

# Coping at the User-Level with Resource Limitations in the Cray Message Passing Toolkit MPI at Scale: **How Not to Spend Your Summer Vacation**

Richard T. Mills<sup>1</sup>, Forrest M. Hoffman<sup>1</sup>, Patrick H. Worley<sup>1</sup>,  
Kalyan S. Perumalla<sup>1</sup>, Art Mirin<sup>2</sup>, Glenn E. Hammond<sup>3</sup>, and  
Barry F. Smith<sup>4</sup>

<sup>1</sup>Oak Ridge National Laboratory, <sup>2</sup>Lawrence Livermore National Laboratory,  
<sup>3</sup>Pacific Northwest National Laboratory, <sup>4</sup>Argonne National Laboratory

**Cray Users Group Meeting • May 6, 2009 • Atlanta, GA**

# We've Experienced An Explosion of Processor Cores

- The number of processing elements being deployed in Cray XT series supercomputers has grown at a prodigious rate!
- The Cray XT5 Jaguar machine at Oak Ridge National Laboratory—**Number 2 on the Top 500 List at 1.059 PFlop/s**—has 150,152 processor cores, 30× that of the original Red Storm XT3 at Sandia National Laboratories!
- **But** along with this growth has come increasing difficulty in scaling MPI codes due to limits in message passing resources.



# Coping Mechanisms

- Some problems due to resource limitations in **Portals** or the **Cray Message Passing Toolkit (MPT)** can be mitigated by setting appropriate environment variables to increase limits or specify alternative algorithms (Johansen, CUG 2008).
- While such settings—usually arrived at by trial and error—may allow a code to run to completion, they can have negative impacts on performance and/or starve the application of needed memory.
- Alternatively, user-level solutions may significantly improve code performance at scale without reducing available memory or resorting to disabling performance-enhancing features of Portals/MPT.
- Apparently arriving at this solution independently, a growing number of Cray XT users have implemented user-level flow control schemes in their application codes.

# Symptoms of Illness

- The Cray Message Passing Toolkit (MPT) is based on **MPICH2 from Argonne National Laboratory** and supports two abstract device interfaces (ADI3): **Portals** for inter-node communication and **SMP** for intra-node communication.
- **Portals** uses an *eager protocol* for sending short messages, assuming that the destination process can buffer or directly store these data.
- If the destination process has posted a matching receive, the data are placed in the user-supplied buffer; otherwise, they are placed in the *unexpected buffer* and two entries are generated in the *unexpected event queue*: a *put start* event and a *put end* event when the data are ready to be used.
- Exhaustion of the *unexpected buffer* and/or overflow of the *unexpected event queue* frequently occur when scaling up application codes.

## Getting Professional Help

- The size of the *unexpected buffer* can be set using the `MPICH_UNEX_BUFFER_SIZE` environment variable.
- The length of the *unexpected event queue* can be set using the `MPICH_PTR_UNEX_EVENTS` environment variable.
- Increasing the default values will decrease the amount of memory available to the application, and it may not be possible to set them large enough to avoid program failures.
- Alternatively, the number of unexpected messages can be decreased by lowering the default maximum size of short message sizes using the `MPICH_MAX_SHORT_MESSAGE_SIZE` environment variable.
- As a final solution, setting `MPICH_PTL_SEND_CREDITS` to `-1` will use a flow control mechanism to prevent overflow of the *unexpected event queue* in any situation.

## Other Conditions

- The **Portals** *other events queue* is used for all other MPI-related events, including MPI-2 remote memory access (RMA) requests, sending of data (*send end* and *reply end* events), and pre-posted receives.

Restructuring code to pre-post receives to avoid failures resulting from *unexpected events* may result in failures due to too many *other events* being generated!

- The size of the *other events queue* can be increased using the **MPICH\_PTL\_OTHER\_EVENTS** environment variable.
- The **SMP** device can cause failures due to the limit on the maximum number of internal MPI message headers, which can be increasing using the **MPICH\_MSGS\_PER\_PROC** environment variable.

# Patient #1: Parallel $k$ -means Cluster Analysis

- Communication in this typical master/slave application involves primarily one-to-one message passing in which
  - 1 the master assigns blocks of work to each slave,
  - 2 each slave works independently then reports back to the master with results (implicitly requesting another block of work),
  - 3 at the end of an iteration each slave sends additional summary data to the master, and
  - 4 the master recomputes centroid locations and broadcasts them to the slaves.

These steps repeat until some convergence criterion is met.

- A new acceleration algorithm was recently added in which some or all slave processes cooperate in the sorting (in parallel) of distance vectors that are then gathered to all slaves using *MPI\_Allgatherv()*.

# Symptoms and Treatment

- For the AmeriFlux data set with  $k = 8000$  clusters, the code performs best using 1,025 cores (1,024 slaves + 1 master).
- With Cray MPT 3.0.3 the program crashes as follows in *MPI\_Allgatherv()* at 2,049 processes, but runs to completion at 4,097 processes.

```
[128] MPICH has run out of unexpected buffer space.  
Try increasing the value of env var MPICH_UNEX_BUFFER_SIZE (cur value is 62914560),  
and/or reducing the size of MPICH_MAX_SHORT_MSG_SIZE (cur value is 128000).  
aborting job:  
out of unexpected buffer space
```

- Setting **MPI\_UNEX\_BUFFER\_SIZE** to  $4\times$  the default of 60 MB allowed the program to run in 28 min.
- With Cray MPT 3.1.0, the same problem runs in 14 min without raising **MPI\_UNEX\_BUFFER\_SIZE**, but runs in 28 min when setting the environment variable to  $4\times$  the default of 60 MB.



# Pre-Posting Therapy

- Additional development was performed to pre-post receives (using *MPI\_Irecv()*) on the master process just before each block of work is assigned to a slave.
- On the AmeriFlux problem on the Cray XT4, the program crashes with the following error:

```
[0] : (/tmp/ulib/mpt/nightly/3.1/112008/mpich2/src/mpid/cray/src/adi/ptldev.c:2854)
PtlMEMDPost() failed : PTL_NO_SPACE
aborting job:
PtlMEMDPost() failed
```
- Apparently, the 2,048 pre-posted receives (of single long integers) exceed some Portals resource limit, but this error was eliminated by disabling the registration of receive requests in Portals by setting **MPICH\_PTL\_MATCH\_OFF**, with ~15% longer runtime than the previous code on the same problem.
- **In this case, pre-posting receives requires disabling a communications feature on the XT4 and it has a deleterious effect on performance.**

## Patient #2: Subsurface Flow and Reactive Transport

- Groundwater code PFLOTRAN is fairly communication-intensive:
  - Message passing (“3D halo exchange”) at subdomain boundaries
  - Gathers of off-processor vector entries for matrix-vector products
  - Numerous *MPI\_Allreduce()* calls inside Krylov solvers
- Despite this, most phases of code are robust in terms of MPT resources.
- Exception is PETSc *VecView()* calls in checkpointing.

# Symptoms and Treatment

- *VecView()* serialized I/O through process 0
  - Nonzero ranks all post sends to process 0
  - Process 0 loops through sends
- Causes problems with the *unexpected buffer* and *unexpected event queue*
- For very large jobs the size of these must be increased to impractical size for the *VecView()* to complete
- Initial treatment: Change *MPI\_Send()* to *MPI\_Ssend()*. No dice.
- Second treatment: Wrote MPI-IO backend for *VecView()*. Crashed with “*PtlMEMDPost() failed : PTL\_NO\_SPACE*”.

# Symptoms and Treatment

- Third treatment: Add flow control
  - All procs participate in series of broadcasts, with proc 0 sending out minimum rank, increased in increments of `flow_control_group_size`.
  - Ranks  $\geq$  minimum may send their message.
- 3.5th treatment: Eliminate broadcasts
  - Form disjoint sets of `flow_control_group_size`.
  - Form sub-communicator from union of these with proc 0.
  - At initiation of `VecView()`, each process issues `MPI_Barrier()` on its subcommunicator.
  - Proc 0 joins each barrier only when ready to process messages from that group.
- Performance of both is identical: Time dominated by write to disk.
- Eventual happy ending: In MPT 3.1.x, using `MPICH_MPIIO_CB_ALIGN` makes the MPI-IO backend not only work, but work very fast!

## Patient #3: Community Atmosphere Model

- Default communication protocol is to pre-post all receive requests, issue all (non-blocking) send requests, then wait for receives to be satisfied.
- At scale, potential of overwhelming any given process with messages for which it has not posted receives.
- Can cause failures if the system cannot allocate sufficient system buffer space to handle all of the requests, and will degrade performance in any case with all of the additional buffer copying.

# Symptoms and Treatment

- Anomalously large communication times when transposing between two dynamics decompositions in which one decomposition was three times smaller.
- Apparently the early arrival of messages from the otherwise idle two-thirds of the processes at the one-third active processes was causing the performance anomaly.
- Similar performance problems have been observed in both gather and scatter operations (using both MPI collective calls and point-to-point implementations).
- Setting appropriate MPI environment variables does eliminate the errors in the previous two examples IF size can be set large enough.

# Symptoms and Treatment

- Use flow control in form of handshaking messages
  - After each non-blocking receive is posted, a “zero-byte” message is sent to the source process.
  - Upon receipt of this signal, the source process can send the message. This eliminates all unexpected messages of size greater than zero.
- Still a potential problem in pre-posting more non-blocking receive requests than are supported on a given system (with any given MPI environment variable settings).
- Potential performance impact from having a large number posted, if only in the cost of matching receive requests with the incoming messages.
- Potential of overwhelming a given process with the handshaking messages.

## Patient #4: $\mu$ sik PDES

- In n PDES, simulation time can be an arbitrary real value (typically, a double precision floating point value), with state updates “scheduled” to be executed in arbitrary time instants in the future on the real-valued simulation-time axis.
- Each processor can potentially be updating its system state at simulation time instants that are later or earlier than the simulation time of other processors.
- Processors exchange data using “timestamped” events scheduled by one processor to another.
- Events, which are payload (data) tagged with a timestamp value, are thus scheduled for arbitrary times in the future.
- To ensure global causality, events are constrained to be executed at every processor in non-decreasing timestamp order.



## Patient #4: *μ*sik PDES

- Schedulability of updates, the timestamp-ordered event execution, and the staggered processing of events across processors, dictate unique parallel execution style of PDES.
- Processors are typically highly staggered in simulation time. so even small levels of local jitter due to communication overheads can accumulate globally.
- Message sizes are typically small (in the range of 64 to 512 bytes each), but the number of messages and the frequency of sends can be very high ( $10^2$ – $10^4$  of inter-processor messages per second).
- Non-blocking messaging is used heavily.
- All these messaging characteristics serve to stress the communication subsystems, amplifying even the slightest inefficiencies and overheads.

# Symptoms and Treatment

- Works up to 32768 cores. At 65536, unexpected events buffer is exhausted.
- Difficult to debug: showed up as segfault due to violation of FIFO message ordering assumptions made in the code.
- Communicating cores must be throttled, or the amount of messages must be reduced.
- First treatment: increase parameter  $\rho$ , which decreases inter-processor communication. Very poor performance.
- Random selection of the destination for the messages, spanning the entire range of MPI ranks, makes the destination list highly dynamic when only a small number of “active connections” per processor are established and maintained by the Cray’s network system.
- Active connection table was easily being flushed and refilled with newer connections established on demand to newer destinations at runtime, resulting in significant latencies.

# Symptoms and Treatment

- Second treatment: Flow control: For every sender-receiver pair exchanging events, user-level ack send by receiver once every  $k$  events; sender stalls to receive the ack before transmitting next message.
- Ensures that the network subsystem is burdened by at most  $k$  messages for any given  $(i, j)$  ordered pair of MPI ranks.
- Improves the performance dramatically, bringing the amortized event cost down to around  $100 \mu\text{s}$  on 100 000 cores.
- Additional improvement: Use message bundling such that *MPI\_Bsend()* is invoked only outside of an event computation (instead of during event computation), making it somewhat less vulnerable to the delays in *MPI\_Bsend()* processing.

# Conclusions

- Resource limitations on MPT can cause various symptoms of illness.
- We can get professional help (from Cray, etc.), e.g., *MPICH\_PTL\_SEND\_CREDITS*.
- We might need second opinions.
- But, ultimately, healing may be a slow process that requires user involvement.