
A UT/ORNL PARTNERSHIP
NATIONAL INSTITUTE FOR COMPUTATIONAL SCIENCES
NICS

# HPC Communication

**R. Glenn Brook**

glenn-brook@tennessee.edu

**Scaling to Petascale and Beyond: Performance Analysis and Optimization of Applications**

**SC11 Tutorial - Seattle, WA – November 13, 2011**

# HPC Communication Considerations

- Communication Cost: $t_{comm} = t_s + L * t_h + m/B_e$
  - $t_s$ = startup time ~ time to setup message and routing
  - $t_h$ = hop time, L = number of hops
  - m = message size, $B_e$ = effective bandwidth
  - Communication is expensive. Avoid it if possible!
- Considerations:
  - Number and size of communications
  - Transmission methods and protocols
  - Network topology and routing
  - Congestion and synchronization
- Applies to message passing and shared access but this discussion focuses on message passing via MPI

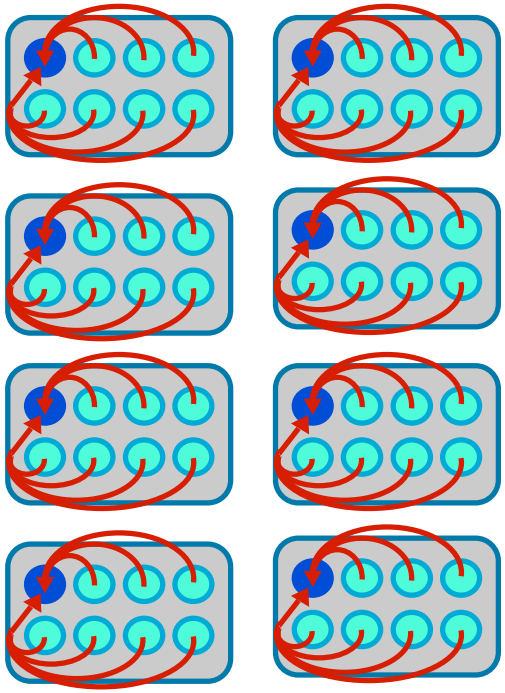
# Number and Size of Communications

- Communication Cost: $t_{comm} = t_s + L * t_h + m/B_e$
- Each communication incurs a minimum cost equal to the startup time $t_s$.
- The size of a communication determines the time required to transmit the associated data.
- In general, minimize both the number and size of communications.
  - Favor a minimum number over a minimum size due to the bandwidth of modern interconnects.
  - It is better to spend time transmitting useful data than to pay startup costs. Aggregate data when possible.

# Collective Communications

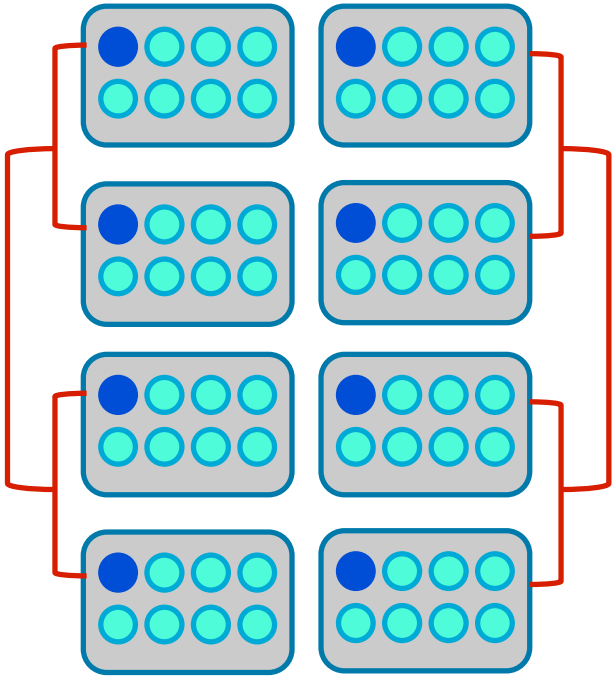
- Collectives typically require a large number of messages with potentially bad routing.
- Major source of problems when scaling due to rapidly increasing number of messages
- Most vendors provide optimized versions
- Optimized Collectives - Cray
  - MPI_Allgather (small messages) & MPI_Allgatherv
  - MPI_Alltoall (optimized exchange order)
  - MPI_Alltoallv / MPI_Alltoallw (windowing algorithm)
- Optimized SMP-aware Collectives - Cray
  - MPI_Allreduce, MPI_Barrier, MPI_Bcast, MPI_Reduce

# SMP-aware Collectives – Allreduce Example



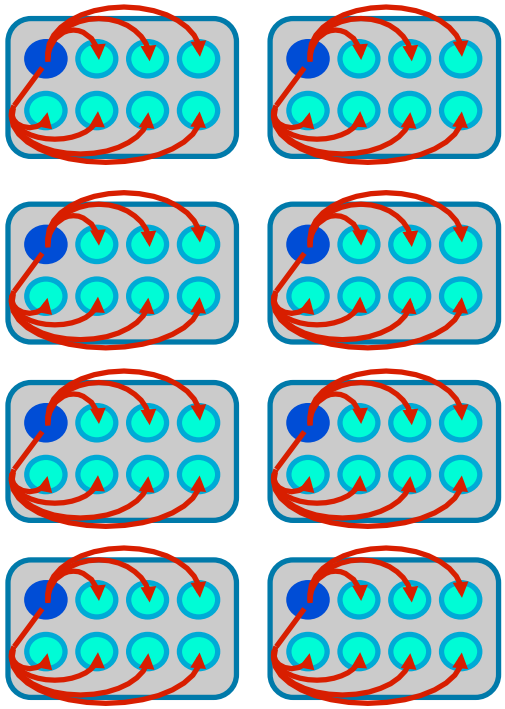
**STEP 1**

Identify Node-Captain rank. Perform a local on-node reduction to node-captain. NO network traffic.



**STEP 2**

Perform an Allreduce with node-captains only. This reduces the process count by a factor of 8 on XT5.

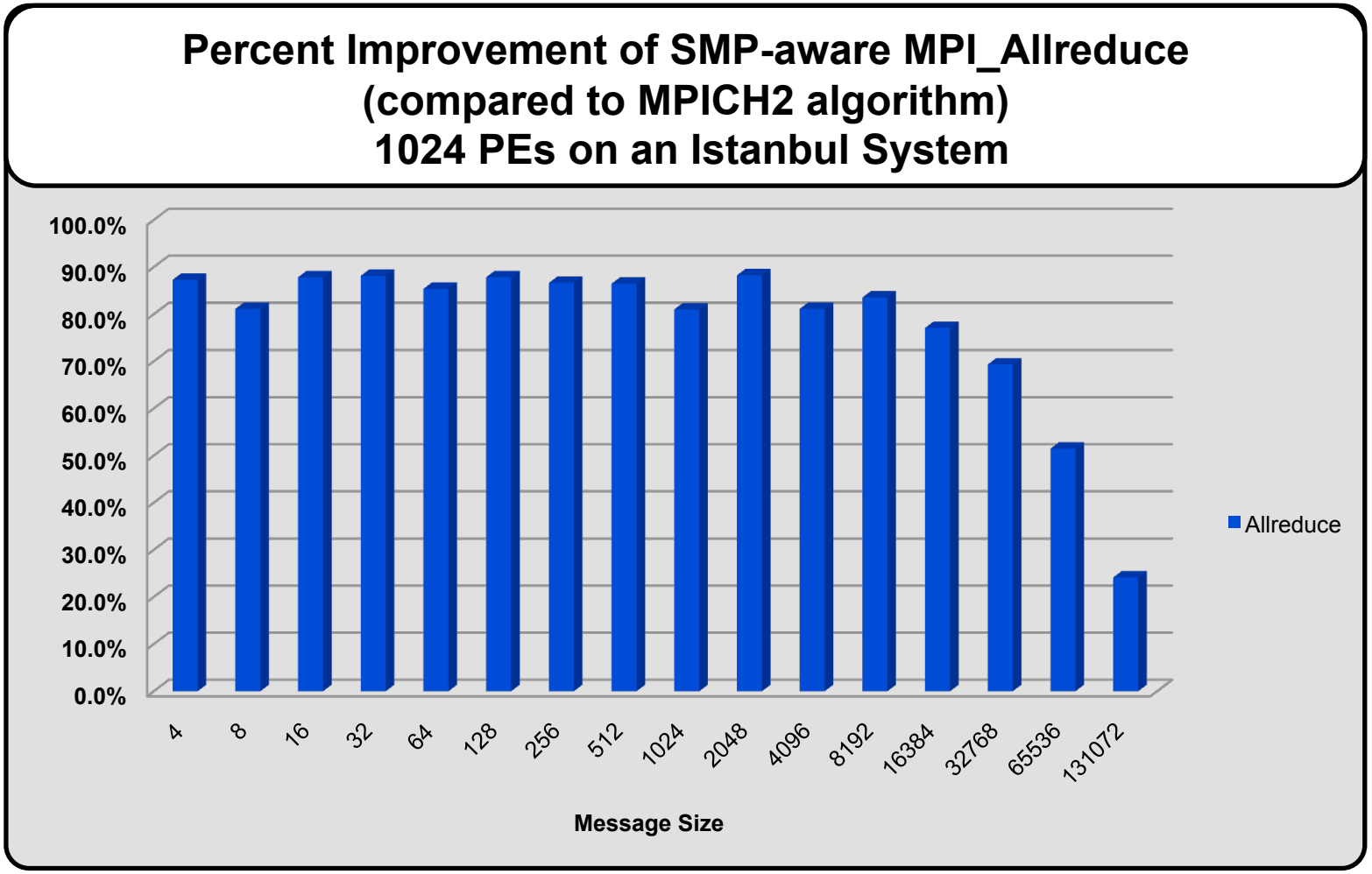


**STEP 3**

Perform a local on-node bcast. NO network traffic.

# Performance Comparison of MPI_Allreduce

Default vs MPICH_COLL_OPT_OFF=MPI_Allreduce



# Transmission Methods and Protocols

- Communication Cost: $t_{comm} = t_s + L * t_h + m/B_e$
- Multiple interconnect devices
  - SMP – Shared memory communication on nodes
  - Portals, IB, TCP/IP – Message passing between nodes
- Multiple message protocols
  - Short messages: eager protocol
  - Long messages: rendezvous protocol (default), eager protocol
- Automatic transitions between devices, protocols, and algorithms (configurable via environment variables)

# Short Message Eager Protocol - Cray

- Sending rank "pushes" message to receiving rank
  - Sender assumes receiver can handle message and blindly transmits to it
- If matching receive is posted, receiver
  - routes incoming data directly into specified receive buffer
  - posts notification event to other event queue
- If no matching receive is posted, receiver
  - routes incoming data into unexpected message buffer
  - posts two events to unexpected event queue
  - copies data into specified receive buffer when matching receive is posted
- Message size <= MPICH_MAX_SHORT_MSG_SIZE bytes

# Long Message Rendezvous Protocol - Cray

- Receiving rank "pulls" message from sending rank
- Sender notifies receiver about waiting message via a small header packet
- Receiver requests message from sender after matching receive is posted
- Receiver routes incoming data directly into specified receive buffer
- Message size > MPICH_MAX_SHORT_MSG_SIZE bytes

# Long Message Eager Protocol - Cray

- Sender assumes receiver will handle message appropriately or will request retransmission
  - Sender blindly transmits data to receiver
- If matching receive is posted, receiver
  - routes incoming data directly into specified receive buffer
  - sends completion acknowledgement to sender
- If no matching receive is posted, receiver
  - creates a long protocol match entry
  - requests retransmission when matching receive is posted
  - routes incoming data directly into specified receive buffer
- Enabled using MPICH_PTLS_EAGER_LONG
- CAUTION: blocking sends and unexpected messages

# Network Topology and Routing

- Communication Cost: $t_{comm} = t_s + L * t_h + m/B_e$
- The routing capabilities and physical layout of the network that interconnects compute nodes determines most of the transmission costs.
  - Number of hops between nodes
  - Time required per hop
  - Maximum bandwidth
- At small scale, the transmission cost associated with distance is often negligible, but such is not typically the case for very large machines.
- Design your communication schemes around the strengths and weaknesses of your target machine. The larger the machine, the more critical it is to do so.
  - For example, consider a hierarchical scheme on large fat trees to minimize the impact of long, high-latency paths.

# Rank Placement

- In some cases, changing how the processes are laid out on the machine may affect performance.
  - When point-to-point communication consumes a significant fraction of program time and a load imbalance detected
  - When using collectives (all-to-all) on subcommunicators (GYRO)
  - When you need to spread out IO across nodes (POP)
- Default is typically SMP-style placement where sequential MPI ranks are placed on the same node.
  - MPI codes typically perform better using SMP placement - nearest neighbor
  - Vendors typically optimize collectives to be SMP aware
- For example, a 12-process job launched on a node with 2 hex-core processors would be placed as:

| PROCESSOR | 0 | 1 |
|---|---|---|
| RANK | 0,1,2,3,4,5 | 6,7,8,9,10,11 |

# Rank Placement – Cray

- The default ordering can be changed using the following environment variable: MPICH_RANK_REORDER_METHOD
- These are the different values that you can set it to:
  - 0: Round-robin placement – Sequential ranks are placed on the next node in the list. Placement starts over with the first node upon reaching the end of the list.
  - 1: SMP-style placement – Sequential ranks fill up each node before moving to the next.
  - 2: Folded rank placement – Similar to round-robin placement except that each pass over the node list is in the opposite direction of the previous pass.
  - 3: Custom ordering. The ordering is specified in a file named MPICH_RANK_ORDER.
- The CrayPat performance measurement tools can generate a suggested custom ordering.

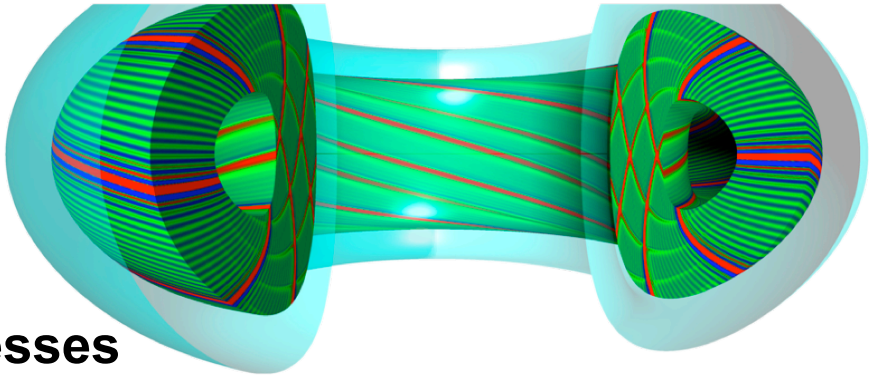
# Example of CrayPat Rank Reordering

```
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

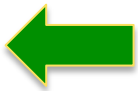
| Reorder method | Comm. time |
|---|---|
| Default | 11.26s |
| 0 – round-robin | 6.94s |
| 2 – folded-rank | 6.68s |
| d-custom from apa | 8.03s |

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

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



Huge win!

# Congestion and Synchronization

- Communication Cost: $t_{comm} = t_s + L * t_h + m/B_e$
- Excessive traffic through a network region reduces the effective bandwidth available to any given communication.
- Avoid routing a large number of messages to or from a region of adjacent nodes at the same time, when possible. Instead, stagger the communications.
- Avoid synchronous communications when possible, as load imbalances can lead to significant idling of resources.

# Tips & Recommendations

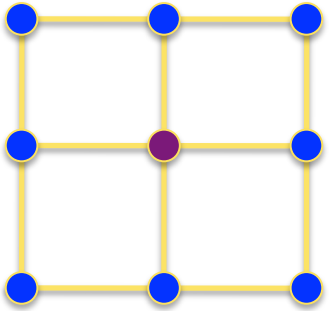
# MPI Environment Variables

- Many environment variables are available to tune MPI performance
  - Well documented on the MPI man page – Read it!
  - Default settings generally focus on attaining the best performance for most codes – not necessarily your code!
- The MPI environment can change between MPT versions - Cray
  - Read the MPI man page and Cray documentation!
- MPICH_ENV_DISPLAY – set to display the MPI environment during MPI initialization
- MPICH_VERSION_DISPLAY - set to display the version of Cray MPT during MPI initialization

# MPI Programming Techniques
# Pre-posting receives

- If possible, pre-post receives before the matching sends
  - Optimization technique for all MPICH installations (not just MPT)
  - Not sufficient to simply put receive immediately before send
  - Put significant amount of computation between receive-send pair
- Do not go crazy pre-posting receives. You can overrun the resources available to deal with them.
- Code example
  - Halo update – with four buffers (n,s,e,w), post all receive requests as early as possible.

# MPI Programming Techniques
# Example: 9-pt stencil pseudo-code



## Basic

9-pt computation

Update ghost cell boundaries

East/West IRECV, ISEND, WAITALL

North/South IRECV, ISEND, WAITALL

## Maximal Irecv preposting

Prepost all IRECV

9-pt computation

Update ghost cell boundaries

East/West ISEND, Wait on E/W IRECV only

North/South ISEND, Wait on the rest

*Makes use of temporary buffers

# MPI Programming Techniques Overlapping communication with computation

- Use non-blocking send/recvs to overlap communication with computation whenever possible
  - Typical pattern:
    1. Pre-post non-blocking receive
    2. Compute a “reasonable” amount to ensure effective pre-posting
    3. Post non-blocking send
    4. Compute as much as possible to maximize overlap of comm. and comp.
    5. Wait on communication to finish only when absolutely necessary

# MPI Programming Techniques Overlapping communication with computation

- In some cases, it may be better to replace collective operations with point-to-point communications to overlap communication with computation
  - Caution: Do not blindly reprogram every collective by hand
  - Concentrate on the parts of your algorithm with significant amounts of computation that can overlap with the point-to-point communications when a [blocking] collective is replaced

# MPI Programming Techniques Reduce Collective Communications

- Avoid using collective communications whenever possible
  - MPI collectives are blocking, leading to large sync times
  - Collective communication can cripple scalability
- Use algorithms that only require local info when possible
  - Consider duplicating computation to reduce communication
- When an algorithm must communicate “globally”:
  - Use MPT collectives that have been optimized by Cray
  - Minimize the scope of the collective operation
  - Minimize the number of collectives through aggregation
  - Consider implementing a non-blocking collective only if justified after careful analysis

# MPI Programming Techniques Aggregating data

- For very small buffers, aggregate data into fewer MPI calls (especially for collectives)
  - 1 all-to-all with an array of 3 reals is clearly better than 3 all-to-alls with 1 real
  - Do not aggregate too much without careful consideration. The MPI protocol switches from a short (eager) protocol to a long message protocol using a receiver pull method once the message is larger than the eager limit. The optimal size for messages most of the time is less than the eager limit.
- Example – DNS
  - Turbulence code (DNS) replaced 3 AllGatherv's by one with a larger message resulting in 25% less runtime for one routine

# MPI Programming Techniques
# Aggregating data: Example from CFD

***Original***

```
    for (index = 0; index < No; index++){
       double tmp;
       tmp = 0.0;
       out_area[index] = Bndry_Area_out(A,
    labels[index]);
       gdsum(&outlet_area[index],1,&tmp);
    }
    for (index = 0; index < Ni; index++){
      double tmp;
      tmp = 0.0;
      in_area[index] = Bndry_Area_in(A,
    labels[index]);
      gdsum(&inlet_area[index],1,&tmp);
    }


void gdsum (double *x, int n, double *work)
    {
      register int i;
      MPI_Allreduce (x, work, n, MPI_DOUBLE,
    MPI_SUM, MPI_COMM_WORLD);
      /* *x = *work; */
      dcopy(n,work,1,x,1);
      return;
    }
```

***Improved***

```
    for (index = 0; index < No; index++){
       out_area[index] = Bndry_Area_out(A,
labels[index]);
    }

    /* Get gdsum out of for loop */
    tmp = new double[No];
    gdsum (outlet_area, No, tmp);
    delete tmp;

    for (index = 0; index < Nin; index++){
      in_area[index] = Bndry_Area_in(A,
labels[index]);
    }

    /*  Get gdsum out of for loop */
    tmp = new double[Ni];
    gdsum(inlet_area, Ni, tmp);
    delete tmp;
```

# Hybridization

# OpenMP

- When does it pay to add/use OpenMP in my MPI code?
  - Add/use OpenMP when code is network bound
  - As collective and/or point-to-point time increasingly becomes a problem, use threading to keep number of MPI processes per node to a minimum
  - Be careful adding OpenMP to memory bound codes – can hurt performance
  - Be careful to match memory affinity to thread affinity
    - Pre-touch memory from correct thread after allocation
  - It is code/situation dependent!
  - Consider one MPI process on each CPU and one OpenMP thread per available core within each process
    - Often gives results almost as good as a fully optimized one-process-per-node code (with OpenMP threads across all of the cores on the node) with significantly less development overhead

# OpenMP
# aprun depth - Cray

- Must get “aprun –d” correct
  - -d (depth) Specifies the number of threads (cores) for each process. ALPS allocates the number of cores equal to depth times processes.
  - The default depth is 1. This option is used in conjunction with the OMP_NUM_THREADS environment variable.
  - Also used to get more memory per process
    - Get 1 or 2 GB limit by default (machine dependent)
  - Many have gotten this wrong, so it is important to understand how to use it properly!
    - If you do not do it correctly, a hybrid OpenMP/MPI code can get multiple threads spawned on the same core which can be disastrous.

# OpenMP aprun depth (cont.)

All on core 0
All on 1 node

One thread per core as desired!!!

```
% setenv OMP_NUM_THREADS 4
% aprun -n 4 -q ./omp1 | sort
Hello from rank 0, thread 0, on nid00291. (core affinity = 0)
Hello from rank 0, thread 1, on nid00291. (core affinity = 0)
Hello from rank 0, thread 2, on nid00291. (core affinity = 0)
Hello from rank 0, thread 3, on nid00291. (core affinity = 0)
Hello from rank 1, thread 0, on nid00291. (core affinity = 1)
Hello from rank 1, thread 1, on nid00291. (core affinity = 1)
Hello from rank 1, thread 2, on nid00291. (core affinity = 1)
Hello from rank 1, thread 3, on nid00291. (core affinity = 1)
Hello from rank 2, thread 0, on nid00291. (core affinity = 2)
Hello from rank 2, thread 1, on nid00291. (core affinity = 2)
Hello from rank 2, thread 2, on nid00291. (core affinity = 2)
Hello from rank 2, thread 3, on nid00291. (core affinity = 2)
Hello from rank 3, thread 0, on nid00291. (core affinity = 3)
Hello from rank 3, thread 1, on nid00291. (core affinity = 3)
Hello from rank 3, thread 2, on nid00291. (core affinity = 3)
Hello from rank 3, thread 3, on nid00291. (core affinity = 3)
```

```
% setenv OMP_NUM_THREADS 4
% aprun -n 4 -d 4 -q ./omp | sort
Hello from rank 0, thread 0, on nid00291. (core affinity = 0)
Hello from rank 0, thread 1, on nid00291. (core affinity = 1)
Hello from rank 0, thread 2, on nid00291. (core affinity = 2)
Hello from rank 0, thread 3, on nid00291. (core affinity = 3)
Hello from rank 1, thread 0, on nid00291. (core affinity = 4)
Hello from rank 1, thread 1, on nid00291. (core affinity = 5)
Hello from rank 1, thread 2, on nid00291. (core affinity = 6)
Hello from rank 1, thread 3, on nid00291. (core affinity = 7)
Hello from rank 2, thread 0, on nid00292. (core affinity = 0)
Hello from rank 2, thread 1, on nid00292. (core affinity = 1)
Hello from rank 2, thread 2, on nid00292. (core affinity = 2)
Hello from rank 2, thread 3, on nid00292. (core affinity = 3)
Hello from rank 3, thread 0, on nid00292. (core affinity = 4)
Hello from rank 3, thread 1, on nid00292. (core affinity = 5)
Hello from rank 3, thread 2, on nid00292. (core affinity = 6)
Hello from rank 3, thread 3, on nid00292. (core affinity = 7)
```

NICS

# Closing Remarks

# Input/Output

- Sometimes I/O causes scalability issues
  - For example, cleaning up some writes improved weak scaling of the CFD code NektarG from 70% to 95% at 1K to 8K cores
- Set file striping appropriately on Lustre
  - The default stripe count will may be suboptimal.
  - The default stripe size is usually fine.
  - Once a file is written, the striping information is set
    - Stripe input directories before staging data
    - Stripe output directories before writing data
  - Stripe for your I/O pattern
    - Many-many – narrow stripes  Many-one – wide stripes
- Reduce output to stdout
  - Remove debugging reports (e.g. “Hello from rank n of N”)

# Conclusions/Last words

- Vendors typically provide optimized, high-performance MPI implementations
  - Sometimes requires guidance and tuning – also patience and perseverance
- Environment variables are an easy way to improve performance
  - Familiarize yourself with 'man mpi' and remain up-to-date
- The is no replacement for good MPI programming
  - Pre-posting receives, overlap computation and communication, reduce collective communications, aggregate data for communication
- Rank reordering can significantly improve performance
- Use depth option to aprun with OpenMP on Cray systems
- Remember your parallel I/O – it can be crippling
- Some of this may not show a benefit at <1K processes, but it can reap huge gains at 10K to 100K processes
- Thanks to Jeff Larkin of Cray for permission to use his slides

# Contact Information

**R. Glenn Brook**
Director, Application Acceleration Center of Excellence
Manager, XSEDE User Engagement
National Institute for Computational Sciences
glenn-brook@tennessee.edu