Poughkeepsie Unix Development Lab

MPI - Today and Tomorrow

ScicomP 9 - Bologna, Italy

Dick Treumann - MPI Development

March 26, 2004

The material presented represents a mix of experimentation, prototyping and development.

While topics discussed may appear in some form in future IBM products there is no guarantee any particular feature will appear precisely as described.

Some work described may never go farther than prototype form.

Topics

- Collective Communications enhancements
- Improvements in MPI-IO
- Some useful environment variables
- Parallel Environment enhancements
- HPS performance enhancements
- Considering use of Large Pages

Collective Communications Enhancements
64 Bit executables only

MPI_Barrier

MPI_Bcast

MPI_Scatter, MPI_Scatterv

MPI_Gather, MPI_Gatherv

MPI_Reduce, MPI_Allreduce, MPI_Reduce_scatter

Steps of an optimized Collective

When we consider optimizing a collective communication to take advantage of shared memory we identify 3 stages. Some collectives lack either the first or the last stage.

On-node prolog using shared memory - one task per node is "node leader"

Inter-node collective communication involving only node-leaders (hidden subset communicator)

On-node epilog using shared memory - one task per node is "node leader"

The prolog and epilog can use whatever shared memory algorithm is fastest. The Inter-node step can use the base collective but just among node leaders.

MPI_ALLREDUCE as an example

PROLOG

The tasks on each node cooperate in a shared memory based reduce.

The node leader ends up with node's partial result

INTER_NODE

Some node's leader gets to be root for this stage

MPI_Reduce among node leaders gives full result to root

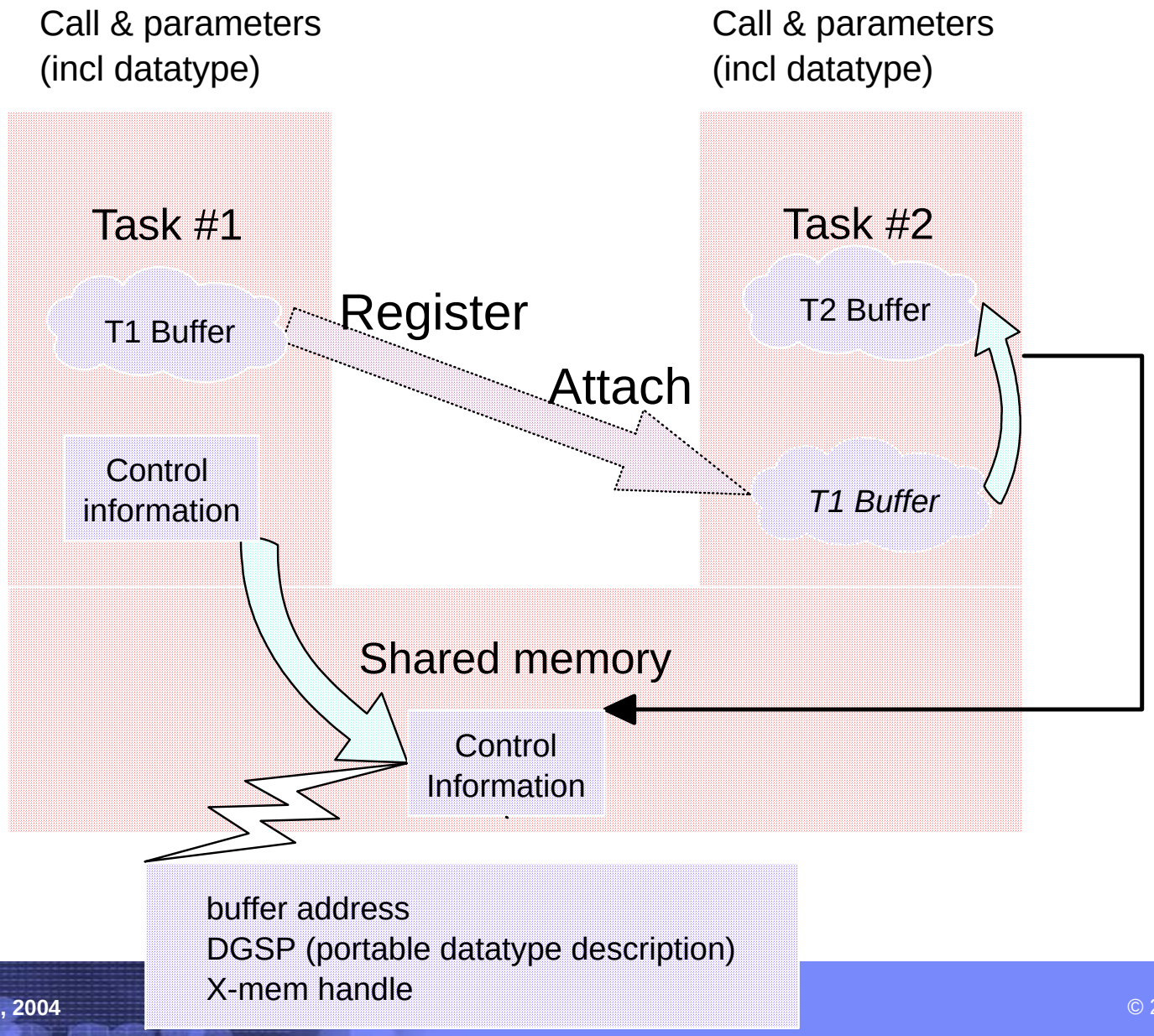
MPI_Bcast from root to other node leaders delivers full result to all node leaders

EPILOG

Every node leader now has the reduction result

The tasks on each node cooperate in a shared memory based broadcast

Cross Memory Attach for on node portion



The Impact of Cross memory Detach costs

Each detach operation has OSI wide impact

Their time cost grows as CPU count grows

Detach operations serialize in the kernel

These costs have limited the value of using xmem attach in collective operations.

We are working on a lazy detach model which will allow system wide detach impacts to occur once for a large number of enqueued detach operations rather than for each.

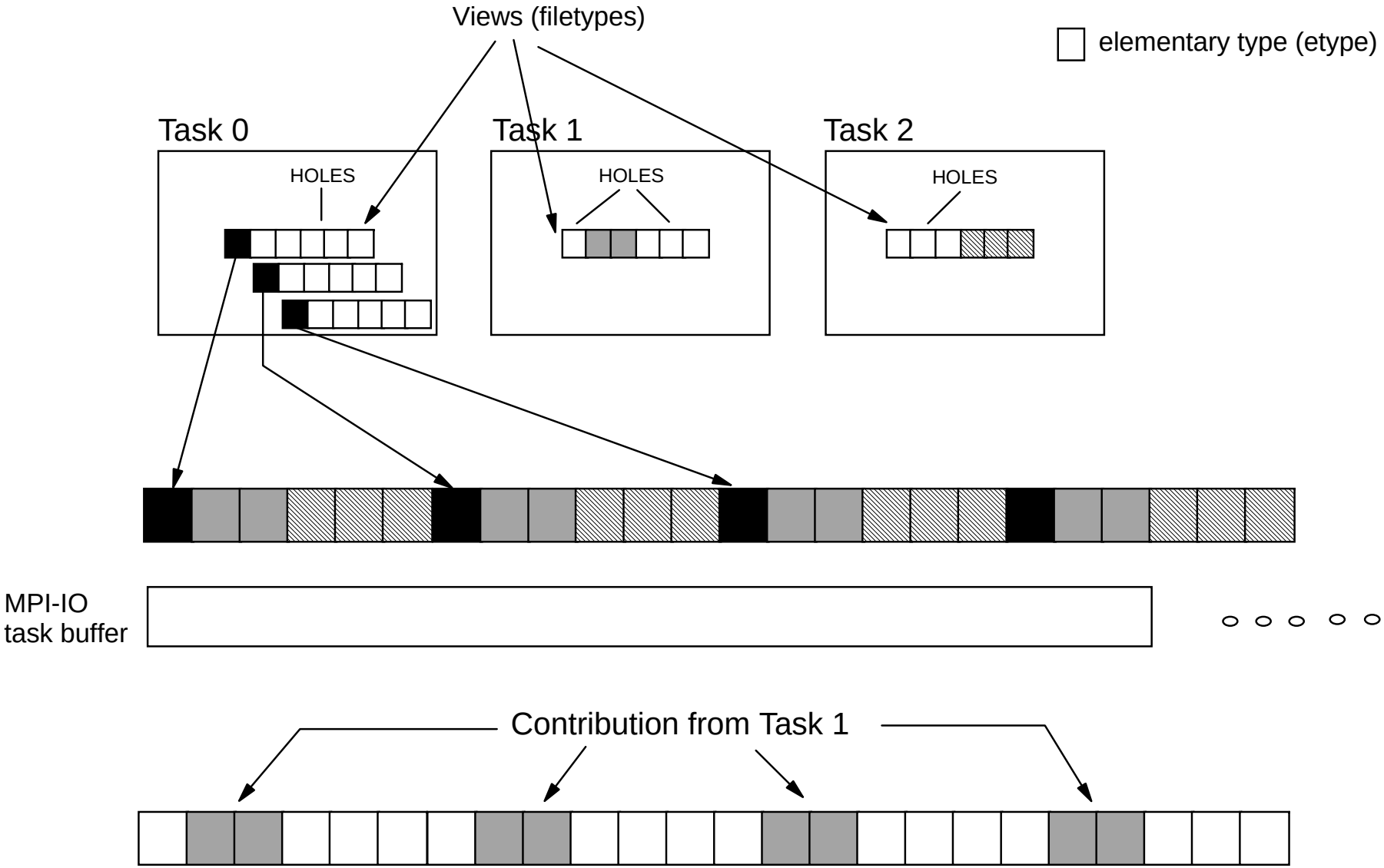
Improvements in MPI-IO

MPI-IO in Parallel Environment uses background threads (responders is our term) to enable tasks of the MPI job do file read/write and shiping of data under remote direction.

File I/O needs a thread to manage it but if we can move the data without paying for thread dispatch we can improve efficiency.

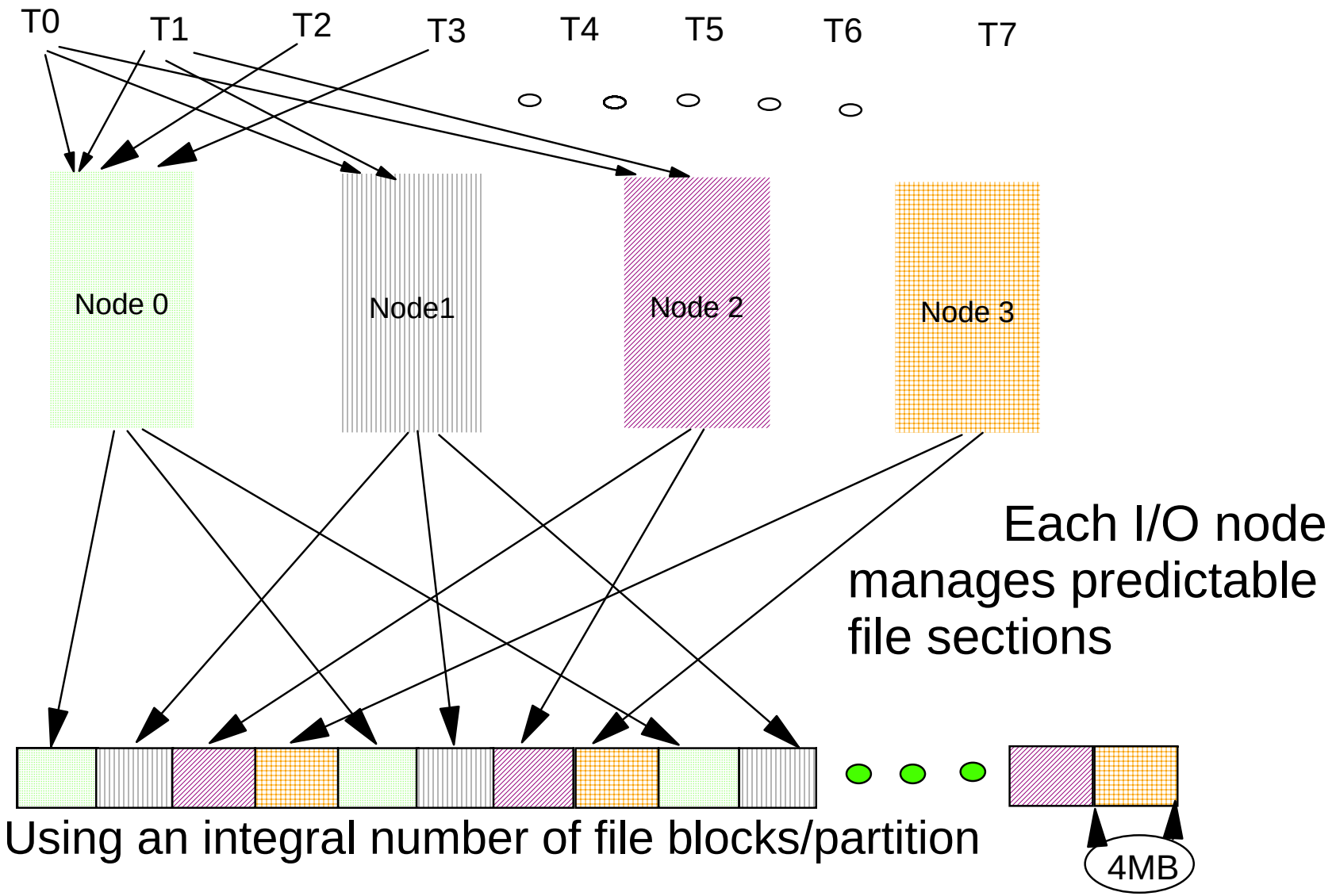
In collective I/O there are many data transfers linked to each file read or write step

Data Marshaling for Collective



Tasks deliver file data to I/O nodes

Partition of a file across I/O nodes



Moving the data with LAPI

LAPI_Putv today allows a task with data to deliver to another task to specify both a data source and data sink buffers as vectors

As a side benefit of doing MPI on LAPI, LAPI was enhanced with DGSP capability. LAPI_Putv now uses DGSP under cover and could be extended to use any MPI datatype

An enhanced LAPI_Put with DGSP can be used to deliver the data without thread activation costs

Environment Variables

Suggestions and Observations

A LAPI Glitch for small MPI_Sends

LAPI's behavior makes sense for a non-blocking API

MPCI copied small messages to a staging buffer and did the send from that. The MPI_Send call could return as soon as the data was in the staging buffer.

LAPI does send from the user buffer. The sent data needs to be preserved until all packet ACKs are in so return from MPI_Send must wait.

This delay in MPI_Send's return can impact applications. Keeping a clone of the data in case retransmit is needed so return can be prompt makes good sense.

The solutions - with environment variables

Changes coming:

MP_COPY_SEND_BUF_SIZE - being removed

Takes too much memory in large jobs.

MP_REXMIT_BUF_SIZE - being added

MP_REXMIT_BUF_CNT - being added

Buffers will now be shared. If no buffer is available when LAPI is doing a LAPI_Xfer (for MPI_Send) completion handler will wait for ACKs. If buffer available, data will be copied to the buffer and completion handler will be immediate

A couple more Environment Variables

MP_SINGLE_THREAD=YES

Before PE 4.1, non-threaded applications were often run with the non-threads library. That library has been dropped from PE so a non-threaded APP on the threads library should save locking costs with this variable

MP_EUIDEVICE=css0 or sn_single

Once LAPI supports one MPI task striping across multiple adapters, csss or sn_all will be reasonable options. Today, the settings can hurt performance if honored.

(LoadL keyword max_protocol_instances=1 is better)

MPI Task Affinity

It is desirable to have an MPI task, its memory pages and the adapter it uses all on the same MCM

The options for enforcing this at user level are awkward. Having pages allocated in MCM where process sits becomes a detriment if process moves.

Parallel Environment will be providing a means for the user to specify an affinity policy which PE and AIX will honor by keeping the process near its pages and adapter.

PE Co-scheduler coming soon

Several investigations and papers have looked at the impacts of random system events on synchronizing MPI operations.

- MPI collectives like MPI_BARRIER
- User written synchronizations like HALO exchange

Three major classes of events:

- OS needs to run its scheduler at each time slice
- System daemons run at seemingly random times
- Work that is not part of parallel jobs, if allowed on system has random impacts.

Most system administrators have long ago learned to keep asynchronous actions (interactive logins, non-vital daemons, file servers) off loaded compute nodes.

PE Co-scheduler coming soon

AIX time slice can be increased so OS events happen less often

By default, AIX does OS scheduling on each CPU round robin. Good for unsynched processes but not for MPI jobs. Better every task takes the hit at the same time. An AIX option will allow all CPUs to run scheduler simultaneously

Daemons can be managed by moving priority of all parallel tasks up and back down in a cycle that puts the parallel tasks on/off together, letting daemons have their turn

With PE Co-scheduler, it will be possible to apply both the AIX scheduler and priority cycling strategy to all nodes of a cluster and keep it in synch across the cluster.

HPS performance enhancements

The version of HPS/US that shipped in Fall 2003 has 2 modes and neither performs as we hoped and expected

- S/R FIFO or Packet mode is well short of the hoped for bandwidth
- S/R FIFO or Packet mode did not meet projected latencies

- Zero Copy comes closer to BW expectation but is still not ideal
- Zero Copy has high enough set up cost so the usefulness threshold is high (the message needs to be big for BW to pay back startup)

(Zero Copy does CPU offload and that is a big plus)

HPS performance enhancements

We are working on a redesign of the HPS firmware.

Version under development is not customer ready but is looking far better. (no numbers to share yet)

Packet mode has vastly better bandwidth

near the current zero copy bandwidth

Packet mode has significantly lower latency

HPS performance enhancements

Zero Copy is being replaced with an RDMA mode

- Looks so far like it will be much better BW than today's zero copy
- RDMA has lower setup overheads - means smaller messages might still amortize the setup cost
- RDMA may be less impacted by whether large pages are used
- RDMA might be appropriate even for small messages with some options we are considering for the future
- Like zero-copy, this will be a CPU offload approach

HPS performance enhancements

The new firmware for HPS and the protocol stack changes for changing from zero-copy to RDMA are on a fast track with a goal of delivery in the near future

LAPI Striping of Single Task Large Message

With the RDMA mode on HPS and multiple adapters per node, spreading a single task large message across 2 or more adapters becomes attractive.

Since RDMA is CPU offload, only startup and finish take CPU attention. One communication thread in one MPI (or LAPI) task should be able to set up multiple stripes that run concurrently

Striping usually not appropriate when MPI task count => than number of adapters.

LAPI Striping and MPI / OpenMP Hybrid

Int*ra* message striping is very attractive for master_only or funneled OpenMP/MPI mix with fewer MPI tasks than adapters.

- Larger messages by one task that result from having many OpenMP threads can make good use of internode bandwidth

Int*er* message striping could be attractive for the multiple OpenMP model where several threads might initiate messages concurrently.

RDMA based striping will not require extra CPUs for transfer but will only apply to contiguous messages

Considering use of Large Pages

I think you have heard enough about pros and cons of Large Pages. I will just say:

If the usage profile for your system is fairly uniform (a few critical apps run in a more or less standard way) there is some hope you can make a clear evaluation of using large pages.

If practical, have the things that work very badly with large pages run on non-large page nodes:

compiles, scripts, short run tools etc

Make large page usage a process by process option, not a default.

Contact Information

Richard Treumann

IBM Poughkeepsie Unix Development Lab

treumann@us.ibm.com