Optimizing MPI Collectives for X1

Howard Pritchard
(howardp@cray.com)
Outline

• X1 MPI – previous work

• X1 and MPI

• New and not so new algorithms for collective operations

• Results
X1 MPI History

• Started from CRAY/SGI MPI
  – coded for RISC microprocessor
  – designed for cluster of SMPs
  – some cluster aware collective optimizations

• Simplified code by removing cluster awareness

• Replaced point-to-point MPI in collectives with AMO's and pointer exchanges
These changes helped, especially for small system sizes

- Eliminate scalar overhead of point-to-point
- In some cases auto-tasking like approaches used (MPI_Bcast)

But some scalability problems showed up for larger system sizes
X1 MPI History(3) - Problems

• Memory hot spots owing to inefficient use of memory banks

• AMOs sometimes caused memory hot spots

• Low aggregate bandwidth in some cases
Barotropic portion of POP
Generally does not scale well
Cray X1 Co-Array Fortran performance is excellent
Cray X1 MPI performance not as good
X1 and MPI – Good Things

- Distributed Memory Program Model makes things easy
  - Don't have to learn another RDMA protocol
  - No need to use buffers unless you want to, just exchange pointers

- Hundreds of outstanding load/stores. Order of magnitude(s) greater than scalar processors.
  - Able to poll vectors of values directly from memory
  - Fast vector stores across many nodes
  - Efficient cache coherency protocol
Vectors let you do everything at once(almost)

Results from a kernel in which one process in an application team of 128 processes issues one or more vector stores striding across other processes followed by a \textit{gsync} and a succeeding vector store.

X1 vector hardware lets one deliver 80 bytes to 128 process memory in less than 5 $\mu$secs!
X1 and MPI – Gotchas

- Need to pay attention to memory bank conflicts (network latency)

- Don't use AMO's when lots of processes are involved – no AMO cache on X1

- Only use gsynCs when necessary, they are expensive when lots of outstanding stores to remote memory have been issued
Memory-bank conflicts

Results from a kernel in which one process in an application team of 128 processes issues one or more vector stores striding across other processes followed by a gsync and a succeeding vector store.

This time the experiment is run with and without padding of the target array.

X1 and MPI – Bad Things

- Slow scalar processor with poor branch prediction
- High function call overhead
- Particular issues with SSP mode
Strategy for Optimizations

• Optimize for short messages first
  – Avoid use of AMO's for synchronization
  – Rely on vector polling and strided or scatter puts

• Better traditional algorithms for longer messages for some operations
Optimizing Short Messages

- Use insights from applications analysts' and benchmarkers' CoArray (CAF) workarounds for MPI problems

- Double buffering with padding to reduce synchronization and avoid memory bank conflicts. Data structures associated with MPI internal communicator structure

- Use arrays of pointer functions to cut down on branches
Most MPI implementations focus on minimum startup cost for collectives involving short messages:

MPI_Allreduce

No vector content here, multiple synchronizations.
For X1 short message collectives are best treated as a vectorization problem – with vectorization over the process dimension

MPI_Allreduce (up to 64 ranks)
MPI_Allreduce (short message)

Vector algorithm is over an order of magnitude faster at higher process counts
Optimizing Short Messages (4)

- Optimized Mellor-Crummey & Scott (MCS) tree barrier (radix 64) for MPI_BARRIER and internal barrier for use in collectives - doesn't use AMOs [MCS]

- Optimized internal MPI_Allgather and MPI_Alltoall for short messages to enable efficient gathering of pointers, datatypes, etc. for use in medium and long length operations
Variants of the MCS tree barrier are under investigation for MPI_Barrier, MPI_Win_fence, and shmem_barrier_all
MPI_Alltoall (short message)

![Graph showing MPI_Alltoall short message overhead (8 bytes)]
Optimizing for longer messages

- Use optimized barrier and short message allgather and alltoall to avoid AMO base synchronization

- Use *put* rather than *get* approach for moving data, gives better bandwidth

- Stream over target rank for medium length messages (MSP mode library)
MPI_Allreduce - longer vectors

- Vectors greater than 32 bytes but 128 or fewer vector elements use binary tree algorithm with buffers associated with communicator to reduce synchronization requirements

- Vectors for which $\frac{nelements}{nranks} \leq 64$ use binary tree with application buffers

- For vectors with $\frac{nelements}{nranks} > 64$, use a reduce-scatter/gather algorithm [geijn,rab]
Better switchover criteria to reduce-scalar/gather approach are being investigated.
MPI_Alltoall - longer vectors

![Graph showing MPI_Alltoall overhead for different vector lengths. The graph plots time (secs) on the y-axis and vector length (bytes) on the x-axis. Two lines are plotted: MPTDEV in red and MPT 2.3 in green. The overhead increases as the vector length increases.]
Application Results

- Many times a timing profile that shows a lot of time being spent in MPI is NOT an MPI problem – especially for collective calls

- Use PAT and a MPI trace library like FMPI to make sure there are no load balance problems – an application which is load balanced on a Beowulf cluster may not be load balanced on X1
The allreduce in CAF and MPT 2.4 are very similar. CAF benefits a lot from inlining of the global sum into the solver.
New MPI_Allgatherv/MPI_Alltoallv with streaming give similar performance to a CAF version of CAM.
Release Process

- Use an internal development library to test new algorithms (MPTDEV)

- After testing, algorithms are integrated into the MPT 2.4 pre-release tree

- Selected mods are pushed back into MPT 2.3 release version

- MPT 2.4 planned for release in fall '04 – all collectives optimized in this release
What's Next?

- Improvements in point-to-point latencies for X1 MPI
- Cray-RS collectives?
References

