Modeling of Parallel Programs

- Goal: produce better parallel algorithms and ultimately parallel programs by creating abstract models of (parallel) computation
  - Note: many models for sequential computing, e.g., random access machine (RAM)

RAM Model for Sequential Computing

- Some features:
  - Unbounded number of local memory cells
  - Each memory cell can hold an integer
  - Instruction set allows register-register operations and also memory-memory operations via indirect addressing
  - Also includes branching
  - All operations take unit time, regardless of how complicated the operation seems

Early Parallel Models: PRAM and its cousins

- Parallel random access machine (PRAM) was first major model for parallelism
  - Basic ideas:
    - Natural extension of RAM model
    - P processors and a single shared memory; MIMD
    - **all instructions take unit time**
  - PRAM is unrealistic
    - Because: (1) unit-time access to memory and (2) inter-processor synchronization and communication is assumed to be free
    - Would be ok if it still led to efficient parallel programs, but it doesn't

EREW PRAM

- One example of a follow-on model
  - Goal: more realism than PRAM
  - Idea: cannot be more than one access to a single memory location at any point in time
    - Still unrealistic, because memory actually exists at the granularity of modules, not individual locations
    - Also does not deal with the unrealistic unlimited communication assumption of PRAM

LogP Model: adds some realism

- LogP developed by a group of theoreticians and practitioners
- Famous model; several researchers have added to LogP
- LogP assumes a cluster, though it can also be used for a multicore machine

LogP Model

- Four parameters in LogP:
  - *L* – latency: the upper bound on the delay incurred in sending a small message from one node to another
  - *o* – overhead: the time to send/receive message in terms of system overhead
    - i.e., receive is time from reception at network device until the time application can continue

LogP Model

- Four parameters in LogP:
  - *g* – gap: minimum time between consecutive message transmissions/receptions at a node.
    - 1/g is the bandwidth, because the bandwidth is defined as amount of data per unit time that can be sent. (Intuitively, if *g* were 0, applications could constantly send and would have infinite bandwidth.)
  - *P* -- # processors
    - Local operations take unit time

Coming Back to LogP vs PRAM

- The unit time for any instruction assumption in PRAM implies that
  - $g = L = o = 0$
  - Therefore, PRAM does not discourage algorithms with arbitrarily large amounts of communication

LogP Model

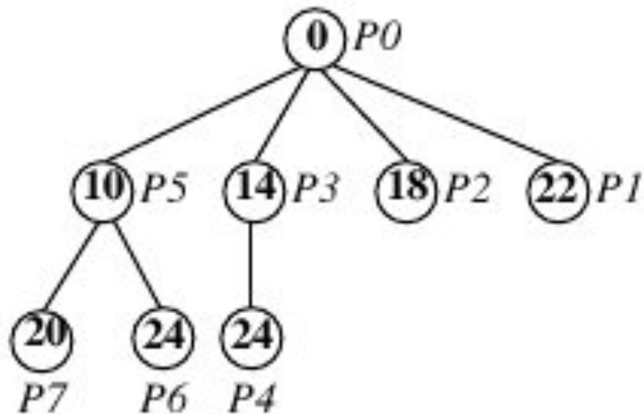
- Constraints in LogP
  - At most ceil(L/g) messages can be in transit to/from any processor, because the network has finite capacity.
    - It takes L time units to get a message from A to B, and A can't send more than every g time units.
  - Messages are of small size

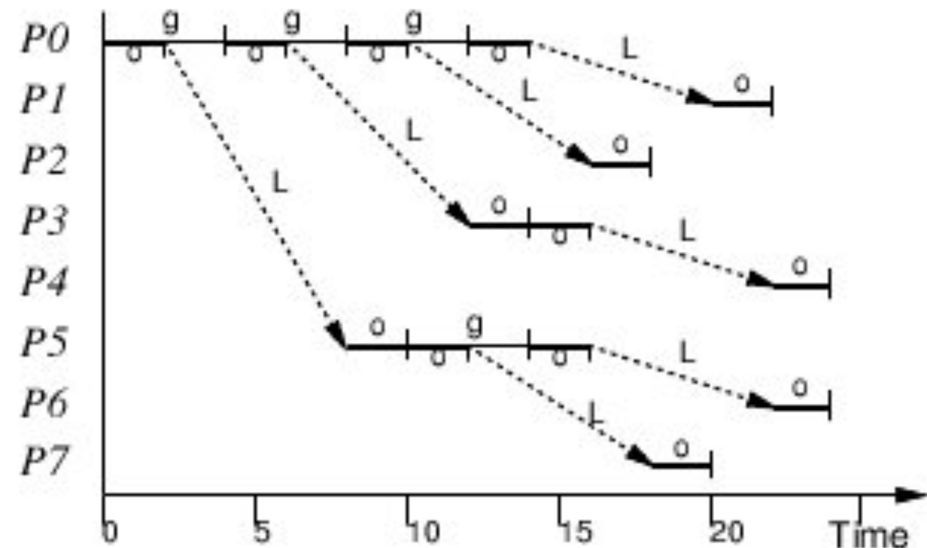
LogP Model

- Sometimes parameters can be ignored or simplified
  - If messages sent in long streams from source to destination, then L may be disregarded.
    - Intuitively, imagine a 500 MB download; does it really matter whether it takes 100 ms or 200 ms to receive the first byte?
    - Formally, g is the dominant factor in this situation.
  - If o >= g, g can be ignored
    - It is impossible to send two messages with less than g time between them anyways
  - If a machine has a network co-processor, o may be zero (if the CPU can do other work)

LogP Broadcast Example

P = 8, L = 6, g = 4, o = 2



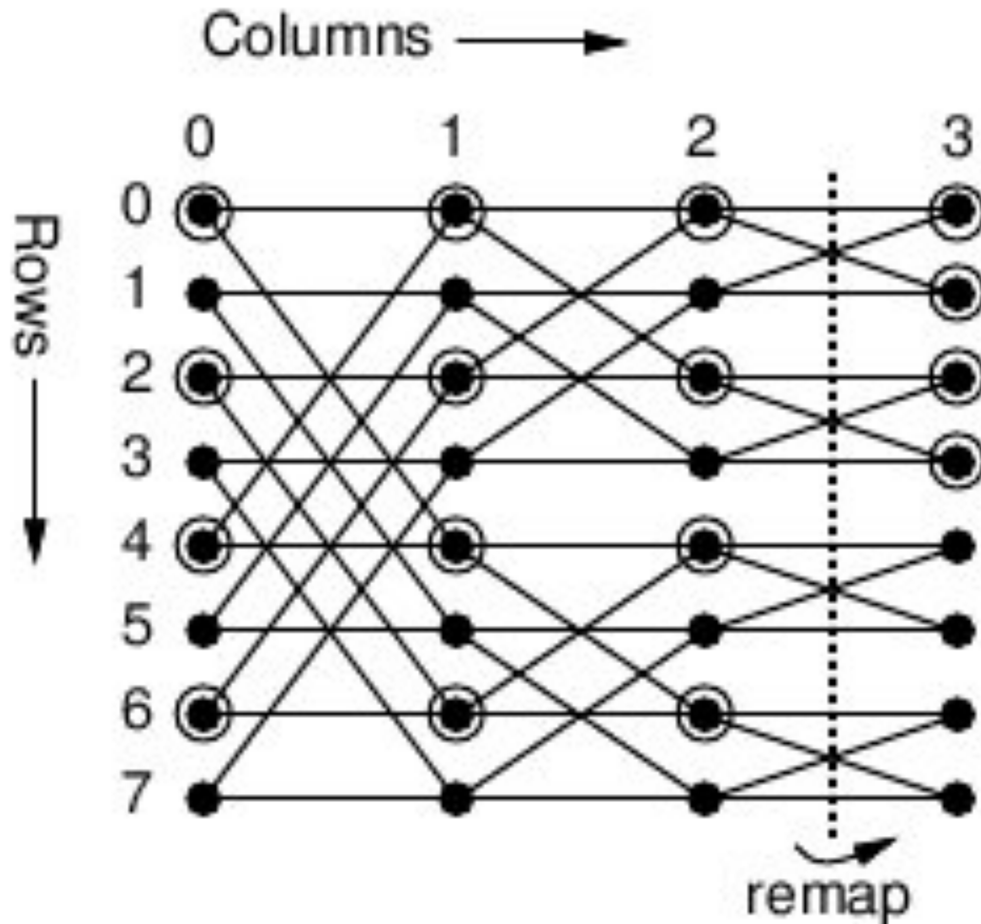


Class Exercises:

(1) what's the optimal broadcast tree if L is 0?

(2) what's the optimal broadcast tree if g is 0?

LogP FFT Example



Analyzing FFT Parallel Algorithm Alternatives

- With a Cyclic distribution
  - Processor 0 is assigned points 0, 2, 4, and 6
  - Take point 0, for example:
    - It needs points 4, 2, and then 1, on the three steps
    - It will need to communicate with Processor 1 on the **third** step

Analyzing FFT Parallel Algorithm Alternatives

- With a Block distribution
  - Processor 0 is assigned points 0, 1, 2, and 3
  - Take point 0, for example:
    - It needs points 4, 2, and then 1, on the three steps
    - It will need to communicate with Processor 1 on the **first** step

Analyzing FFT Parallel Algorithm Alternatives

- Cyclic and Block distributions have the exact same computation and communication requirements
  - Assume that *n* is the number of points, and *p* is the number of processors
  - Compute time: *log n * n/p,* because there are *log n* steps and in each one, a processor has to compute *n/p* points (each one takes unit time, per the LogP model)

Analyzing FFT Parallel Algorithm Alternatives

- Cyclic and Block distributions have the exact same computation and communication requirements
  - Communication time: $\log p * (g * n/p + L)$, because there are *log p* communication steps, and in each one, there are *n/p* points communicated, and the last of those points is sent at time $g * n/p$, and then that last point reaches the destination after a *L* time

Analyzing FFT Parallel Algorithm Alternatives

- Can redistribute the data (from Cyclic to Block) in between step 2 and 3
  - Computation time is unchanged
  - Communication time: $(g * n/p - g * n/p^2 + L)$, because all points must be redistributed, except for ones that a processor owns in **both** Cyclic and Block distributions
    - Overall, communication time is lower by a factor of $log\ p$
    - In general, the number of points that need not be redistributed is n/p^2
      - Can be important if n is large

Scheduling All-to-All

- Could have every node send to node 0; then, every node send to node 1, etc.
  - Creates bottleneck at node 0, then node 1, etc.
  - Means that only L/g messages can be in transit at a time (this is again the capacity constraint)
- Better: node i sends what it needs to i+1, then i+2, etc. (with wraparound)

Scheduling All-to-All

- More broadly, this has implications for message-passing implementations (e.g., MPI) to *schedule* messages for collectives
  - This is also a good reason to invoke collectives instead of manually implementing a collective (via Send/Receive)

LogP Extensions

- Are many:
  - LogGP: adds support for long messages (LogP assumes essentially one byte messages)
    - Basically, allows one message and charges a per-byte cost (eliminates *o* and *g* per [small] message)
  - LogGPS: adds support for synchronization cost incurred to to rendezvous when large messages are sent (e.g., MPI_Send when the message is sufficiently large---the matching MPI_Recv must be invoked before MPI_Send can complete)
  - LognP: adds support for cost of middleware (e.g., marshalling a column into a vector)