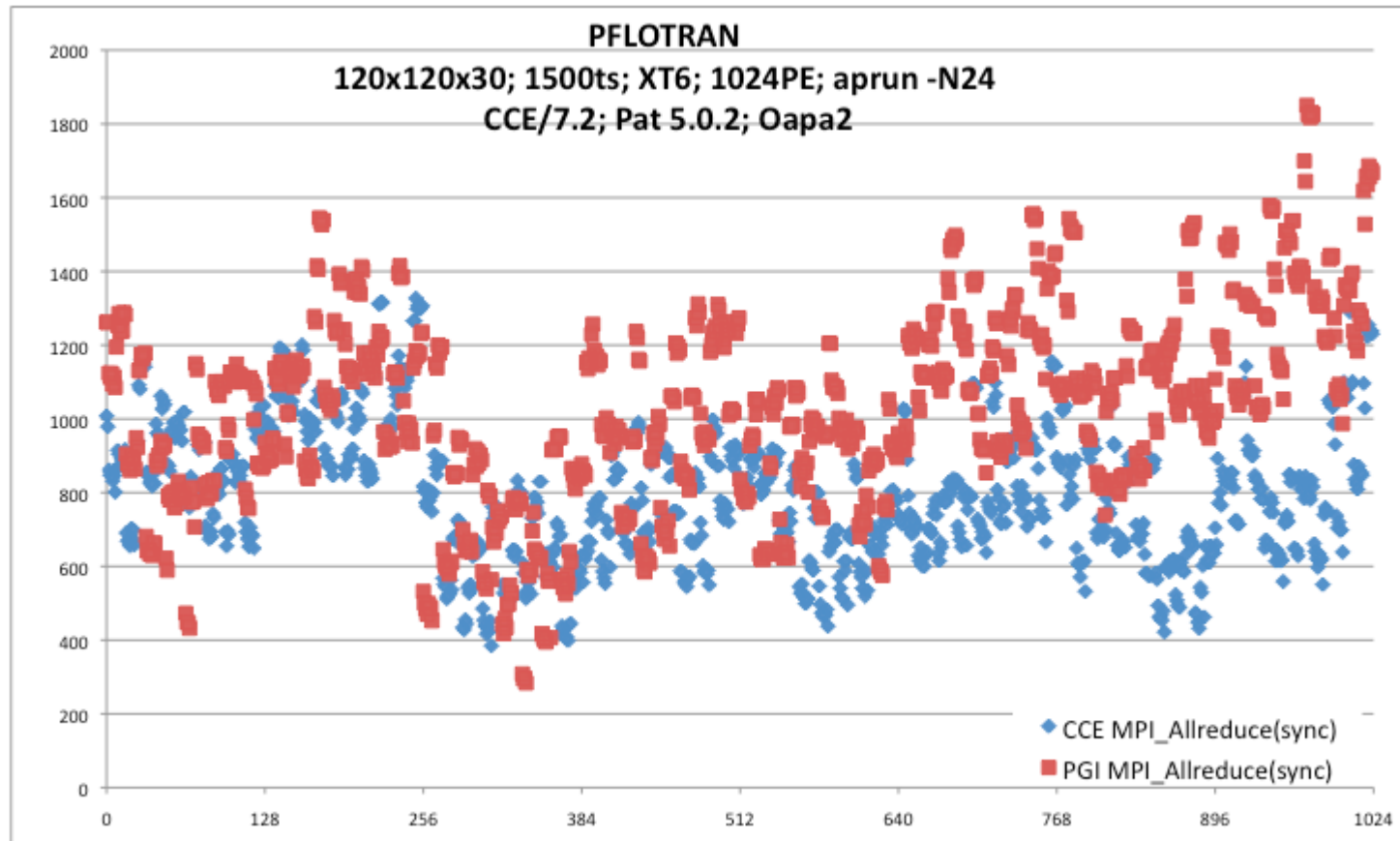
TRAN Phase Scales



FLOW Phase Doesn't Scale



Allreduce synchronization time – sorted by ranks



Experiment with rank placement suggestions

- Run with placement file provided by pat_report
- PFLOTRAN reported wall clock times:
 - Default MPI rank placement on (1024 ranks)
 - Wall Clock Time: 2,065.4 sec
 - Placement from auto-detected communication pattern
 - Wall Clock Time: 1,489.1 sec
- > 25% speedup with no source file changes!

Documentation

- man intro_mpi
- <http://www-unix.mcs.anl.gov/mpi/index.html>
- <http://www-unix.mcs.anl.gov/mpi/mpich2>
- <http://www.mpi-forum.org/>
- <http://docs.cray.com/>