Lecture 6: Performance Modeling/ Advanced MPI: Collective Communication

Outline

- Performance modeling

- MPI Recap
- Advanced MPI: Collective Communication

Performance Modeling

Shared Memory Performance Models

- Parallel Random Access Memory (PRAM)
- All memory access operations complete in one clock period -- no concept of memory hierarchy ("too good to be true").
- Ignores costs of synchronization, communication

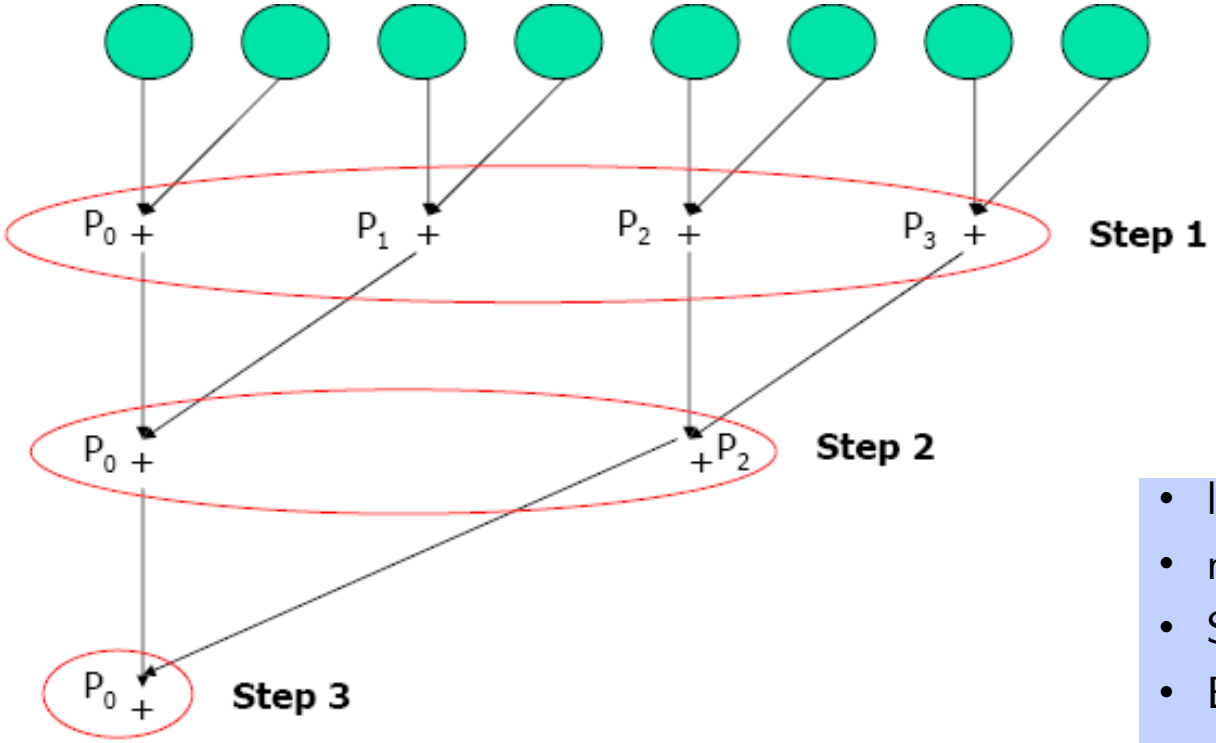
- Other assumptions
  - There is no limit on the number of processors in the machine.
  - Any memory location is uniformly accessible from any processor.
  - There is no limit on the amount of shared memory in the system.
  - Resource contention is absent.
  - The programs written on these machines are, in general, of type SIMD.

PRAM Complexity Measures

- for each individual processor
  - ***time***: number of instructions executed
  - ***space***: number of memory cells accessed

- PRAM machine
  - ***time***: time taken by the longest running processor
  - ***hardware***: maximum number of active processors

Example: Parallel Addition



- log(n) steps=time needed
- n/2 processors needed
- Speed-up = n/log(n)
- Efficiency = 2/log(n)
- Applicable for other operations too
  +, *, <, >, == etc.

Shared Memory Access Variations

- Different variations:
  - Exclusive Read Exclusive Write (EREW) PRAM: no two processors are allowed to read or write the same shared memory cell simultaneously
  - Concurrent Read Exclusive Write (CREW): simultaneous read allowed, but only one processor can write
  - Concurrent Read Concurrent Write (CRCW)
- Concurrent writes:
  - Priority CRCW: processors assigned fixed distinct priorities, highest priority wins
  - Arbitrary CRCW: one randomly chosen write wins
  - Common CRCW: all processors are allowed to complete write if and only if all the values to be written are equal

PRAM Model Flaws

- OK for understanding whether an algorithm has enough parallelism at all
  - But often too simple to give practical guidance.
- Parallel algorithm design strategy: first do a PRAM algorithm, then worry about memory/communication time (sometimes works)
- Some variations are more realistic
  - E.g., Concurrent Read Exclusive Write (CREW) PRAM.
  - Still missing the memory hierarchy

BSP: Bulk Synchronous Parallel

- Leslie Valiant and Bill McColl, 1990s
- Differs from PRAM in that it accounts for costs of synchronization and communication
- Still being developed; In 2017, McColl developed a extension that provides fault tolerance and tail tolerance for large-scale parallel computations in AI, Analytics and HPC

BSP Computer

A BSP computer consists of

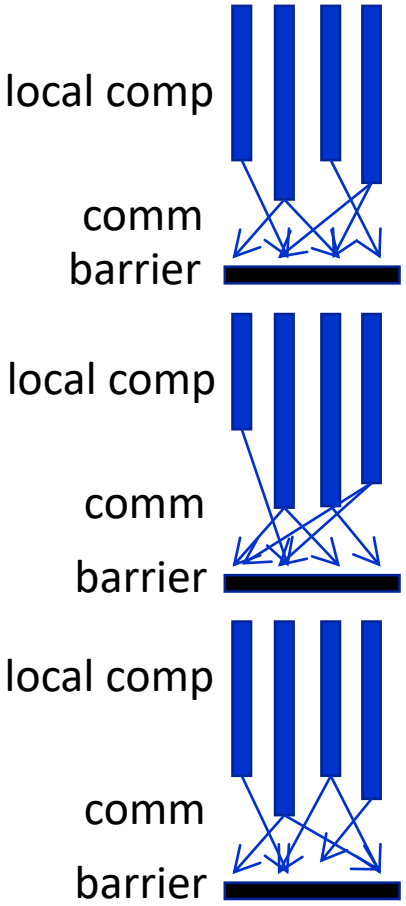
1. components capable of processing and/or local memory transactions (i.e., processors),
2. a network that routes messages between pairs of such components, and
3. a hardware facility that allows for the synchronization of all or a subset of components.

Commonly interpreted as a set of processors which may follow different threads of computation, with each processor equipped with fast local memory and interconnected by a communication network

BSP Computer

Computation proceeds in a series of global *supersteps*, which consists of three components:

1. *Concurrent computation*: every participating processor may perform local computations, i.e., each process can only make use of values stored in the local fast memory of the processor. The computations occur asynchronously of all the others but may overlap with communication.
2. *Communication*: The processes exchange data between themselves to facilitate remote data storage capabilities.
3. *Barrier synchronization*: When a process reaches this point (the *barrier*), it waits until all other processes have reached the same barrier.



Communication in BSP

- BSP model considers communication actions *en masse*
  - gives an upper bound on the time taken to communicate a set of data
- Considers all communication actions of a superstep as one unit, and assumes all individual messages sent as part of this unit have a fixed size.

Communication in BSP

- Parameters:
  - h: max # of incoming or outgoing messages for any process
  - g: single-word delivery time under continuous traffic conditions, defined such that it takes time hg for a processor to deliver h messages of size 1; determined empirically for each parallel computer
  - m: length of message
  - w: max cost of local computation for any process
  - L: cost of a barrier synchronization

Cost of a superstep:

$$w + hg + L$$

Cost of algorithm (sum of S supersteps):

$$W + Hg + SL = \sum_{s=1}^{S} w_s + g \sum_{s=1}^{S} h_s + SL$$

Further Reading

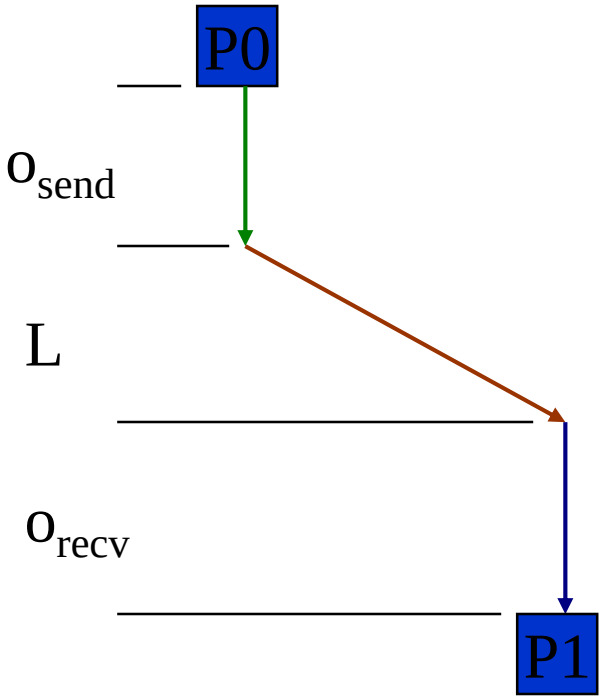
- Sections 1-3 of Scalable Computing, McColl: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.5671&rep=rep1&type=pdf
- McColl, Tiskin. Memory-Efficient Matrix Multiplication in the BSP Model https://link.springer.com/article/10.1007%2FPL00008264

LogP

- Culler, Karp, Patterson, 1993
- Aims at being more practical than PRAM model
- LogP machine consists of arbitrarily many processing units with distributed memory
- Processing units connected through abstract communication medium which allows point to point communication
- name comes from the 4 parameters: L, o, g, and P:
  - L, the latency of the communication medium.
  - o, the overhead of sending and receiving a message.
  - g, the gap required between two send/receive operations. A more common interpretation of this quantity is as the inverse of the bandwidth of a processor-processor communication channel.
  - P, the number of processing units. Each local operation on each machine takes the same time ('unit time'). This time is called a processor cycle.

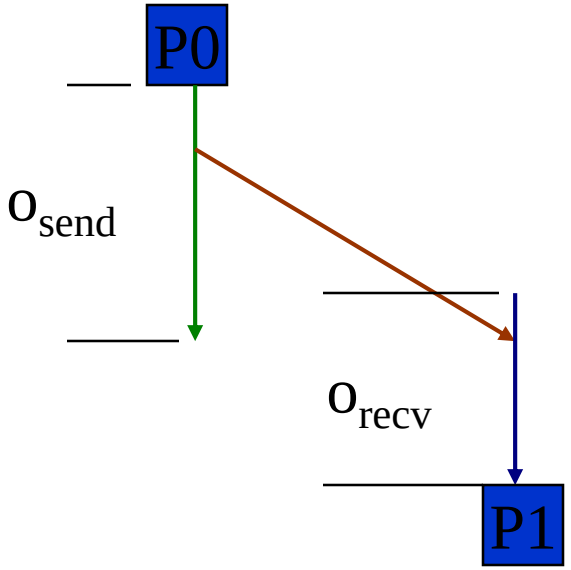
LogP Parameters: Overhead & Latency

- Non-overlapping overhead



EEL = End-to-End Latency
$= o_{send} + L + o_{recv}$

- Send and recv overhead can overlap

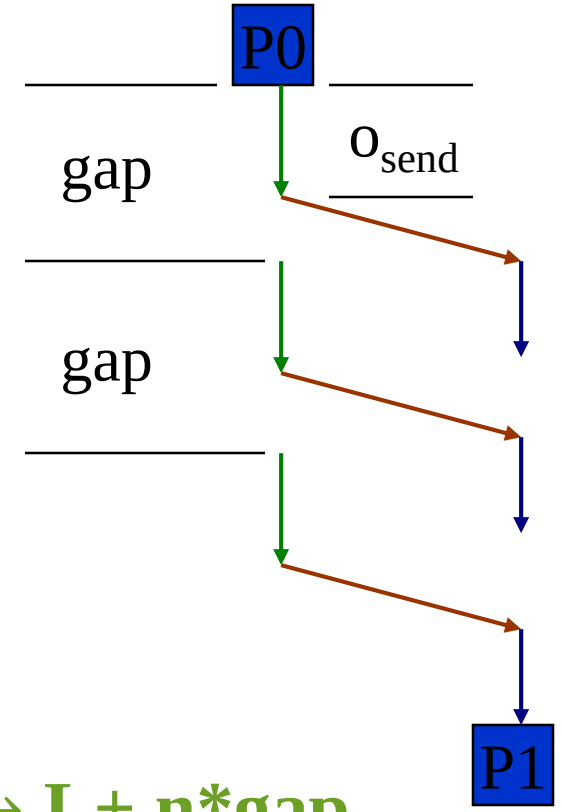


$EEL = f(o_{send}, L, o_{recv})$
$\geq \max(o_{send}, L, o_{recv})$

LogP Parameters: gap

- The Gap is the delay between sending messages
- Gap could be greater than send overhead
  - NIC may be busy finishing the processing of last message and cannot accept a new one.
  - Flow control or backpressure on the network may prevent the NIC from accepting the next message to send.
- No overlap ⇒

  time to send n messages (pipelined) =



$$(o_{send} + L + o_{recv} - gap) + n*gap \rightarrow L + n*gap$$

Limitations of the LogP Model

- The LogP model has a fixed cost for each message
  - This is useful in showing how to quickly broadcast a single word
  - Other examples also in the LogP papers
- For larger messages, there is a variation LogGP
  - Two gap parameters, one for small and one for large messages
- No topology considerations (including no limits for bisection bandwidth)
  - Assumes a fully connected network
  - OK for some algorithms with nearest neighbor communication, but with "all-to-all" communication we need to refine this further
- This is a flat model, i.e., each processor is connected to the network
  - Clusters of multicores are not accurately modeled

Latency and Bandwidth Model

- Time to send message of length n is roughly

$$\begin{aligned} \text{Time} &= \text{latency} + \text{n*cost\_per\_word} \\ &= \text{latency} + \text{n/bandwidth} \end{aligned}$$

- Topology is assumed irrelevant.
- Often called "α–β model" and written

$$\text{Time} = \alpha + \text{n}*\beta$$

- Usually $\alpha >> \beta >>$ time per flop.
  - One big message is cheaper than many small ones.

$$\alpha + \text{n}*\beta << \text{n}*(\alpha + 1*\beta)$$

  - Can do hundreds or thousands of flops for cost of one message.
- Lesson: Need large computation-to-communication ratio to be efficient.
- [Chan, Heimlich, Purkayastha, van de Geijn, 2007].

The $\alpha - \beta - \gamma$ model

- Let γ be the cost of a floating point operation (seconds per flop)
- Let α be the latency cost of a message (seconds)
- Let β be the inverse bandwidth cost (seconds/word)

- The cost (in seconds) of a computation that performs F flops and sends S messages consisting of W words (along the critical path) is

$$\gamma F + \alpha S + \beta W$$

- If we can overlap computation and communication,

$$\max(\gamma F, \alpha S + \beta W)$$

- Note: you can have model of sequential/shared memory computation (where α, β correspond to the cost of moving data throughout memory hierarchy) and also model of distributed computation (where α, β correspond to cost of moving data over the interconnect network)
  - Models can be composed!

Advanced MPI

Review: Message Passing Libraries

- All communication, synchronization require subroutine calls
  - No shared variables
  - Program runs on a single processor just like any uniprocessor program, except for calls to message passing library
- Subroutines for
  - Communication
    - Pairwise or point-to-point: Send and Receive
    - Collectives all processor get together to
      - Move data: Broadcast, Scatter/gather
      - Compute and move: sum, product, max, prefix sum, ... of data on many processors
  - Synchronization
    - Barrier
    - No locks because there are no shared variables to protect
  - Enquiries
    - How many processes? Which one am I? Any messages waiting?

Review: MPI Concepts

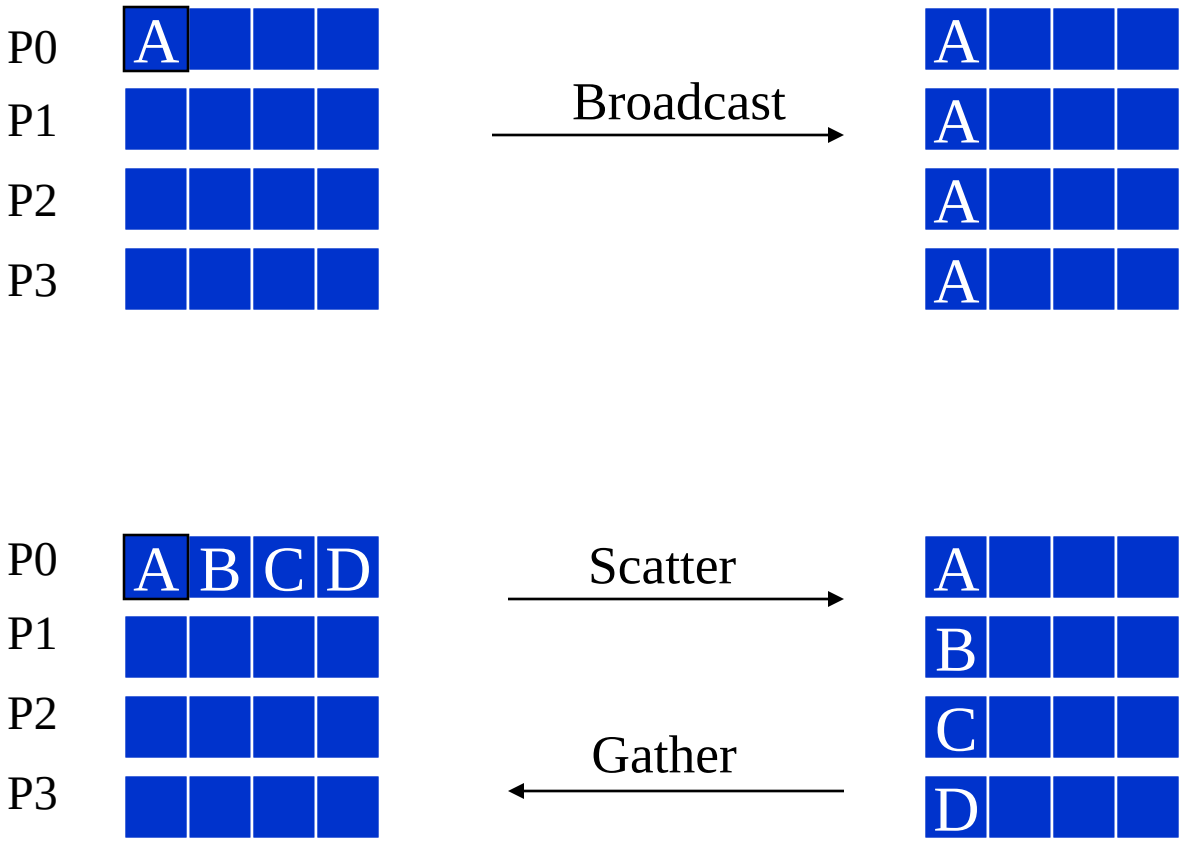
- Communicator
  - Communicator objects connect groups of processes in the MPI session.
  - Each communicator gives each contained process an independent identifier and arranges its contained processes in an ordered topology
  - Default: MPI_COMM_WORLD
- Point-to-point communication
  - Sends and receives between individual processes
  - Both blocking and nonblocking types

Another Approach to Parallelism

- *Collective* routines provide a higher-level way to organize a parallel program
- Each process executes the same communication operations
- MPI provides a rich set of collective operations...

- Also non-blocking collective operations.

Collective Data Movement

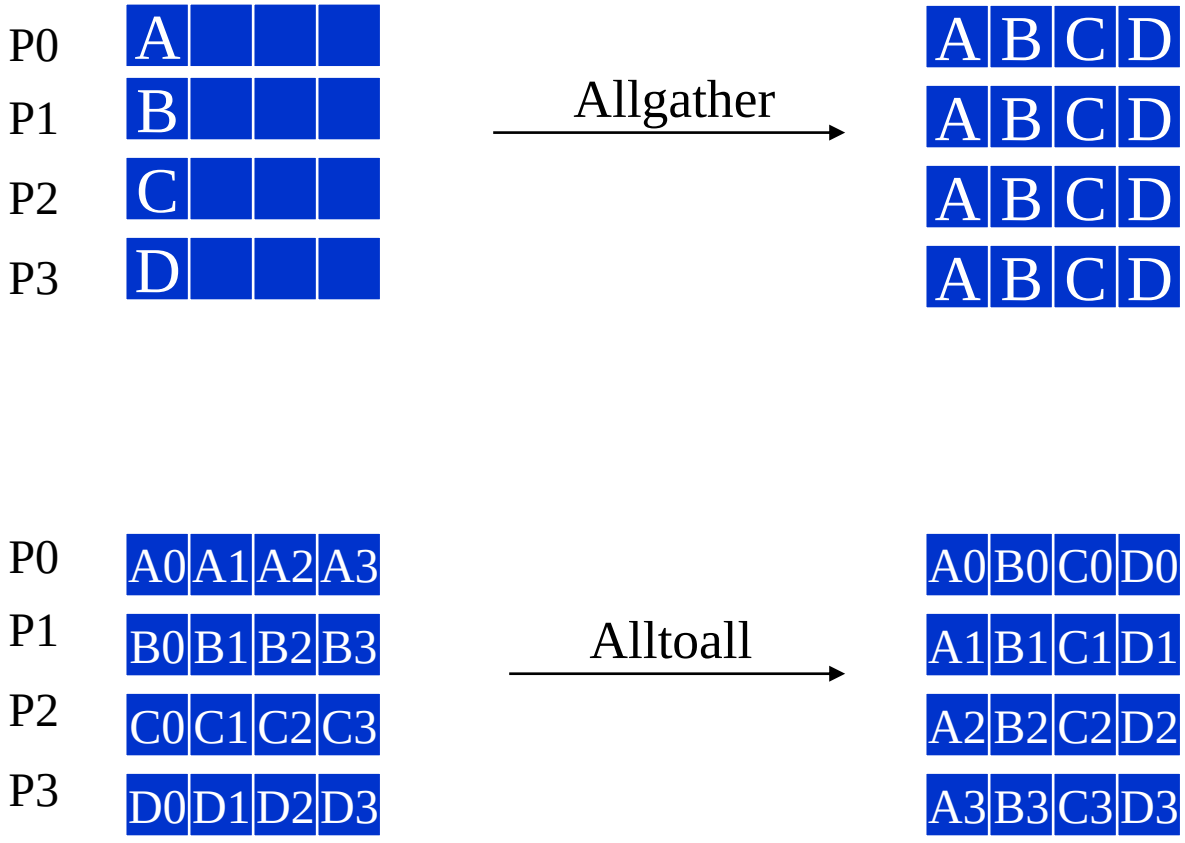


Practical Comment on Broadcast, other Collectives

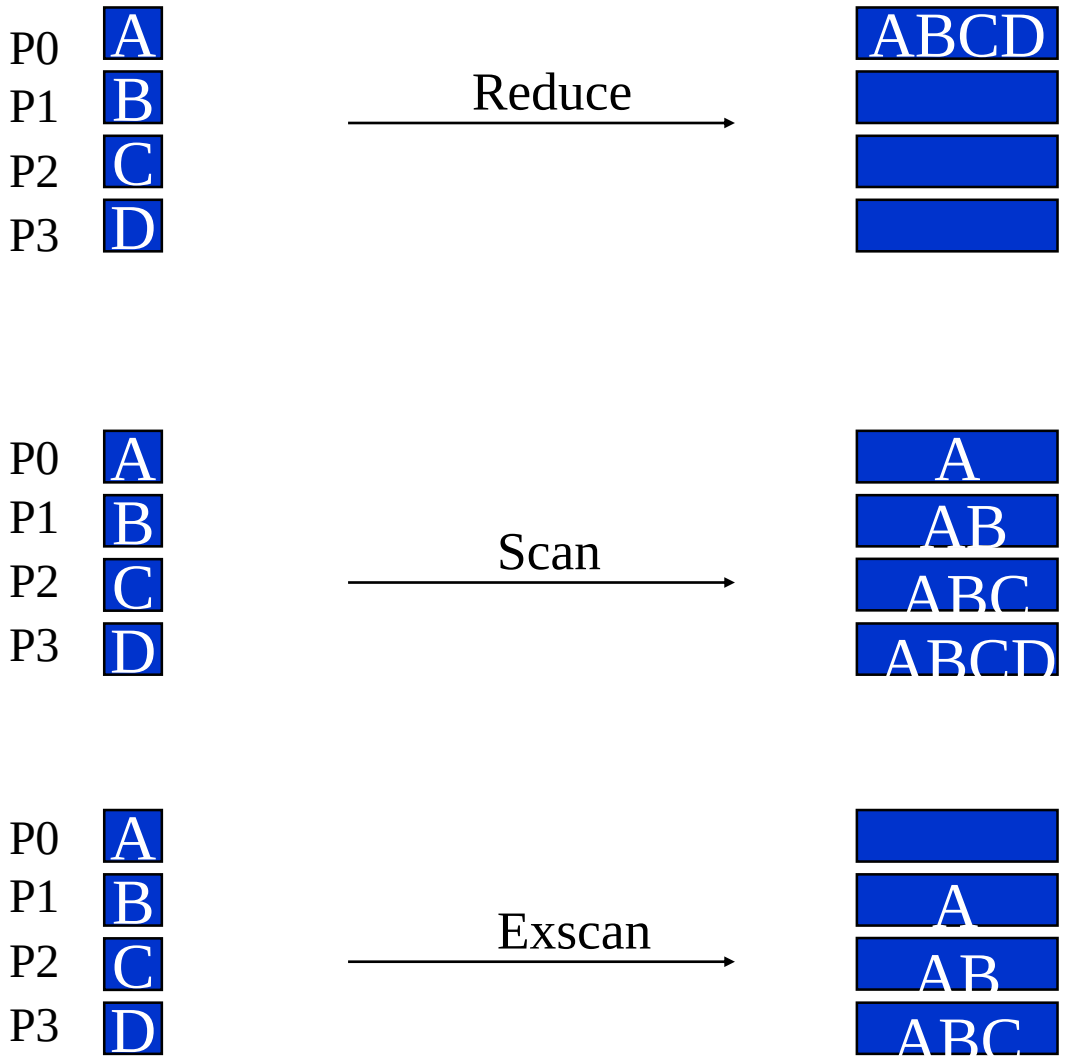
- All collective operations must be called by *all* processes in the communicator
- MPI_Bcast is called by both the sender (called the root process) and the processes that are to receive the broadcast
  - "root" argument is the rank of the sender; this tells MPI which process originates the broadcast and which receive

- Will get experience with this in today's lab

More Collective Data Movement



Collective Computation



MPI Built-in Collective Computation Operations

• **MPI_MAX**	Maximum
• **MPI_MIN**	Minimum
• **MPI_PROD**	Product
• **MPI_SUM**	Sum
• **MPI_LAND**	Logical and
• **MPI_LOR**	Logical or
• **MPI_LXOR**	Logical exclusive or
• **MPI_BAND**	Binary and
• **MPI_BOR**	Binary or
• **MPI_BXOR**	Binary exclusive or
• **MPI_MAXLOC**	Maximum and location
• **MPI_MINLOC**	Minimum and location

Lower Bounds

- Using "α–β–γ model", we can give lower bounds on the cost of collectives
  - Assume one MPI process per node
  - Latency
    - lower bound on latency is derived by the simple observation that for all collective communications at least one node has data that must somehow arrive at all other nodes.
    - Under the model, at each step, we can at most double the number of nodes that get the data
  - Computation
    - Only some collectives require computation
  - Bandwidth

- Let x be a vector of length n
- For some operations, x is divided into subvectors $x_i, i = 0, \ldots, p-1$ where p is number of nodes
- Superscript indicates vector that must be reduced with other vectors from other nodes

Lower bounds for simple send/recv

Before		After	
Node 0	Node 1	Node 0	Node 1
x			x

- Number of messages: 1
- Number of words moved: n
- Flops: 0

Lower bound on cost:

$$\alpha + \beta n$$

Scatter

Before				After			
Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
x_0				x_0			
x_1					x_1		
x_2						x_2	
x_3							x_3

- Can be done in tree-like manner
- Number of messages $=\lceil \log_2 p \rceil$
- Words moved $= \frac{(p-1)n}{p}$ (doesn't need to send to itself)
- flops:0

Lower bound on cost:

$$\alpha \lceil \log_2 p \rceil + \beta \frac{(p-1)n}{p}$$

Gather

Before				After			
Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
x_0				x_0			
	x_1			x_1			
		x_2		x_2			
			x_3	x_3			

- Again, done in tree-like manner
- Number of messages $=\lceil \log_2 p \rceil$
- Words moved $= \frac{(p-1)n}{p}$ (doesn't need to receive from itself)
- flops:0

- Lower bound on cost:

$$\alpha \lceil \log_2 p \rceil + \beta \frac{(p-1)n}{p}$$

Allgather

Before				After			
Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
x_0				x_0	x_0	x_0	x_0
	x_1			x_1	x_1	x_1	x_1
		x_2		x_2	x_2	x_2	x_2
			x_3	x_3	x_3	x_3	x_3

- Butterfly communication scheme
- Number of messages $=\lceil \log_2 p \rceil$
- Words moved $= \frac{(p-1)n}{p}$ (doesn't need to receive from itself)
- flops:0

- Lower bound on cost:

$$\alpha \lceil \log_2 p \rceil + \beta \frac{(p-1)n}{p}$$

Broadcast

Before				After			
Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
x				x	x	x	x

- Scatter + Allgather
- flops = 0

Lower bound on cost:

$$2\alpha\lceil\log_2 p\rceil + 2\beta\frac{p-1}{p}n$$

Reduce-Scatter

Before				After			
Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
$x_0^{(0)}$ $x_1^{(0)}$ $x_2^{(0)}$ $x_3^{(0)}$	$x_0^{(1)}$ $x_1^{(1)}$ $x_2^{(1)}$ $x_3^{(1)}$	$x_0^{(2)}$ $x_1^{(2)}$ $x_2^{(2)}$ $x_3^{(2)}$	$x_0^{(3)}$ $x_1^{(3)}$ $x_2^{(3)}$ $x_3^{(3)}$	$\sum_j x_0^{(j)}$	$\sum_j x_1^{(j)}$	$\sum_j x_2^{(j)}$	$\sum_j x_3^{(j)}$

- Number of messages $= \lceil \log_2 p \rceil$
- Words moved $= \frac{(p-1)n}{p}$ (doesn't need to receive from itself)
- flops: $\frac{(p-1)n}{p}$

- Lower bound on cost:

$$\alpha \lceil \log_2 p \rceil + \beta \frac{(p-1)n}{p} + \gamma \frac{(p-1)n}{p}$$

Reduce

Before				After			
Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	$\sum_j x^{(j)}$			

- Reduce-scatter + gather

Lower bound on cost:

$$2\alpha\lceil\log_2 p\rceil + 2\beta\frac{(p-1)n}{p} + \gamma\frac{(p-1)n}{p}$$

Allreduce

Before				After			
Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$

- Reduce-scatter + Allgather

Lower bound on cost:

$$2\alpha\lceil\log_2 p\rceil + 2\beta\frac{(p-1)n}{p} + \gamma\frac{(p-1)n}{p}$$

Special Variants of Collectives

- The basic routines send the same amount of data from each process
  - E.g., MPI_Scatter(&v,1,MPI_INT,...) sends 1 int to each process
- What if you want to send a different number of items to each process?
- Use MPI_Scatterv
  - The “v” (for vector) routines allow the programmer to specify a different number of elements for each destination (one to all routines) or source (all to one routines).
  - Efficient algorithms exist for these cases, though not as fast as the simpler, basic routines

- In one case (MPI_Alltoallw), there are two "vector" routines, to allow more general specification of MPI datatypes for each source
- Alltoallw : allows different data types

Determinism in Collective Computations

- In exact arithmetic, you always get the same results
  - but roundoff error, truncation can happen
- MPI does not require that the same input gives the same output every time
  - Implementations are encouraged but not required to provide exactly the same output given the same input
  - Round-off error may cause slight differences
- Allreduce does guarantee that the same value is received by all processes for each call
- Why didn't MPI mandate determinism?
  - Not all applications need it
  - Would be sacrificing performance!

Nonblocking Collectives

- Introduced in MPI-3 for all collectives
  - e.g., MPI_Ibcast()
- As in point-to-point case, all calls return immediately
- All return an "MPI_Request" - handle on a nonblocking operation
  - used by MPI_Wait or MPI_Test to know when non-blocking operation completes
- Can give performance benefit (not necessarily)
  - Application must be able to do enough work between when collective begins and when collective must be completed to offset additional overhead of Wait() or Test()
  - Larger message size = requires more computation to offset costs

MPI Topologies

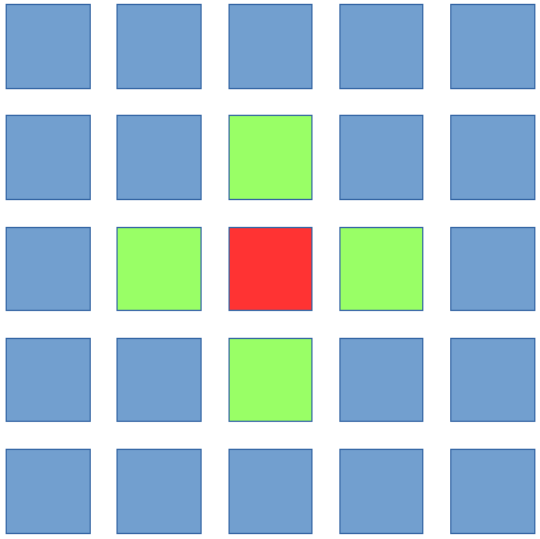
- A communicator describes a group of processes, but the structure of your computation may not be such that every process will communicate with every other process.
- Ex: in a computation that is mathematically defined on a Cartesian 2D grid, the processes themselves act as if they are two-dimensionally ordered and communicate with N/S/E/W neighbors.
- If MPI had this knowledge about your application, could conceivably optimize for it
  - e.g., by renumbering the ranks so that communicating processes are closer together physically in your cluster.
- Mechanism to declare this structure of a computation to MPI is known as a *virtual topology*

MPI Topologies

- Defined types of topologies:
- MPI_UNDEFINED
  - holds for communicators where no topology has been explicitly specified
- MPI_CART
  - Cartesian topology
- MPI_DIST_GRAPH
  - general graph topology

Cartesian Topology

- A Cartesian topology is a mesh
- neighborhood communication simply involves the nearest neighbors in all directions
- Two processes are in the same neighborhood if all coordinates are the same except for possibly one coordinate, and that coordinate must be no more than one in difference between the two processes



Cartesian Topology

The routine MPI_Cart_create creates a Cartesian decomposition of the processes, with the number of dimensions given by the ndim argument.

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int
*periods, int reorder, MPI_Comm *comm_cart)
```

Reorder argument

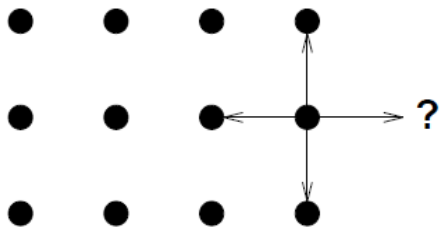
- The reorder parameter to *MPI_Cart_create* indicates whether processes can have a rank in the new communicator that is different from in the old one.

- In many parallel computer interconnects, some processors are closer to than others. These routines allow the MPI implementation to provide an ordering of processes in a topology that makes logical neighbors close in the physical interconnect.

- The reorder argument is used to request the best ordering.

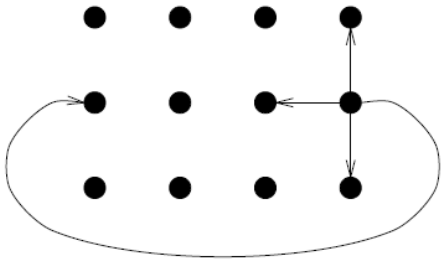
Periods argument

Who are my neighbors if I am at the edge of a Cartesian Mesh?



int* periods is an array that, for each dimension, holds 0 if nonperiodic, 1 if periodic

- Periodic
- Nonperiodic
  - indicated by a rank of MPI_PROC_NULL.
  - This rank may be used in send and receive calls in MPI. The action in both cases is as if the call was not made.



Cartesian Topology

- Each point in this new communicator has a coordinate and a rank. They can be queried with *MPI_Cart_coords* and *MPI_Cart_rank* respectively.

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,int
*coords);
```

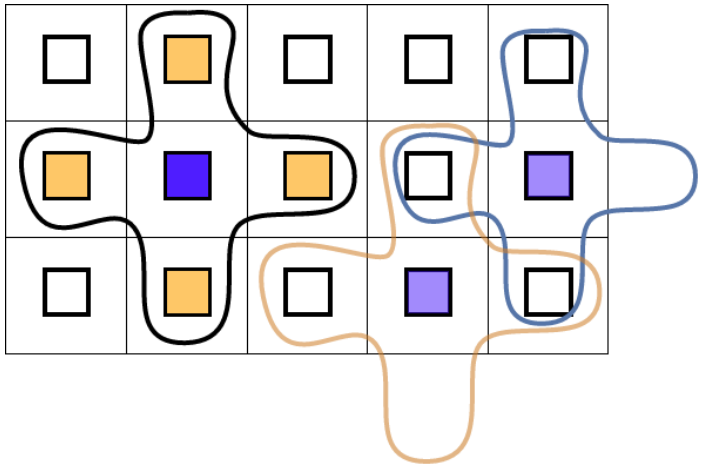
```
int MPI_Cart_rank(MPI_Comm comm, init *coords, int *rank);
```

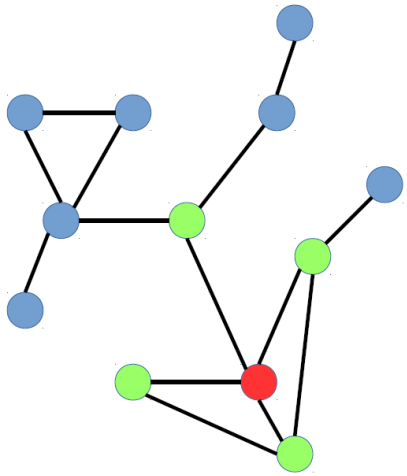
- Note that these routines can give the coordinates for any rank, not just for the current process.

Distributed Graph Topology

- In many calculations on a grid (using the term in its mathematical, *FEM*, sense), a grid point will collect information from grid points around it.
- Under a sensible distribution of the grid over processes, this means that each process will collect information from a number of neighbor processes
- number of neighbors is dependent on the process.
  - e.g., in a 2D grid (assuming a five-point stencil) most processes communicate with four neighbors; processes on the edge with three, and processes in the corners with two.

Illustration of a distributed graph topology where each node has four neighbors
http://pages.tacc.utexas.edu/~eijkhout/pcse/html/mpi-topo.html





Distributed Graph Topology

- Two creation routines for process graphs
- MPI_Dist_graph_create_adjacent
  - assumes that process knows both who it is sending to and who will send to it
- MPI_Dist_graph_create
  - specify on each process only who the process will be sending to

Neighborhood Collectives

- MPI offers "neighborhood collectives", i.e., collectives defined only on neighboring processes as defined by dist graph structure
- eg: MPI_Neighbor_allgather

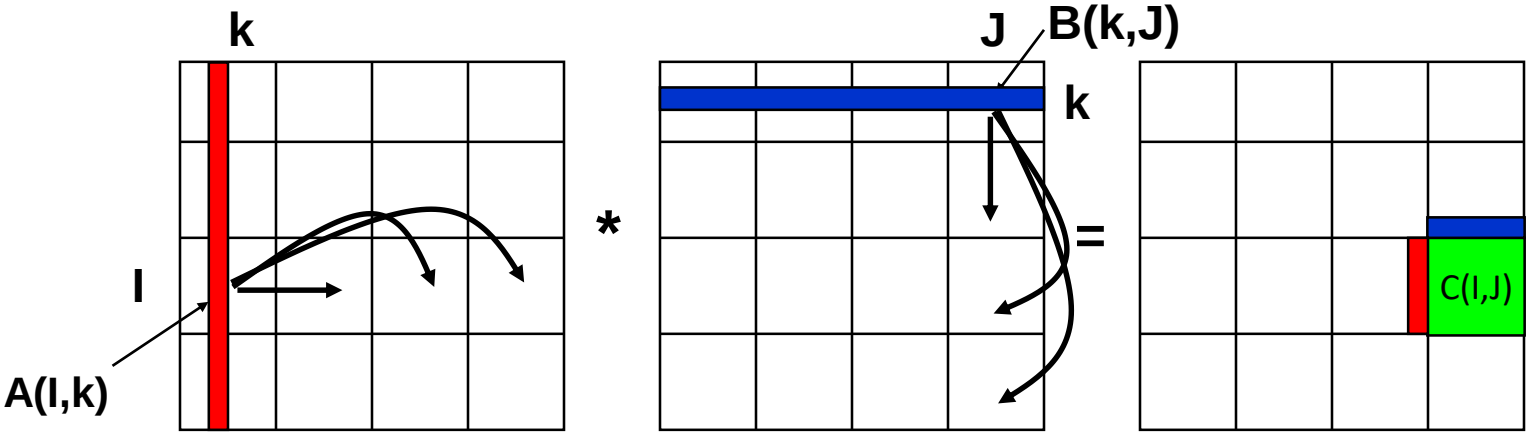
```
int MPI_Neighbor_allgather (  const void *sendbuf, int sendcount,
                              MPI_Datatype sendtype, void *recvbuf,
                              int recvcount, MPI_Datatype recvtype,
                              MPI_Comm comm)
```

- Neighbor collectives have the same argument list as the regular collectives, but they apply to a graph communicator
- Others: MPI_Neighbor_alltoall, MPI_Neighbor_allgatherv, MPI_Neighbor_alltoallv, MPI_Neighbor_alltoallw
- Also called "sparse collectives"

Example: SUMMA Algorithm

- SUMMA = Scalable Universal Matrix Multiply
- Slightly less efficient than Cannon
  ... but simpler and easier to generalize
- Presentation from van de Geijn and Watts
  - www.netlib.org/lapack/lawns/lawn96.ps
  - Similar ideas appeared many times
- Used in practice in PBLAS = Parallel BLAS
  - www.netlib.org/lapack/lawns/lawn100.ps

SUMMA



- I, J represent all rows, columns owned by a processor
- k is a single row or column
  - or a block of b rows or columns

- C(I,J) = C(I,J) + A(I,k)*B(k,J)

- Assume a p_r by p_c processor grid ($p_r = p_c = 4$ above)
  - Need not be square

MPI_Comm_split

```
int MPI_Comm_split( MPI_Comm comm,
                    int color,
                    int key,
                    MPI_Comm *newcomm)
```

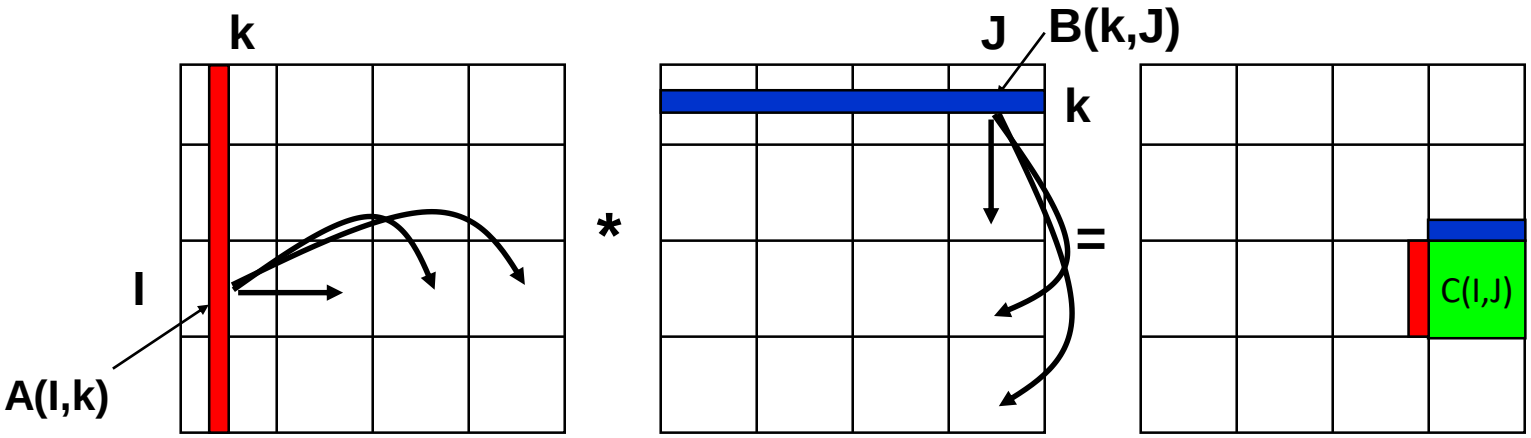
MPI's internal Algorithm:

1. Use MPI_Allgather to get the color and key from each process
2. Count the number of processes with the same color; create a communicator with that many processes. If this process has MPI_UNDEFINED as the color, create a process with a single member.
3. Use key to order the ranks

Color: controls assignment to new communicator

Key: controls rank assignment within new communicator

SUMMA



For k=0 to n-1 ... or n/b-1 where b is the block size

... = # cols in A(I,k) and # rows in B(k,J)

for all I = 1 to p_r ... in parallel

owner of A(I,k) broadcasts it to whole processor row

for all J = 1 to p_c ... in parallel

owner of B(k,J) broadcasts it to whole processor column

Receive A(I,k) into Acol

Receive B(k,J) into Brow

C(myproc , myproc) = C(myproc , myproc) + Acol * Brow

(naïve) SUMMA in MPI

```
void SUMMA(double *mA, double *mB, double *mC, int p_c)
{
    int row_color = rank / p_c; // p_c = sqrt(p) for simplicity
    MPI_Comm row_comm;
    MPI_Comm_split(MPI_COMM_WORLD, row_color, rank, &row_comm);

    int col_color = rank % p_c;
    MPI_Comm col_comm;
    MPI_Comm_split(MPI_COMM_WORLD, col_color, rank, &col_comm);

    for (int k = 0; k < p; ++k) {
        if (col_color == k)  memcpy(Atemp, mA, size);
        if (row_color == k)  memcpy(Btemp, mB, size);

        MPI_Bcast(Atemp, size, MPI_DOUBLE, k, row_comm);
        MPI_Bcast(Btemp, size, MPI_DOUBLE, k, col_comm);

        SimpleDGEMM(Atemp, Btemp, mC, N/p, N/p, N/p);
    }
}
```

Reading for Today

- PRAM paper
- BSP paper
- LogP paper
- Collective communication: theory, practice, and experience (abg model)

- All the above located in today's folder on Moodle