Communication in Shared and Distributed Memory

CS121 Parallel Computing

Spring 2017

Today

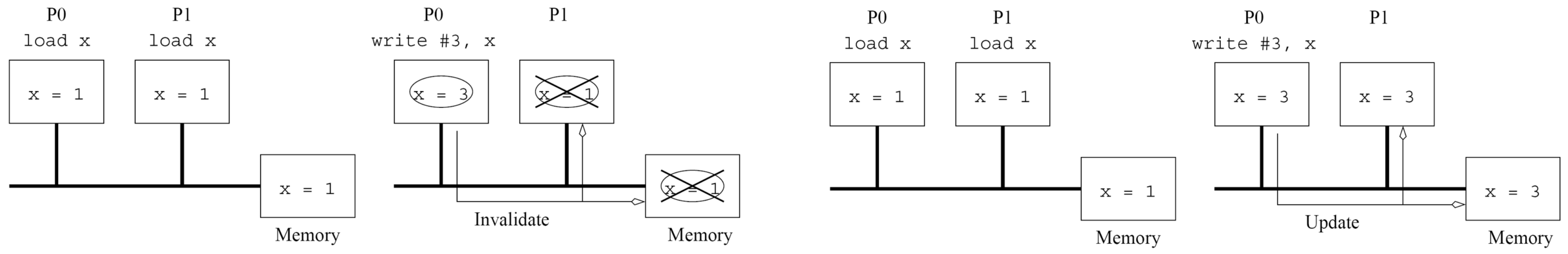
- Recitations Thursdays 5-6pm in SIST 1A-108.
- TA 彭镜铨 pengjq@shanghaitech.edu.cn
  - Turn in your solutions to TA before recitations.
- Start forming teams and thinking about your paper topics.
- Outline
  - Cache coherency
  - Communication costs
  - Routing
  - Topology mappings

The cache coherence problem

- In shared memory systems, there is only one logical copy of each variable.
- However, if several processors want to access the variable, they can each store it in their cache, to avoid latency to access main memory.
- When one processor modifies the variable, other processors must be aware of the change.
- Since the variable is stored in multiple caches, the caches have be made coherent through some procedure / protocol.

Cache coherence protocols

- Two main types of protocols, invalidate and update.
- Invalidate When one process modifies variable, all other cached copies and copy in memory are declared invalid.
    - If another process wants to access the variable, first process writes back new value to memory, and second process reads from memory.
- Update When one process modifies variable, it writes new value to other caches and memory.
- Invalidate and update tradeoff communication vs speed.
    - If other processes don't read the variable, update wastes communication.
    - If other processes do read the variable, invalidate causes stall for writeback and read.
- Since bandwidth is limited in parallel systems, most use invalidate.



Source: Introduction to Parallel Computing, Grama et al.

MSI protocol

- Invalidation based coherency protocol.
- The basis for widely used and higher performance protocols, e.g. MESI and MOESI.
- A variable X can be in the M (modified), S (shared) or I (invalid) state.
- When a processor performs an action (e.g. read or write), it may cause a state change and coherence messages to be sent to the main memory and all other processors.

MSI protocol

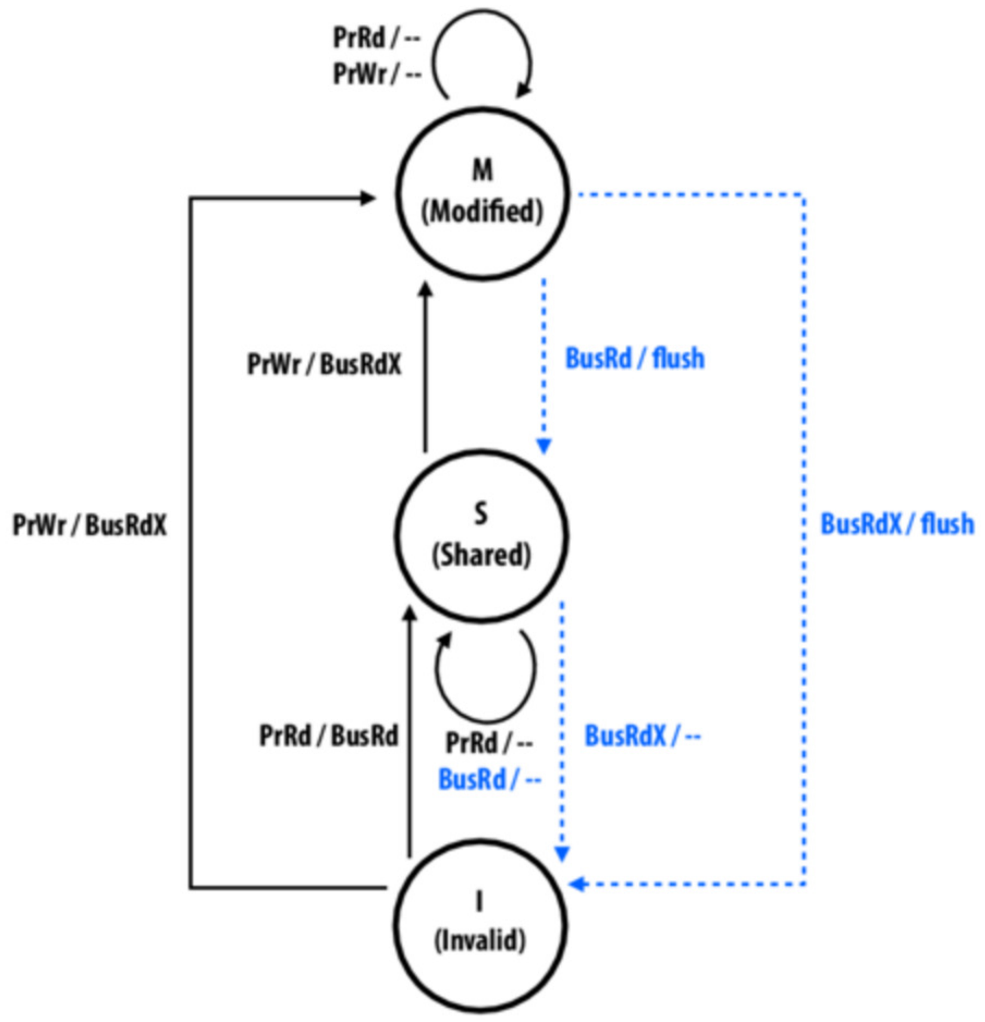
- S (shared): X may exist in multiple caches.
  - X's value hasn't changed since last time each processor read X from main memory.
    - All copies of X are up to date.
  - Processors can continue to read X from own caches.
- M (modified / dirty) state
  - A processor p changed its cached value of X.
  - p must inform other processors their cached X value now invalid.
  - p can still read / write to X in its own cache.
  - When p evicts X, must write back (flush) its latest value to main memory.
- I (invalid): Another processor has changed X's value.
  - This processor's value for X is out of date.
  - Must read X from main memory.
  - When reading X, will cause processor holding X in M state to first write its value to main memory.

MSI example

Time

Instruction at Processor 0	Instruction at Processor 1	Variables and their states at Processor 0	Variables and their states at Processor 1	Variables and their states in Global mem.
				x = 5, D
				y = 12, D
read x		x = 5, S		x = 5, S
	read y		y = 12, S	y = 12, S
x = x + 1		x = 6, D		x = 5, I
	y = y + 1		y = 13, D	y = 12, I
read y		y = 13, S	y = 13, S	y = 13, S
	read x	x = 6, S	x = 6, S	x = 6, S
x = x + y		x = 19, D	x = 6, I	x = 6, I
	y = x + y	y = 13, I	y = 19, D	y = 13, I
x = x + 1		x = 20, D		x = 6, I
	y = y + 1		y = 20, D	y = 13, I

MSI state machine



Source: http://15418.courses.cs.cmu.edu/spring2015content/lectures/10_cachecoherence1/

- A / B: Input request A, output action B.

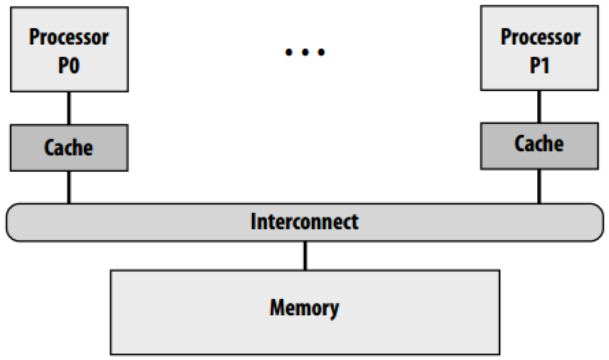
- PrRd: processor read request
- PrWr: processor write request
- flush: processor writes back data to memory
- BusRd: processor with out of date data wans to read its current value
- BusRdX: processor with out of date data wants to write new value

Examples

- Var currently shared. Proc writes new value (generates PrWr), leading to BusRdX msg. Proc goes to M state. All other procs (currently in S state) go to I state.
- Var currently dirty (last updated by proc in M state). Proc in I state wants to read var (generates PrRd), leading to BusRd msg. Proc in M state flushes current var value to memory. All procs go to S state.

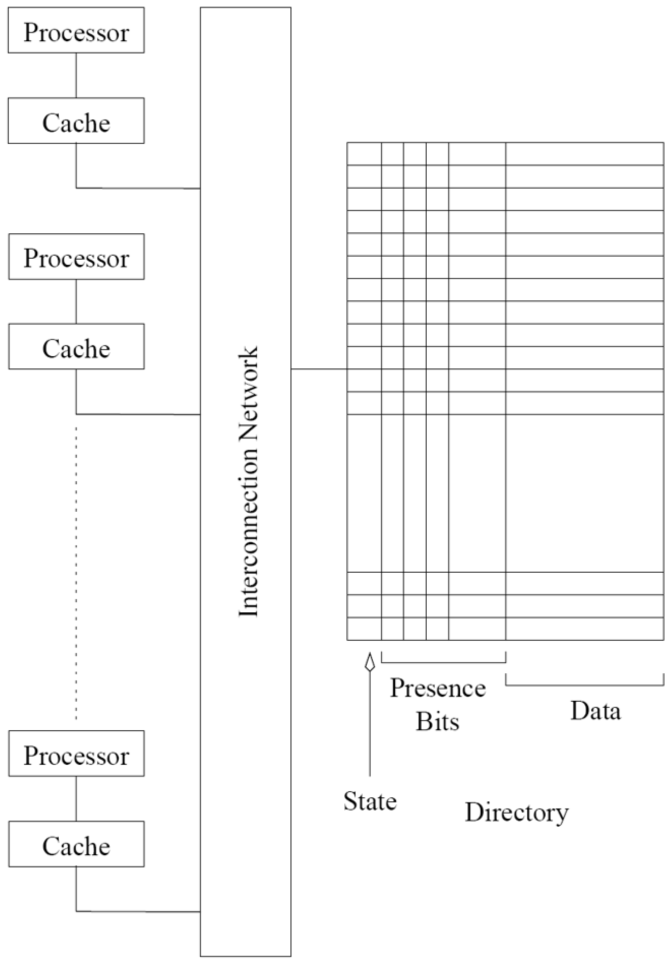
Snoopy cache system

- MSI (and other protocols) can be implemented on different types of hardware.
- A basic (i.e. cheap) setup is a snoopy coherence protocol implemented on a bus or ring.
  - All processes listen on the bus for coherency traffic and respond accordingly.
  - Protocol works correctly because messages are broadcast on bus, so all processes hear all messages.
- If only one processor modifies a variable, all accesses are to its cache, and no coherence traffic $\Rightarrow$ good performance.
- If multiple processors modify the variable, many flushes and coherency traffic $\Rightarrow$ bus becomes performance bottleneck.



Directory cache system

- Based on observation that only processors with shared or dirty copy of variable need to hear its coherence messages.
- Keep a directory data structure in memory, indicating for each memory block which processors hold shared or dirty copies.
  - Given p procs, for each memory block use p bits indicating processors relevant for the block.

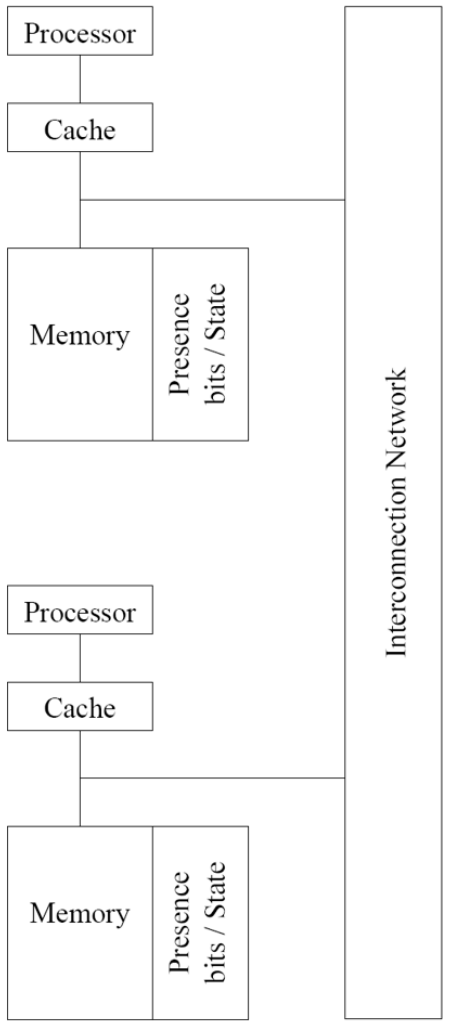


Example

- Initially several procs hold var X in S state. They're marked in X's presence bits in directory.
- Proc 0 modifies X. Main mem and all present procs besides proc 0 recv invalidate msg.
- All their presence bits for X are reset; only proc 0 still present in directory.
- Later (invalid) proc 1 reads X. Since proc 1 in I state, it reads directory, finds proc 0 in M state, and causes proc 0 to flush X.
- Procs 0 and 1 marked in directory in S state.

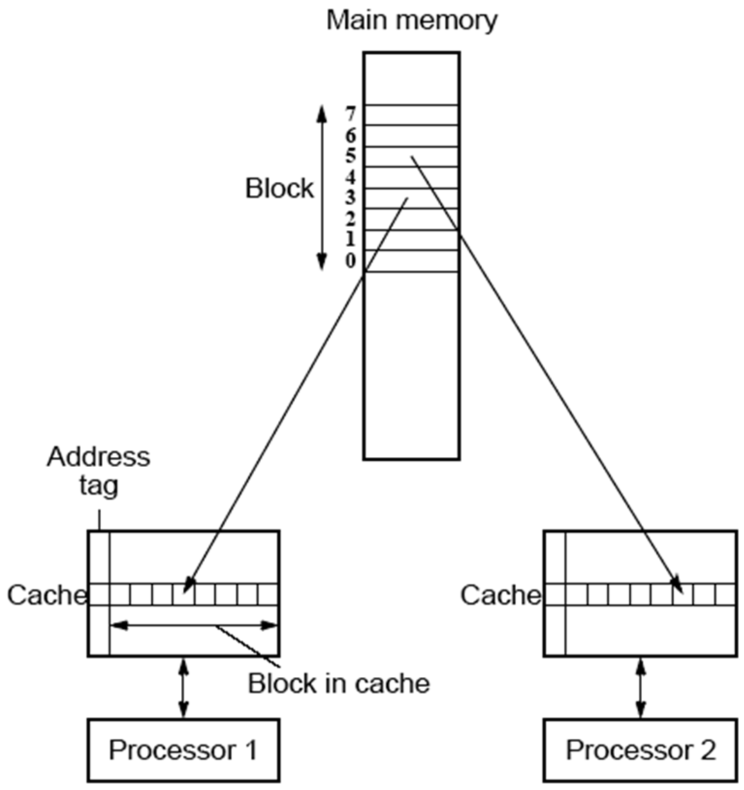
Distributed directory cache

- Since directory stored in memory, all coherency traffic access memory, which becomes bottleneck.
- Also, directory needs large O(mp) amount of storage, m = # memory blocks, p = # processors.
- To improve performance, distribute directory among the processors.
  - Each processor handles a fixed memory range, and stores directory for the range.
  - Processor needing directory info on a var queries processor responsible for the var.



False sharing

- Data transferred into / from cache in units of cache lines (aka blocks).
  - Typical L1 cache line size is 64-128B.
  - Multiple words fit into one cache line.
- Processors accessing different words can access same cache line.
  - If one processor modifies cache line, other processors with same cache line must be updated or invalidated.
- Solution is (try to) lay out data so data accessed by different processors in different lines.
  - Padding.
  - Wastes precious cache space.

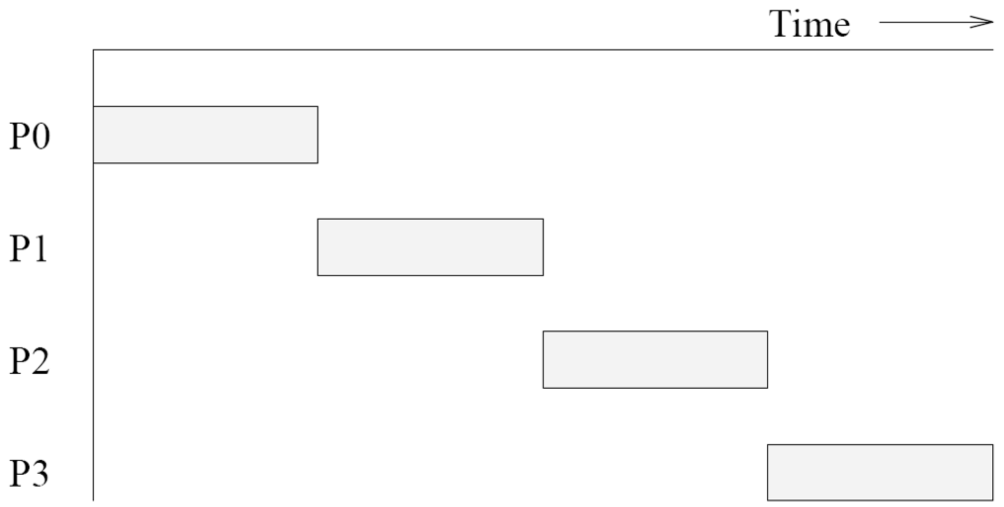


Communication costs

- First consider cost in message passing systems, then shared memory.
- Time to transfer a message from one node to another consists of
  - Startup time t_s: Prepare message header, error correction, running routing protocol, interfacing with router, etc.
    - Once per message.
  - Per hop time t_h: Used by router to determine next hop.
  - Per word transfer time t_w: If bandwidth is r, per word transfer time from source to dest is 1/r.

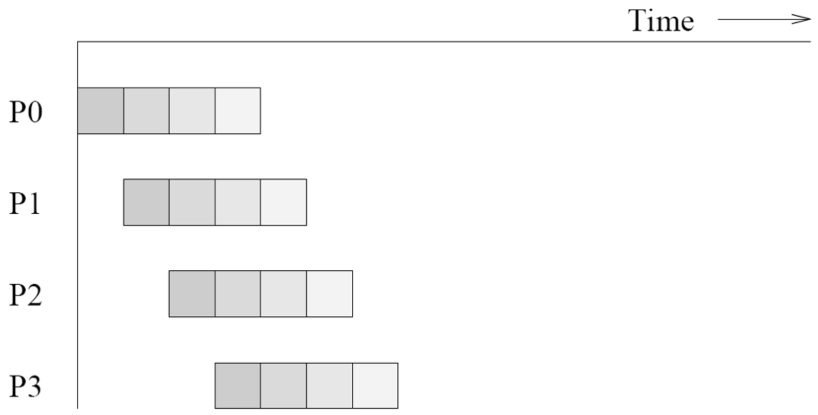
Store and forward routing

- Each intermediate node waits to receive entire message before forwarding it to next hop on path.
- Takes m t_w+t_h time per link, so for l links t_s+ l(m t_w+t_h) total time.
  - t_h small compared to m t_w, so time is ≈ t_s+ l m t_w.



Packet routing

- Break large message into small packets.
- Intermediate node can receive a packet as soon as it finishes forwarding previous packet it received.
  - Pipeline parallelism.
- Also lets packets take different routes, and easier error correction.
  - Essential for Internet, with unpredictable network state, high error rates.
- But overhead from each packet's routing info, error correction, sequence number etc.
- Total time $t_s + t_h l + t_w m$.
  - First packet arrives at time $t_s + t_h l + t_w$.
  - Remaining m-1 packets arrive every t_w time.



Cut-through switching

- Similar to packet routing, but optimized for parallel computers.
- Messages take same path. No routing info.
- Small (~ 32B) flits (flow control digits) transmitted at high rate.
  - First flit (header) is allocated path by switches / routers. Remaining (body and tail) flits follow same path.
- Simple, per message (instead of per packet) error correction.
- To reduce latency for high priority flits (e.g. cache lines), use multilane cut-through routing.
- Also called virtual cut-through. A related technique is wormhole routing.

Overall cost model

- Total communication time depends on data volume m and number of hops l.
- However, programmer has little control over l.
  - Usually can't control process to processor mapping.
  - Many systems use randomized two step routing to minimize congestion: first send message to random node, then send it to destination.
  - Per hop time t_h usually small compared to t_s or m t_w.
- Communication time can be simplified to t_s+m t_w.
- Assumes uncongested network.
  - If congestion on link is c, time becomes t_s + c m t_w.
- Ex $\sqrt{p} \times \sqrt{p}$ mesh where nodes randomly communicate with each other.
  - Across a bisection there are O(p) communications.
  - But mesh bisection width is $\sqrt{p}$.
  - So some link carries $O(p) / \sqrt{p} = O(\sqrt{p})$ messages.
  - So communication time is $t_s + O(\sqrt{p}\, m\, t_w)$.

Comm cost in shared memory

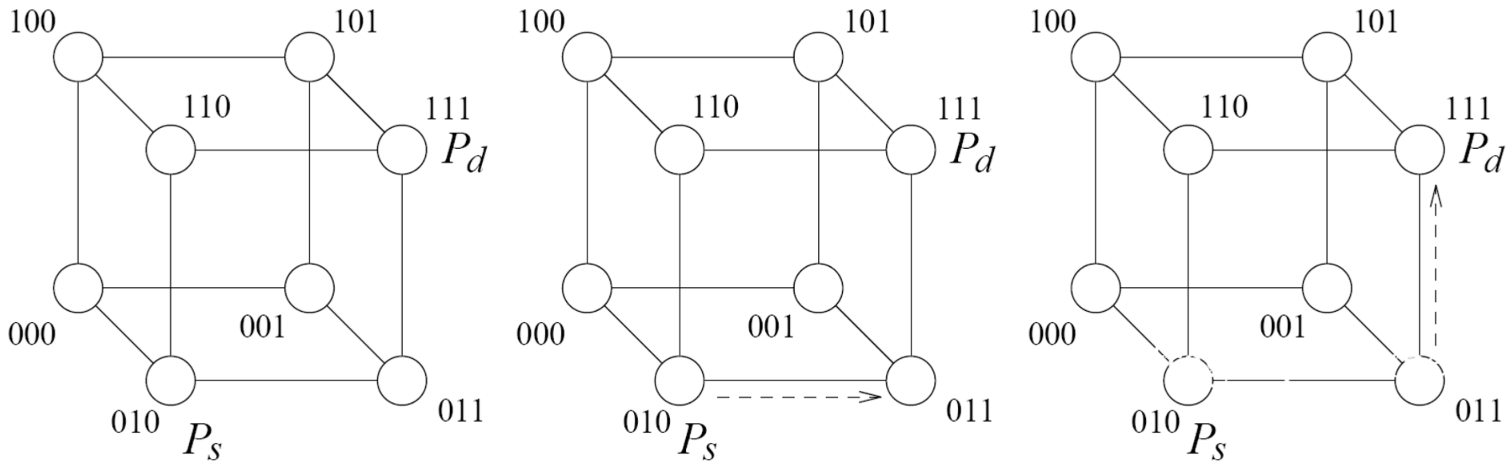
- Shared memory communication costs are more unpredictable.
  - Memory layout determined by system. Hard to estimate amount of local / remote memory accesses.
  - Cache thrashing depends on scheduling and process allocation.
  - Coherency traffic, spatial locality and false sharing also very nondeterministic.
- But to first order, still model communication time as $t_s + t_w m$.
  - t_s and t_m are much smaller in shared memory than distributed memory architectures.

Routing

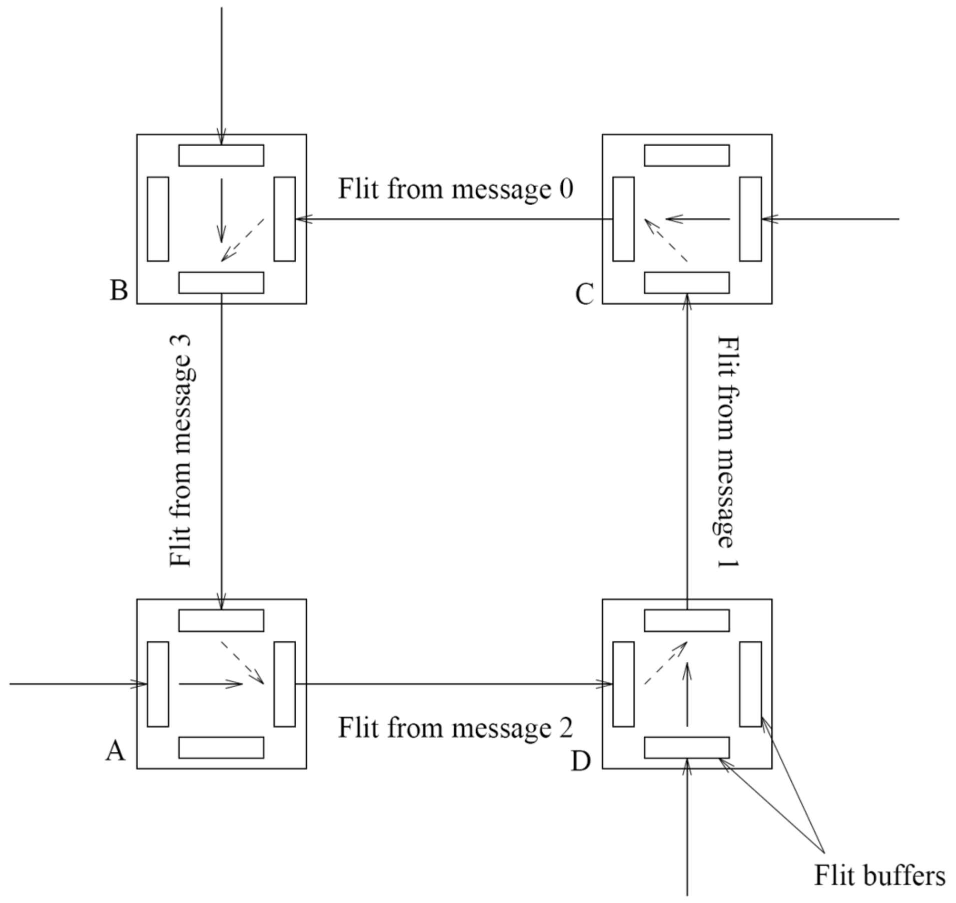
- We saw interconnection networks such as mesh, hypercube, tree, etc. How do processes communicate on these networks?
- Routing algorithm selects a path between two communicating processes.
- Routes are ideally short, uncongested and easy to compute. But tradeoffs exist.
  - Ex Short (minimal) routes may be more congested.
  - Ex Deterministic, i.e. fixed given source and destination, or adaptive, i.e. route around congestion.

Dimension and E-cube routing

- Dimension routing for a k-dim mesh orders the dimensions (e.g. XYZ) and routes messages in order of dimension.
- E-cube routing does dimension routing on dimensions of a hypercube.
  - Let s, d be bit representations of source and destination.
  - Compute r = s XOR d, and from least to most significant digit of r, route along dimensions with 1-bit.
- Ex Routing from 010 to 111.



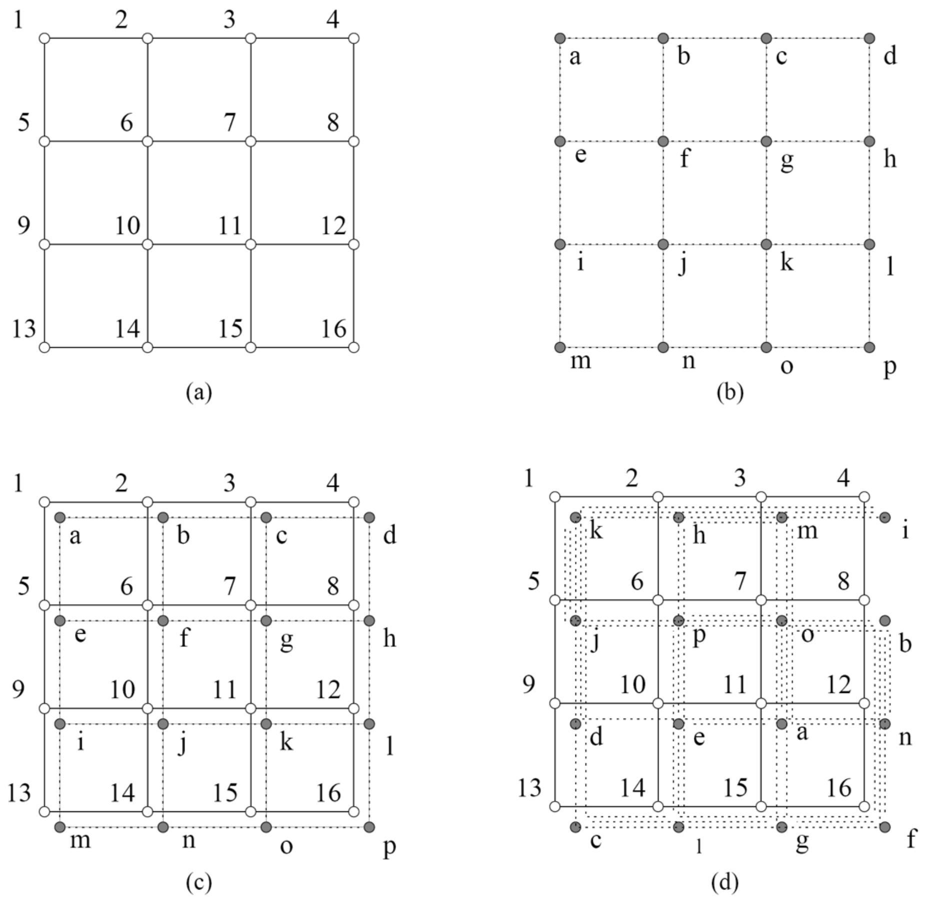
Deadlocks



- Dimension and E-cube routing prevent deadlocks.
- Ex Without dimension routing.
  - A→D→C, D→C→B, C→B→A, B→A→D
  - A's flit can't move from AD to DC, because DC is occupied by D's flit.
  - D's flit can't move from DC to CB, because CB is occupied by C's flit. Etc.
- With XY routing
  - A→D→C, D→A→B, C→B→A, B→C→D.
  - A and C's flits first move along X dim, then B and D's flits.

Process-processor mappings

- Process communication pattern determined by the program.
- Processor communication pattern determined by the hardware.
- To run a program on a piece of hardware, must map processes to processors.
- Poor mappings can cause communication bottlenecks, reduce performance.
- Ex (a) and (b) show processor and process communication patterns.
    - Mapping (c) causes no bottlenecks.
    - Mapping (d) creates bottlenecks at links 15, 23, etc.

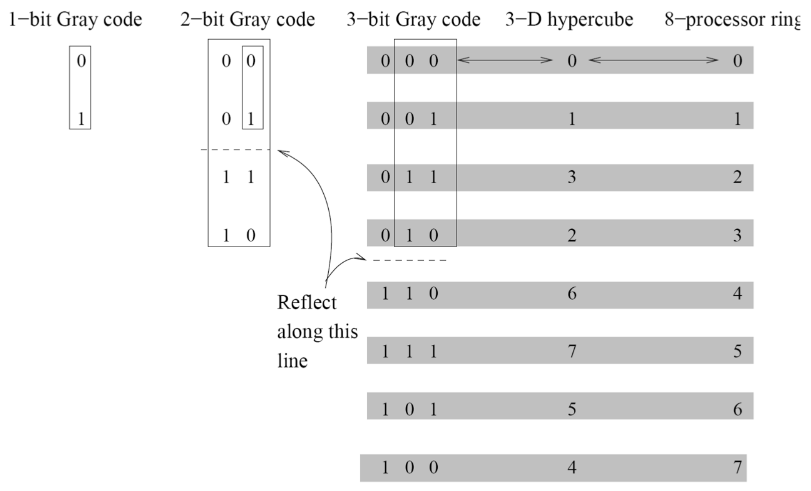


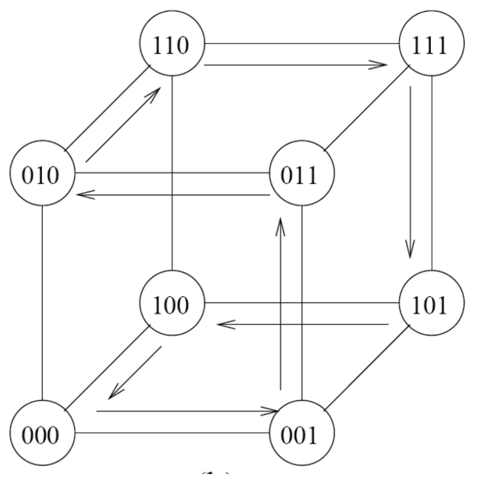
Embeddings

- Give a map f from the graph P of process communication to graph R of processor communication pattern.
  - Each process of P maps to a processor in R.
  - Each edge from p_1 and p_2 in P maps to a shortest path between $f(p_1)$ to $f(p_2)$ in R.
- Congestion is max number of routing paths that cross an edge in R.
  - Ex On previous example congestion = 5, e.g. on edge 15.
- Dilation is max length of route between any two neighbors in P.
  - Ex On previous example dilation = 5, e.g. for edge bc.
- Expansion is (# processes) / (# processors).
  - Assume for simplicity it is 1.

Line to hypercube

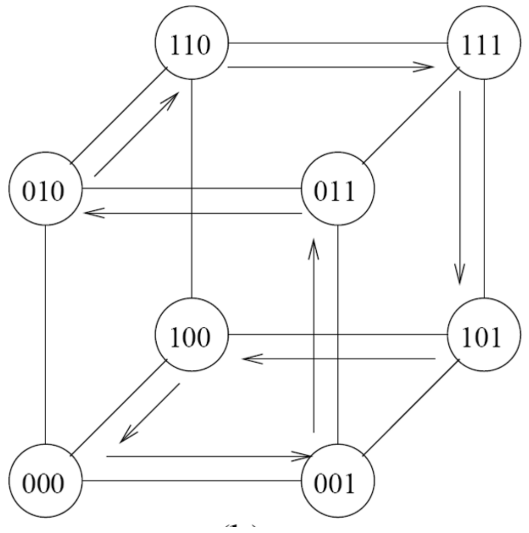
- Map size 2^k line graph into k-dim hypercube.
- Gray codes An ordering of k digit binary numbers where consecutive values differ in one digit.
  - Recursive construction Let G_{k-1} be a k-1 digit Gray code. Prepend G_{k-1} with 0. Then take another G_{k-1}, reverse it and prepend with 1. Concatenate the two copies.
- Map node i in line graph to node $G_k(i)$ in hypercube.
- i and i+1 are mapped to neighbors in hypercube.
  - $G_k(i)$ and $G_k(i+1)$ differ in one bit, so are connected in hypercube.
  - So dilation is 1.
- Each hypercube edge is mapped onto by at most one line graph edge.
  - So congestion is 1.





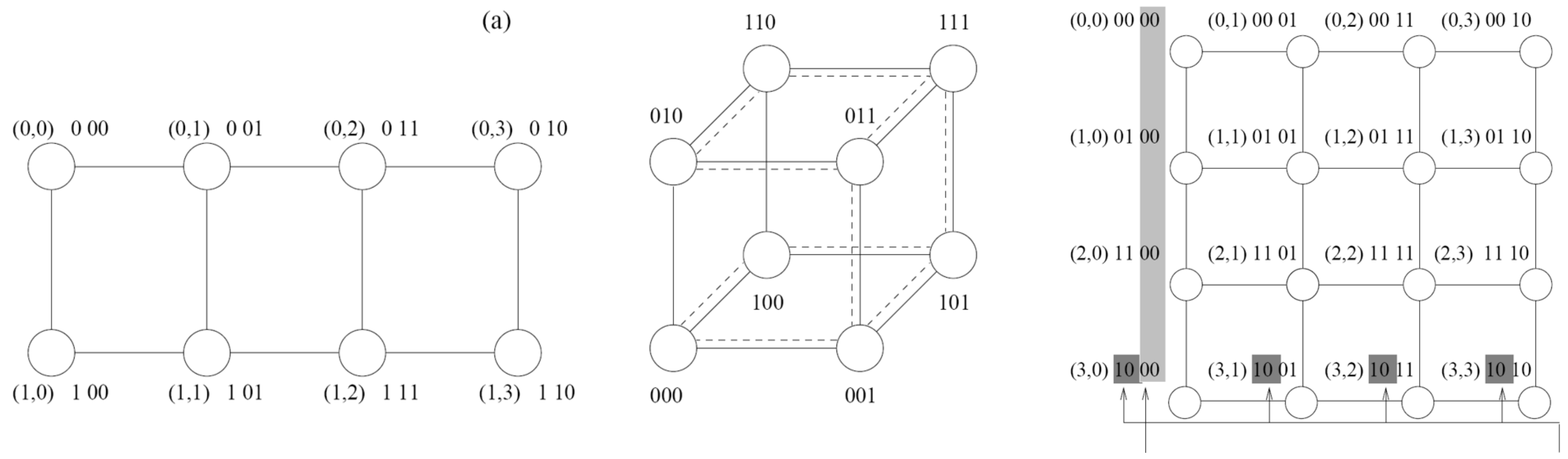
Hypercube to line

- To map k-dim hypercube into size 2^k line grap, just use the reverse mapping, i.e. node i in hypercube maps to $G_k^{-1}(i)$ in line.
- To compute congestion, recall bisection width of k-dim hypercube is 2^{k-1}.
  - Bisection width of line is 1.
  - Thus, 2^{k-1} edges of the hypercube need to cross the middle edge of the line graph, and the congestion is 2^{k-1}.
- Dilation is also 2^k, e.g. from node 0 to node 2^{k-1}.



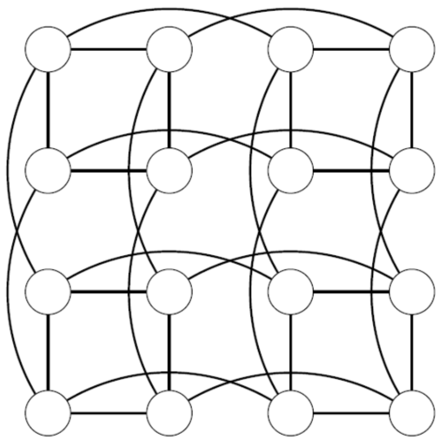
2D mesh to hypercube

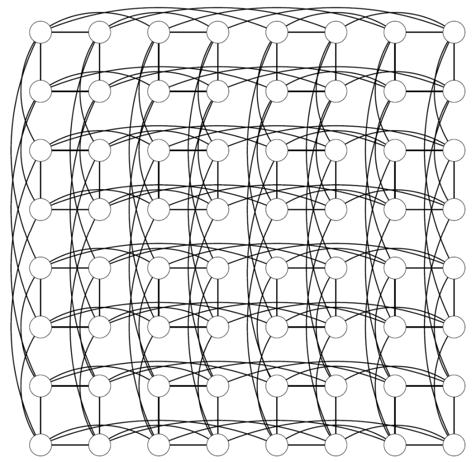
- Map a 2^r x 2^s mesh into an (r +s)-dim hypercube.
- Map node (i, j) in mesh to node $G_r(i)$ || $G_s(j)$ in hypercube.
  - Neighbors in the mesh map to nodes that differ in one bit, i.e. neighbors in the hypercube.
  - So dilation and stretch are both 1.
- Note that different rows and columns of the mesh map to distinct sub-hypercubes.



Hypercube to 2D mesh

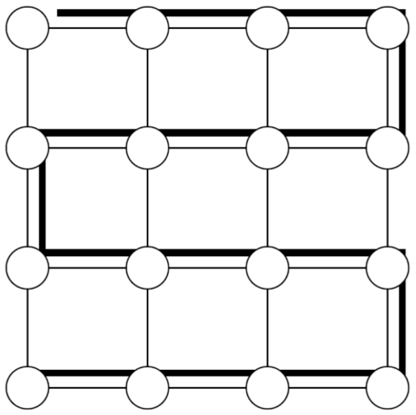
- Map (log *p*)-dim hypercube into √*p* x √*p* mesh.
  - Assume √*p* for simplicity is a power of 2, and let *q* = log √*p* = (log *p*) / 2.
- If we fix last *q* binary digits of the nodes in hypercube, we get another *q*-dim hypercube.
  - Ex Take 4-dim hypercube, and fix last two digits to 10. Then get another hypercube (0010, 0110, 1110, 1010).
- For each *q* digit binary number *r*, map the hypercube from fixing the last *q* digits in hypercube to row *r* of mesh.
  - Use the hypercube to line mapping.
  - Congestion is √*p* / 2.
- For each *q* digit binary number *r*, map the hypercube from fixing the first *q* digits in hypercube to column *r* of mesh.
- Congestion in rows and column independent. So overall congestion is √*p* / 2.

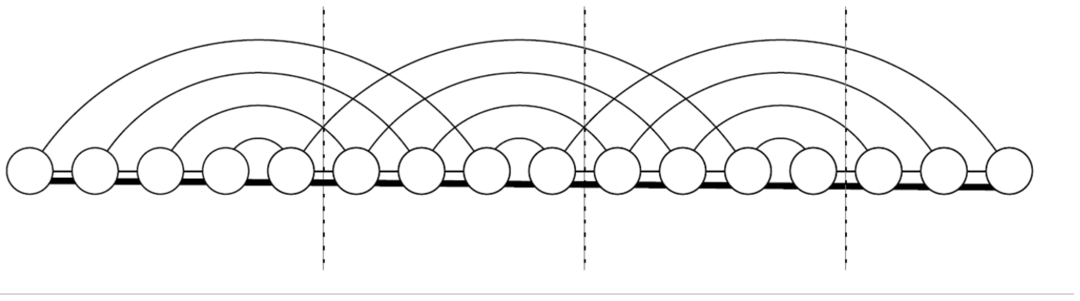




Line and mesh

- Line maps into mesh using zig-zag embedding.
  - Congestion and dilation are 1.
- To map mesh to line, invert the mapping.
- For a $\sqrt{p} \times \sqrt{p}$ mesh, congestion is $\sqrt{p}$.
  - Every two consecutive rows must traverse edge connecting them in the line graph.
- Dilation is $2\sqrt{p} - 1$, between the first nodes in consecutive rows.





Cost performance tradeoffs

- Cost of a network can be measured in terms of e.g. number of wires, bisection width (complexity), etc.
- Wire complexity Square *p* node mesh has with O(log *p*) wires per link has same cost as *p* node hypercube.
- Average distance in mesh is $O(\sqrt{p})$, and in hypercube is O(log *p*).
- With O(log *p*) wires per link in mesh, t_w gets reduced by O(log *p*) factor.
  - Avg latency in mesh is $t_s + O(t_h\sqrt{p}) + O(m\ t_w / \log p)$.
  - Avg latency in hypercube is $t_s + O(t_h \log p) + m\ t_w$.
  - For fixed number of processors and large message size, $m\ t_w >> t_h$, so mesh has lower latency.
- Hypercube better under contention, since it has much greater bisection width.