MPI for Scalable Computing
(continued from yesterday)

Bill Gropp, University of Illinois at Urbana-Champaign
Rusty Lusk, Argonne National Laboratory
Rajeev Thakur, Argonne National Laboratory
Topology Mapping and Neighborhood Collectives
Topology Mapping Basics

- **First type: Allocation mapping**
  - Up-front specification of communication pattern
  - Batch system picks good set of nodes for given topology

- **Properties:**
  - Not widely supported by current batch systems
  - Either predefined allocation (BG/P), random allocation, or “global bandwidth maximization”
  - Also problematic to specify communication pattern upfront, not always possible (or static)
Topology Mapping Basics

- Rank reordering
  - Change numbering in a given allocation to reduce congestion or dilation
  - Sometimes automatic (early IBM SP machines)

- Properties
  - Always possible, but effect may be limited (e.g., in a bad allocation)
  - Portable way: MPI process topologies
    - Network topology is not exposed
  - Manual data shuffling after remapping step
On-Node Reordering

Naïve Mapping

Optimized Mapping

Gottschling and Hoefler: Productive Parallel Linear Algebra Programming with Unstructured Topology Adaption
Off-Node (Network) Reordering

Application Topology

Naïve Mapping

Optimal Mapping
MPI Topology Intro

- **Convenience functions (in MPI-1)**
  - Create a graph and query it, nothing else
  - Useful especially for Cartesian topologies
    - Query neighbors in n-dimensional space
  - Graph topology: each rank specifies full graph 😞

- **Scalable Graph topology (MPI-2.2)**
  - Graph topology: each rank specifies its neighbors or an arbitrary subset of the graph

- **Neighborhood collectives (MPI-3.0)**
  - Adding communication functions defined on graph topologies (neighborhood of distance one)
MPI_Cart_create

- Specify ndims-dimensional topology
  - Optionally periodic in each dimension (Torus)
- Some processes may return MPI_COMM_NULL
  - Product of dims must be \( \leq P \)
- Reorder argument allows for topology mapping
  - Each calling process may have a new rank in the created communicator
  - Data has to be remapped manually
MPI_Cart_create Example

- But we’re starting MPI processes with a one-dimensional argument (-p X)
  - User has to determine size of each dimension
  - Often as “square” as possible, MPI can help!

```c
int dims[3] = {5,5,5};
int periods[3] = {1,1,1};
MPI_Comm topocomm;
MPI_Cart_create(comm, 3, dims, periods, 0, &topocomm);
```
MPI_Dims_create

- Create dims array for Cart_create with nnodes and ndims
  - Dimensions are as close as possible (well, in theory)
- Non-zero entries in dims will not be changed
  - nnodes must be multiple of all non-zeroes in dims
MPI_Dims_create Example

```
int p;
int dims[3] = {0,0,0};
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Dims_create(p, 3, dims);

int periods[3] = {1,1,1};
MPI_Comm topocomm;
MPI_Cart_create(comm, 3, dims, periods, 0, &topocomm);
```

- Makes life a little bit easier
  - Some problems may be better with a non-square layout though
Cartesian Query Functions

- **Library support and convenience!**
- **MPI_Cartdim_get()**
  - Gets dimensions of a Cartesian communicator
- **MPI_Cart_get()**
  - Gets size of dimensions
- **MPI_Cart_rank()**
  - Translate coordinates to rank
- **MPI_Cart_coords()**
  - Translate rank to coordinates
MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)

- Shift in one dimension
  - Dimensions are numbered from 0 to ndims-1
  - Displacement indicates neighbor distance (-1, 1, ...)
  - May return MPI_PROC_NULL
- Very convenient, all you need for nearest neighbor communication
Don’t use! Use one of the Dist_graph functions instead

\[
\text{MPI\_Graph\_create}(\text{MPI\_Comm}\ comm\_old, \text{int}\ nnodes, 
\text{const}\ \text{int}\ *\text{index}, \text{const}\ \text{int}\ *\text{edges}, \text{int}\ \text{reorder}, 
\text{MPI\_Comm}\ *\text{comm}\_graph)
\]

- nnodes is the total number of nodes
- index \(i\) stores the total number of neighbors for the first \(i\) nodes (sum)
  - Acts as offset into \text{edges} array
- edges stores the edge list for all processes
  - Edge list for process \(j\) starts at \text{index}[j] in \text{edges}
  - Process \(j\) has \text{index}[j+1]-\text{index}[j] edges
Distributed graph constructor

- MPI_Graph_create is discouraged
  - Not scalable
  - Not deprecated yet but hopefully soon

- New distributed interface:
  - Scalable, allows distributed graph specification
    - Either local neighbors or any edge in the graph
  - Specify edge weights
    - Meaning undefined but optimization opportunity for vendors!
  - Info arguments
    - Communicate assertions of semantics to the MPI library
    - E.g., semantics of edge weights
MPI_Dist_graph_create_adjacent

- indegree, sources, sourceweights – source proc. spec.
- outdegree, destinations, destweights – dest. proc. spec.
- info, reorder, comm_dist_graph – as usual
- directed graph
- Each edge is specified twice, once as out-edge (at the source) and once as in-edge (at the dest)
MPI_Dist_graph_create_adjacent

- Process 0:
  - Indegree: 0
  - Outdegree: 2
  - Dests: \{3,1\}

- Process 1:
  - Indegree: 3
  - Outdegree: 2
  - Sources: \{4,0,2\}
  - Dests: \{3,4\}

...
MPI_Dist_graph_create

- n – number of source nodes
- sources – n source nodes
- degrees – number of edges for each source
- destinations, weights – dest. process specification
- info, reorder – as usual
- More flexible and convenient
  - Requires global communication
  - Slightly more expensive than adjacent specification

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2
MPI_Dist_graph_create

- Process 0:
  - N: 2
  - Sources: \{0,1\}
  - Degrees: \{2,2\}
  - Dests: \{3,1,4,3\}

- Process 1:
  - N: 2
  - Sources: \{2,3\}
  - Degrees: \{1,1\} *
  - Dests: \{1,2\}

* Note that in this example, process 1 specifies only one of the two outgoing edges of process 3; the second outgoing edge needs to be specified by another process.
Distributed Graph Neighbor Queries

- **MPI_Dist_graph_neighbors_count()**
  
  ```c
  MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree, int *outdegree, int *weighted)
  ```
  
  - Query the number of neighbors of **calling process**
  - Returns indegree and outdegree!
  - Also info if weighted

- **MPI_Dist_graph_neighbors()**
  
  ```c
  MPI_Dist_graph_neighbors(MPI_Comm comm,
  int maxindegree, int sources[], int sourceweights[], int maxoutdegree,
  int destinations[], int destweights[])
  ```
  
  - Query the neighbor list of **calling process**
  - Optionally return weights
Further Graph Queries

- Status is either:
  - MPI_GRAPH
  - MPI_CART
  - MPI_DIST_GRAPH
  - MPI_UNDEFINED (no topology)

- Enables to write libraries on top of MPI topologies!

MPI_Topo_test(MPI_Comm comm, int *status)
Algorithms and Topology

- Complex hierarchy:
  - Multiple chips per node; different access to local memory and to interconnect; multiple cores per chip
  - Mesh has different bandwidths in different directions
  - Allocation of nodes may not be regular (you are unlikely to get a compact brick of nodes)
  - Some nodes have GPUs
- Most algorithms designed for simple hierarchies and ignore network issues

Recent work on general topology mapping e.g.,
Generic Topology Mapping Strategies for Large-scale Parallel Architectures, Hoefler and Snir
Dynamic Workloads Require New, More Integrated Approaches

- Performance irregularities mean that classic approaches to decomposition are increasingly ineffective
  - Irregularities come from OS, runtime, process/thread placement, memory, heterogeneous nodes, power/clock frequency management

- Static partitioning tools can lead to persistent load imbalances
  - Mesh partitioners have incorrect cost models, no feedback mechanism
  - “Regrid when things get bad” won’t work if the cost model is incorrect; also costly

- Basic building blocks must be more dynamic without introducing too much overhead
Communication Cost Includes More than Latency and Bandwidth

- Communication does not happen in isolation
- Effective bandwidth on shared link is \( \frac{1}{2} \) point-to-point bandwidth
- Real patterns can involve many more (integer factors)
- Loosely synchronous algorithms ensure communication cost is worst case
Halo Exchange on BG/Q and Cray XE6

- 2048 doubles to each neighbor
- Rate is MB/sec (for all tables)

<table>
<thead>
<tr>
<th>BG/Q</th>
<th>8 Neighbors</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Irecv/Send</td>
<td>Isend</td>
</tr>
<tr>
<td>World</td>
<td>662</td>
<td>1167</td>
</tr>
<tr>
<td>Even/Odd</td>
<td>711</td>
<td>1452</td>
</tr>
<tr>
<td>1 sender</td>
<td></td>
<td>2873</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cray XE6</th>
<th>8 Neighbors</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Irecv/Send</td>
<td>Isend</td>
</tr>
<tr>
<td>World</td>
<td>352</td>
<td>348</td>
</tr>
<tr>
<td>Even/Odd</td>
<td>338</td>
<td>324</td>
</tr>
<tr>
<td>1 sender</td>
<td></td>
<td>5507</td>
</tr>
</tbody>
</table>
Discovering Performance Opportunities

- Lets look at a single process sending to its neighbors.
- Based on our performance model, we *expect* the rate to be roughly twice that for the halo (since this test is only sending, not sending and receiving).

<table>
<thead>
<tr>
<th>System</th>
<th>4 neighbors</th>
<th>8 Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>BG/L</td>
<td>488</td>
<td>490</td>
</tr>
<tr>
<td></td>
<td>Periodic</td>
<td>389</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Periodic</td>
</tr>
<tr>
<td>BG/P</td>
<td>1139</td>
<td>1136</td>
</tr>
<tr>
<td></td>
<td></td>
<td>892</td>
</tr>
<tr>
<td></td>
<td></td>
<td>892</td>
</tr>
<tr>
<td>BG/Q</td>
<td></td>
<td>2873</td>
</tr>
<tr>
<td>XT3</td>
<td>1005</td>
<td>1007</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1053</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1045</td>
</tr>
<tr>
<td>XT4</td>
<td>1634</td>
<td>1620</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1773</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1770</td>
</tr>
<tr>
<td>XE6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5507</td>
</tr>
</tbody>
</table>
Discovering Performance Opportunities

- Ratios of a single sender to all processes sending (in rate)
- *Expect* a factor of roughly 2 (since processes must also receive)

<table>
<thead>
<tr>
<th>System</th>
<th>4 neighbors</th>
<th>8 Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Periodic</td>
<td>Periodic</td>
</tr>
<tr>
<td>BG/L</td>
<td>2.24</td>
<td>2.01</td>
</tr>
<tr>
<td>BG/P</td>
<td>3.8</td>
<td>2.2</td>
</tr>
<tr>
<td>BG/Q</td>
<td>1.98</td>
<td></td>
</tr>
<tr>
<td>XT3</td>
<td>7.5</td>
<td>8.1</td>
</tr>
<tr>
<td>XT4</td>
<td>10.7</td>
<td>10.7</td>
</tr>
<tr>
<td>XE6</td>
<td></td>
<td>15.6</td>
</tr>
</tbody>
</table>

- BG gives roughly double the halo rate. XTn and XE6 are much higher.
- It should be possible to improve the halo exchange on the XT by scheduling the communication
- Or improving the MPI implementation
Neighborhood Collectives
Neighborhood Collectives

- Topologies implement no communication!
  - Just helper functions

- Collective communications only cover some patterns
  - E.g., no stencil pattern

- Several requests for “build your own collective” functionality in MPI
  - Neighborhood collectives are a simplified version
  - Cf. Datatypes for communication patterns!
Cartesian Neighborhood Collectives

- Communicate with direct neighbors in Cartesian topology
  - Corresponds to cart_shift with disp=1
  - Collective (all processes in comm must call it, including processes without neighbors)
- Buffers are laid out as neighbor sequence:
  - Defined by order of dimensions, first negative, then positive
  - 2*ndims sources and destinations
  - Processes at borders (MPI_PROC_NULL) leave holes in buffers (will not be updated or communicated)!
Cartesian Neighborhood Collectives

- Allgather
- Buffer ordering example:
Graph Neighborhood Collectives

- Collective Communication along arbitrary neighborhoods
  - Order is determined by order of neighbors as returned by `(dist_)graph_neighbors`.
  - Distributed graph is directed, may have different numbers of send/recv neighbors.
  - Can express dense collective operations 😊
  - Any persistent communication pattern!
MPI Neighbor allgather

- Sends the same message to all neighbors
- Receives indegree distinct messages
- Similar to MPI_Gather
  - The all prefix expresses that each process is a “root” of his neighborhood
- Also a vector “v” version for full flexibility
MPI_Neighbor_alltoall

- Sends outdegree distinct messages
- Received indegree distinct messages
- Similar to MPI_Alltoall
  - Neighborhood specifies full communication relationship
- Vector and w versions for full flexibility
Nonblocking Neighborhood Collectives

- Very similar to nonblocking collectives
- Collective invocation
- Matching in-order (no tags)
  - No wild tricks with neighborhoods! In order matching per communicator!

MPI_Ineighbor_allgather(..., MPI_Request *req);
MPI_Ineighbor_alltoall(..., MPI_Request *req);
Topology Summary

- Topology functions allow users to specify application communication patterns/topology
  - Convenience functions (e.g., Cartesian)
  - Storing neighborhood relations (Graph)
- Enables topology mapping (reorder=1)
  - Not widely implemented yet
  - May requires manual data re-distribution (according to new rank order)
- MPI does not expose information about the network topology (would be very complex)
Neighborhood Collectives Summary

- Neighborhood collectives add communication functions to process topologies
  - Collective optimization potential!

- Allgather
  - One item to all neighbors

- Alltoall
  - Personalized item to each neighbor

- High optimization potential (similar to collective operations)
  - Interface encourages use of topology mapping!
Thanks to Torsten Hoefler and Pavan Balaji for some of the slides in this tutorial