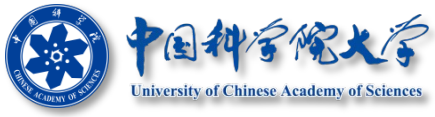
课程编号180086085404P2008H 2024-2025学年秋季学期

计算机体系结构

第9讲 内存一致性、片上网络互联

主讲教师： 刘珂

2024年10月29日



Parallel Programming Model

- Shared-memory:
  - Well-understood programming model
  - Communication is implicit and hardware handles protection
    - 隐式传播write的效果到每个线程，
  - Hardware-controlled caching
  - But need to handle synchronization
    - CC不保证多个指令的操作的原子性，如一个递增操作（increment）涉及到read, modify、write三个指令，CC不能保证这三个步骤最终是原子性。
- Message-passing:
  - No cache coherence -> simpler hardware
  - Explicit communication -> easier for the programmer to restructure code
    - 显式传播write的效果到每个线程
  - Sender can initiate data transfer

Ocean Kernel

```
Procedure Solve(A)
begin
  diff = done = 0;
  while (!done) do
      diff = 0;
      for i ← 1 to n do
        for j ← 1 to n do
          temp = A[i,j];
          A[i,j] ← 0.2 * (A[i,j] + neighbors);
          diff += abs(A[i,j] – temp);
        end for
      end for
      if (diff < TOL) then done = 1;
  end while
end procedure
```

Shared Address Space Model

```
int  n, nprocs;
float  **A, diff;
LOCKDEC(diff_lock);
BARDEC(bar1);


main()
begin
   read(n); read(nprocs);
   A ← G_MALLOC();
   initialize (A);
   CREATE (nprocs,Solve,A);
   WAIT_FOR_END (nprocs);
end main
```

```
procedure Solve(A)
   int i, j, pid, done=0;
   float temp, mydiff=0;
   int mymin = 1 + (pid * n/procs);
   int mymax = mymin + n/nprocs -1;
   while (!done) do
      mydiff = diff = 0;
      BARRIER(bar1,nprocs);
      for i ← mymin to mymax
         for j ← 1 to n do
            …
         endfor
      endfor
      LOCK(diff_lock);
      diff += mydiff;
      UNLOCK(diff_lock);
      BARRIER (bar1, nprocs);
      if (diff < TOL) then done = 1;
      BARRIER (bar1, nprocs);
   endwhile
```

Message Passing Model

```
main()
  read(n); read(nprocs);
  CREATE (nprocs-1, Solve);
  Solve();
  WAIT_FOR_END (nprocs-1);

procedure Solve()
  int i, j, pid, nn = n/nprocs, done=0;
  float temp, tempdiff, mydiff = 0;
  myA ← malloc(…)
  initialize(myA);
  while (!done) do
    mydiff = 0;
    if (pid != 0)
      SEND(&myA[1,0], n, pid-1, ROW);
    if (pid != nprocs-1)
      SEND(&myA[nn,0], n, pid+1, ROW);
    if (pid != 0)
      RECEIVE(&myA[0,0], n, pid-1, ROW);
    if (pid != nprocs-1)
      RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
    for i ← 1 to nn do
      for j ← 1 to n do
        …
      endfor
    endfor
    if (pid != 0)
      SEND(mydiff, 1, 0, DIFF);
      RECEIVE(done, 1, 0, DONE);
    else
      for i ← 1 to nprocs-1 do
        RECEIVE(tempdiff, 1, *, DIFF);
        mydiff += tempdiff;
      endfor
      if  (mydiff < TOL)  done = 1;
      for i ← 1 to nprocs-1  do
        SEND(done, 1, I, DONE);
      endfor
    endif
  endwhile
```

Synchronization

Constructing Locks

- Applications have phases (consisting of many instructions) that must be executed atomically, without other parallel processes modifying the data
- A lock surrounding the data/code ensures that only one program can be in a critical section at a time
- The hardware must provide some basic primitives that allow us to construct locks with different properties
- Lock algorithms assume an underlying cache coherence mechanism – when a process updates a lock, other processes will eventually see the update

Synchronization

- The simplest hardware primitive that greatly facilitates synchronization implementations (locks, barriers, etc.) is an atomic read-modify-write
- Atomic exchange: swap contents of register and memory
- Special case of atomic exchange: test & set: transfer memory location into register and write 1 into memory
- lock: t&s register, location
  bnz register, lock
  CS
  st location, #0

Caching Locks

- Spin lock: to acquire a lock, a process may enter an infinite loop that keeps attempting a read-modify till it succeeds

- If the lock is in memory, there is heavy bus traffic => other processes make little forward progress

- Locks can be cached:
  - Cache coherence ensures that a lock update is seen by other processors
  - The process that acquires the lock in exclusive state gets to update the lock first
  - Spin on a local copy – the external bus sees little traffic

Coherence Traffic for a Lock

- If every process spins on an exchange, every exchange instruction will attempt a write
  - many invalidates and the locked value keeps changing ownership
- Hence, each process keeps reading the lock value – a read does not generate coherence traffic and every process spins on its locally cached copy
- When the lock owner releases the lock by writing a 0,
  - other copies are invalidated, each spinning process generates a read miss, acquires a new copy, sees the 0, attempts an exchange (**requires acquiring the block in exclusive state so the write can happen**),
  - first process to acquire the block in exclusive state acquires the lock, others keep spinning

Test-and-Test-and-Set

```
lock:   test    register, location
        bnz     register, lock
        t&s     register, location
        bnz     register, lock
        CS
        st      location, #0
```

Load-Linked and Store Conditional

- LL-SC is an implementation of atomic read-modify-write with very high flexibility
  - Atomic exchange is an atomic unit/instruction, but LL-SC does not require the whole unit to be atomic
  - LL and SC 之间不要求原子性，更加灵活
- LL: read a value and update a table indicating you have read this address, then perform any amount of computation
- SC: attempt to store a result into the same memory location, the store will succeed only if the table indicates that no other process attempted a store since the local LL (success only if the operation was "effectively" atomic)
- SC implementations do not generate bus traffic if the SC fails – hence, more efficient than test&test&set
  - 更少的CC 通讯数据，硬件开销低

Spin Lock with Low Coherence Traffic

```
lockit:  LL      R2, 0(R1)   ; load linked, generates no coherence traffic
         BNEZ    R2, lockit    ; not available, keep spinning
         DADDUI R2, R0, #1 ; put value 1 in R2
         SC      R2, 0(R1)   ; store-conditional succeeds if no one
                             ; updated the lock since the last LL
         BEQZ    R2, lockit    ; confirm that SC succeeded, else keep trying
```

- If there are i processes waiting for the lock, how many bus transactions happen?

Spin Lock with Low Coherence Traffic

```
lockit:  LL      R2, 0(R1)    ; load linked, generates no coherence traffic
         BNEZ    R2, lockit    ; not available, keep spinning
         DADDUI R2, R0, #1 ; put value 1 in R2
         SC      R2, 0(R1)   ; store-conditional succeeds if no one
                             ; updated the lock since the last LL
         BEQZ    R2, lockit    ; confirm that SC succeeded, else keep trying
```

- If there are i processes waiting for the lock, how many bus transactions happen?
- 1 write by the releaser + i read-miss requests + i (or 1) responses + 1 write by acquirer + 0 (i-1 failed SCs) + i-1 read-miss requests + i-1 (or 1) responses
  - (The i and i-1 read misses and responses can be reduced to 1)

Further Reducing Bandwidth Needs

- Ticket lock: every arriving process atomically picks up a ticket and increments the ticket counter (with an LL-SC), the process then keeps checking the now-serving variable to see if its turn has arrived, after finishing its turn it increments the now-serving variable
- Array-Based lock: instead of using a "now-serving" variable, use a "now-serving" array and each process waits on a different variable – fair, low latency, low bandwidth, high scalability, but higher storage
- Queueing locks: the directory controller keeps track of the order in which requests arrived – when the lock is available, it is passed to the next in line (only one process sees the invalidate and update)

Memory Consistence Model

Coherence Vs. Consistency

- Recall that coherence guarantees (i) that a write will eventually be seen by other processors, and (ii) write serialization (all processors see writes to the same location in the same order)
  - 每个core都遵守缓存一致性，如果没有顺序一致性，不同核心上的线程可能看到的指令顺序可能会不同，这会导致数据不一致和竞态条件 race condition

- The consistency model defines the ordering of writes and reads to different memory locations
  - The hardware guarantees a certain consistency model
  - The programmer attempts to write correct programs with those assumptions

Example Programs

Initially, A = B = 0

P1	P2
A = 1	B = 1
if (B == 0)	if (A == 0)
critical section	critical section

Initially, Head = Data = 0

P1	P2
Data = 2000	while (Head == 0)
Head = 1	{ }
	… = Data

Initially, A = B = 0

P1	P2	P3
A = 1		
	if (A == 1)	
	B = 1	
		if (B == 1)
		register = A

Sequential Consistency

P1	P2
Instr-a	Instr-A
Instr-b	Instr-B
Instr-c	Instr-C
Instr-d	Instr-D
…	…

We assume:

- Within a program, program order is preserved
- Each instruction executes atomically
- Instructions from different threads can be interleaved arbitrarily

Valid executions:

abAcBCDdeE… or ABCDEFabGc… or abcAdBe… or
aAbBcCdDeE… or …..

Sequential Consistency

- Programmers assume SC; makes it much easier to reason about program behavior
- Hardware innovations can disrupt the SC model
- For example, if we assume write buffers, or out-of-order execution, or if we drop ACKS in the coherence protocol, the previous programs yield unexpected outputs

Consistency Example - I

- An ooo core will see no dependence between instructions dealing with A and instructions dealing with B; those operations can therefore be re-ordered; this is fine for a single thread, but not for multiple threads

```
Initially A = B = 0
  P1                    P2
A ← 1                 B ← 1
 …                     …
if (B == 0)           if (A == 0)
  Crit.Section          Crit.Section
```

The consistency model lets the programmer know what assumptions they can make about the hardware's reordering capabilities

Consistency Example - 2

Initially, A = B = 0

P1	P2	P3
A = 1		
	if (A == 1) B = 1	
		if (B == 1) register = A

If a coherence invalidation didn't require ACKs, we can't confirm that everyone has seen the value of A.

Sequential Consistency

- A multiprocessor is sequentially consistent if the result of the execution is achievable by maintaining program order within a processor and interleaving accesses by different processors in an arbitrary fashion

- Can implement sequential consistency by requiring the following: program order, write serialization, everyone has seen an update before a value is read – very intuitive for the programmer, but extremely slow

- This is very slow… alternatives:
  - Add optimizations to the hardware (e.g., verify loads)
  - Offer a relaxed memory consistency model and fences

Relaxed Consistency Models

- We want an intuitive programming model (such as sequential consistency) and we want high performance
- We care about data races and re-ordering constraints for some parts of the program and not for others – hence, we will relax some of the constraints for sequential consistency for most of the program, but enforce them for specific portions of the code
- Fence instructions are special instructions that require all previous memory accesses to complete before proceeding (sequential consistency)

Fences

```
P1
{
  Region of code
  with no races
}

Fence
Acquire_lock
Fence

{
   Racy code
}

Fence
Release_lock
Fence
```

```
P2
{
Region of code
 with no races
}

Fence
Acquire_lock
Fence

{
Racy code
}

Fence
Release_lock
Fence
```

Transaction Memory

Transactions

- Lock is troublesome, prone to cause deadlock
  - Hardware architects always want to simply programming
- TM: New paradigm to simplify programming
  - Instead of lock-unlock, use transaction begin-end
  - locks are blocking, transactions execute speculatively in the hope that there will be no conflicts
- Can yield better performance; Eliminates deadlocks
- Programmer can freely encapsulate code sections within transactions and not worry about the impact on performance and correctness (for the most part)

Example

- Producer-consumer relationships – producers place tasks at the tail of a work-queue and consumers pull tasks out of the head

```
Enqueue
  transaction begin
     if (tail == NULL)
       update head and tail
     else
       update tail
  transaction end
```

```
Dequeue
    transaction begin
        if (head->next == NULL)
         update head and tail
         else
           update head
    transaction end
```

With locks, neither thread can proceed in parallel since head/tail may be updated – with transactions, enqueue and dequeue can proceed in parallel – transactions will be aborted only if the queue is nearly empty

Transactions

- Transactional semantics:
  - when a transaction executes, it is as if the rest of the system is suspended and the transaction is in isolation
  - the reads and writes of a transaction happen as if they are all a single atomic operation
  - if the above conditions are not met, the transaction fails to commit (abort) and tries again

```
transaction begin
    read shared variables
    arithmetic
    write shared variables
transaction end
```

TM Implementation

- Caches track read-sets and write-sets
- Writes are made visible only at the end of the transaction
- At transaction commit, make your writes visible; others may abort

Detecting Conflicts – Basic Implementation

- Writes can be cached (can't be written to memory) – if the block needs to be evicted, flag an overflow (abort transaction for now) – on an abort, invalidate the written cache lines
- Keep track of read-set and write-set (bits in the cache) for each transaction
- When another transaction commits, compare its write set with your own read set – a match causes an abort
- At transaction end, express intent to commit, broadcast write-set (transactions can commit in parallel if their write-sets do not intersect)

Summary of TM Benefits

- As easy to program as coarse-grain locks
- Performance similar to fine-grain locks
- Avoids deadlock

Design Space

- Data Versioning
  - Eager: based on an undo log
  - Lazy: based on a write buffer

- Conflict Detection
  - Optimistic detection: check for conflicts at commit time (proceed optimistically thru transaction)
  - Pessimistic detection: every read/write checks for conflicts (reduces work during commit)

“Lazy” Implementation

- An implementation for a small-scale multiprocessor with a snooping-based protocol
- Lazy versioning and lazy conflict detection
  - Optimistic on the conflict can be resolve eventually
  - Lazy to handle conflict
- Does not allow transactions to commit in parallel

“Lazy” Implementation

- When a transaction issues a read, fetch the block in read-only mode (if not already in cache) and set the rd-bit for that cache line
- When a transaction issues a write, fetch that block in read-only mode (if not already in cache), set the wr-bit for that cache line and make changes in cache
- If a line with wr-bit set is evicted, the transaction must be aborted (or must rely on some software mechanism to handle saving overflowed data)

“Lazy” Implementation

- When a transaction reaches its end, it must now make its writes permanent
  - 最后才提交事务，让其持久化

- A central arbiter is contacted (easy on a bus-based system), the winning transaction holds on to the bus until all written cache line addresses are broadcasted (this is the commit) (need not do a writeback until the line is evicted – must simply invalidate other readers of these cache lines)

- When another transaction (that has not yet begun to commit) sees an invalidation for a line in its rd-set, it realizes its lack of atomicity and aborts (clears its rd- and wr-bits and re-starts)

“Lazy” Implementation

- Lazy versioning: changes are made locally – the “master copy” is updated only at the end of the transaction
- Lazy conflict detection: we are checking for conflicts only when one of the transactions reaches its end
- Aborts are quick (must just clear bits in cache, flush pipeline and reinstate a register checkpoint)

- Commit is slow (must check for conflicts, all the coherence operations for writes are deferred until transaction end)
- **No fear of deadlock** – the first transaction to acquire the bus will commit successfully
- Starvation is possible – need additional mechanisms

"Eager" Implementation

- A write is made permanent immediately (we do not wait until the end of the transaction)
- This means that if some other transaction attempts a read, the latest value is returned and the memory may also be updated with this latest value

- Can't lose the old value (in case this transaction is aborted) – hence, before the write, we copy the old value into a log (the log is some space in virtual memory
  - -- the log itself may be in cache, so not too expensive)
- This is eager versioning

"Eager" Implementation

- Since Transaction-A's writes are made permanent right away, it is possible that another Transaction-B's rd/wr miss is re-directed to Tr-A

- At this point, we detect a conflict (neither transaction has reached its end, hence, eager conflict detection): two transactions handling the same cache line and at least one of them does a write

- One solution: requester stalls: Tr-A sends a NACK to Tr-B; Tr-B waits and re-tries again; hopefully, Tr-A has committed and can hand off the latest cache line to B
  - Neither transaction needs to abort

“Eager” Implementation

- Can lead to deadlocks: each transaction is waiting for the other to finish

- Need a separate (hw/sw) contention manager to detect such deadlocks and force one of them to abort

Tr-A	Tr-B
write X	write Y
…	…
read Y	read X

"Eager" Implementation

- Note that if Tr-B is doing a write, it may be forced to stall because Tr-A may have done a read and does not want to invalidate its cache line just yet
- If new reading transactions keep emerging, Tr-B may be starved – again, need other sw/hw mechanisms to handle starvation

- Since logs are stored in virtual memory, there is no cache overflow problem and transactions can be large

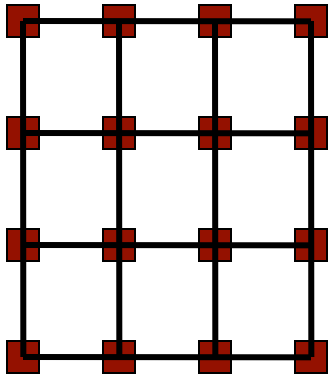
- Commits are inexpensive (no additional step required); Aborts are expensive, but rare (must reinstate data from logs)

Discussion

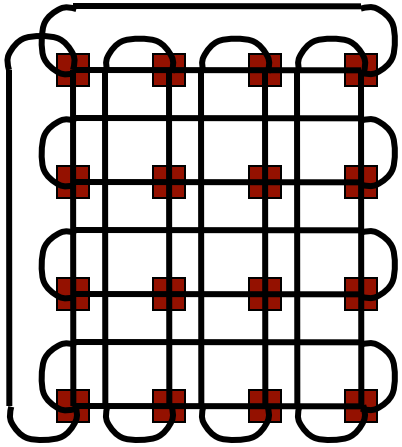
- "Eager" optimizes the common case and does not waste energy when there's a potential conflict
- TM implementations require relatively low hardware support
- Multiple commercial examples: Sun Rock, AMD ASF, IBM BG/Q, Intel Haswell
- Transactions are often short – more than 95% of them will fit in cache
- Transactions often commit successfully – less than 10% are aborted
- 99.9% of transactions don't perform I/O
- Amdahl's Law again: optimize the common case!

Interconnection

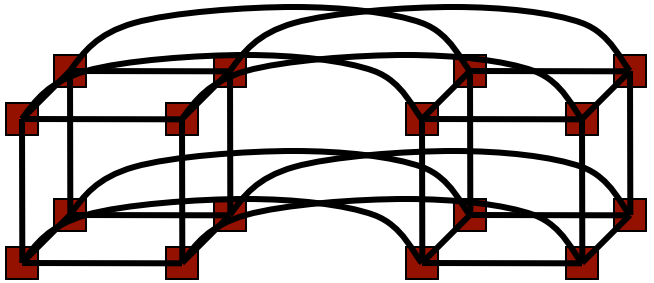
Network Topology Examples



Grid



Torus



Hypercube

Routing

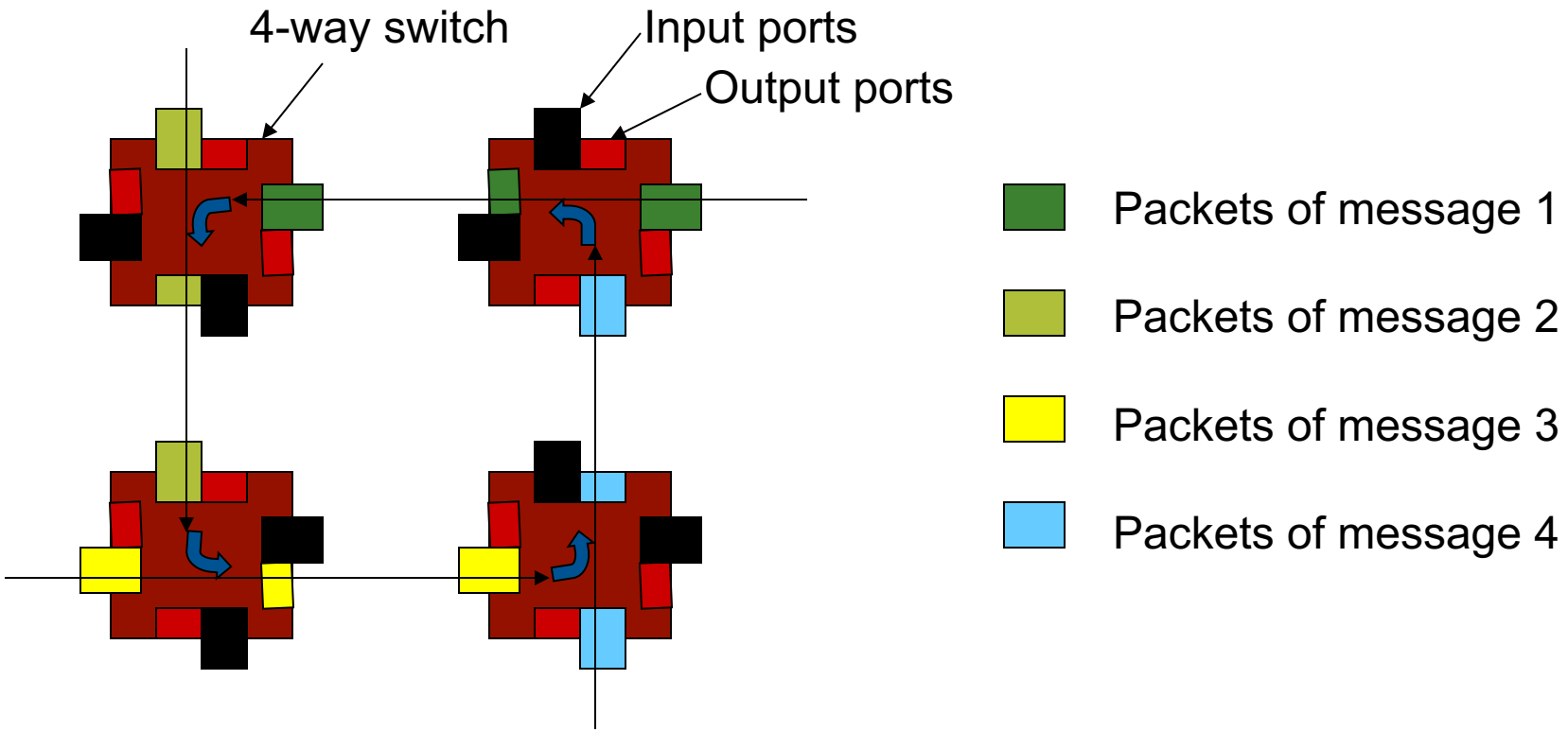
- Deterministic routing: given the source and destination, there exists a unique route

- Adaptive routing: a switch may alter the route in order to deal with unexpected events (faults, congestion) – more complexity in the router vs. potentially better performance

- Example of deterministic routing: dimension order routing: send packet along first dimension until destination co-ord (in that dimension) is reached, then next dimension, etc.

Deadlock Example

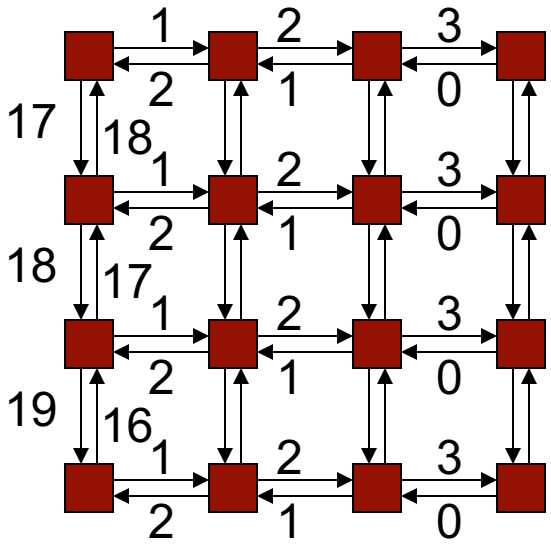
- Deadlock happens when there is a cycle of resource dependencies – a process holds on to a resource (A) and attempts to acquire another resource (B) – A is not relinquished until B is acquired



Each message is attempting to make a left turn – it must acquire an output port, while still holding on to a series of input and output ports

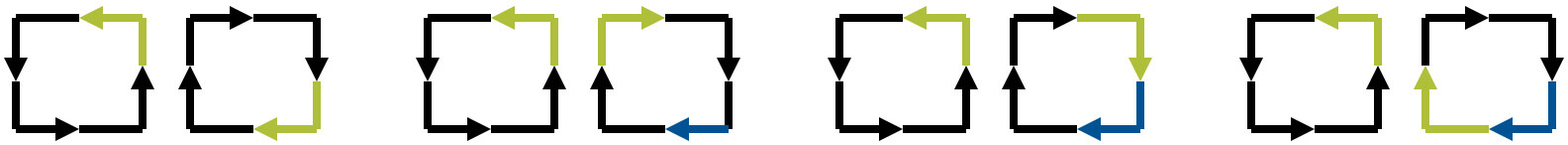
Deadlock-Free Proofs

- Number edges and show that all routes will traverse edges in increasing (or decreasing) order – therefore, it will be impossible to have cyclic dependencies
- Example: k-ary 2-d array with dimension routing: first route along x-dimension, then along y



Breaking Deadlock

- Consider the eight possible turns in a 2-d array (note that turns lead to cycles)
- By preventing just two turns, cycles can be eliminated
- Dimension-order routing disallows four turns
- Helps avoid deadlock even in adaptive routing



West-First | North-Last | Negative-First | Can allow deadlocks

Packets/Flits

- A message is broken into multiple packets (each packet has header information that allows the receiver to re-construct the original message)
- A packet may itself be broken into flits – flits do not contain additional headers
- Two packets can follow different paths to the destination Flits are always ordered and follow the same path
- Such an architecture allows the use of a large packet size (low header overhead) and yet allows fine-grained resource allocation on a per-flit basis

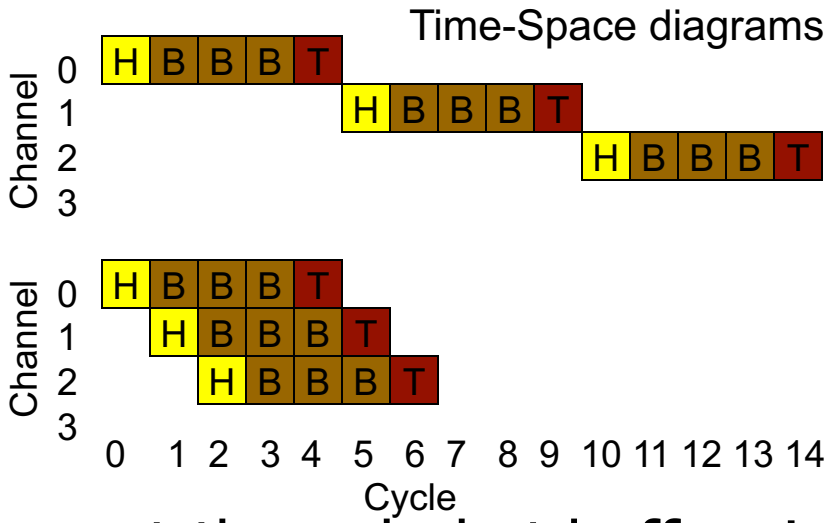
Flow Control

- The routing of a message requires allocation of various resources: the channel (or link), buffers, control state
- Bufferless: flits are dropped if there is contention for a link, NACKs are sent back, and the original sender has to re-transmit the packet
- Circuit switching: a request is first sent to reserve the channels, the request may be held at an intermediate router until the channel is available (hence, not truly bufferless), ACKs are sent back, and subsequent packets/flits are routed with little effort (good for bulk transfers)

Buffered Flow Control

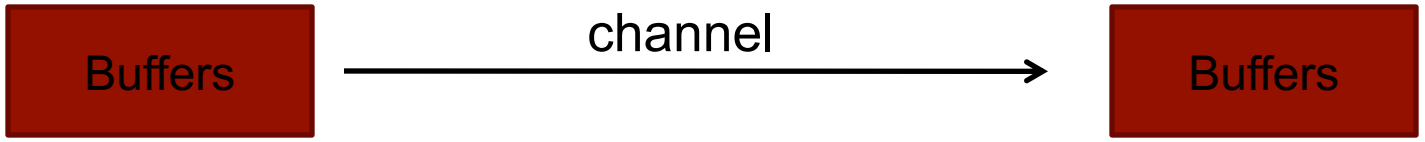
- A buffer between two channels decouples the resource allocation for each channel

- Packet-buffer flow control: channels and buffers are allocated per packet
  - Store-and-forward
  - Cut-through



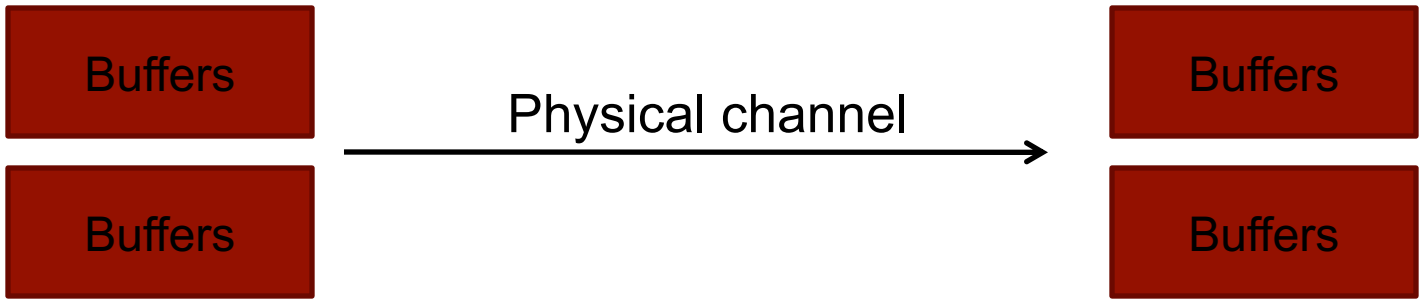
- Wormhole routing: same as cut-through, but buffers in each router are allocated on a per-flit basis, not per-packet

Virtual Channels



Flits do not carry headers. Once a packet starts going over a channel, another packet cannot cut in (else, the receiving buffer will confuse the flits of the two packets). If the packet is stalled, other packets can't use the channel.

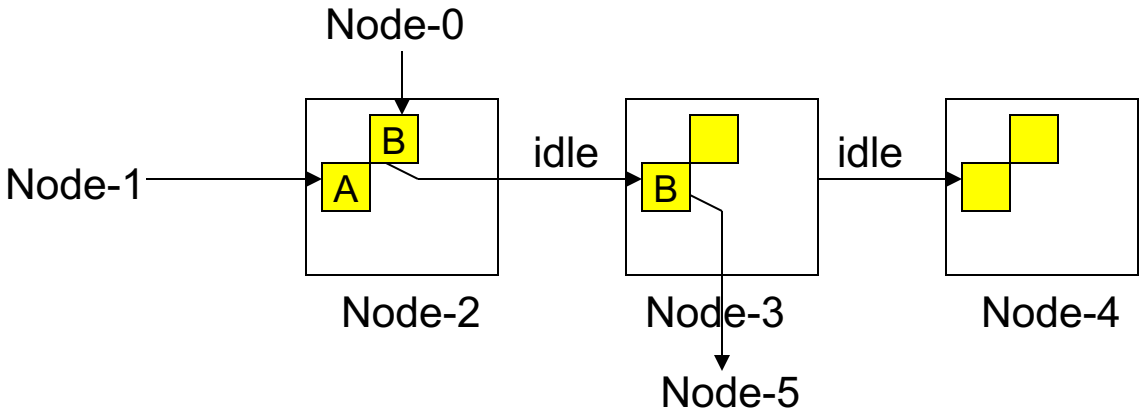
With virtual channels, the flit can be received into one of N buffers. This allows N packets to be in transit over a given physical channel. The packet must carry an ID to indicate its virtual channel.



Example

- Wormhole:

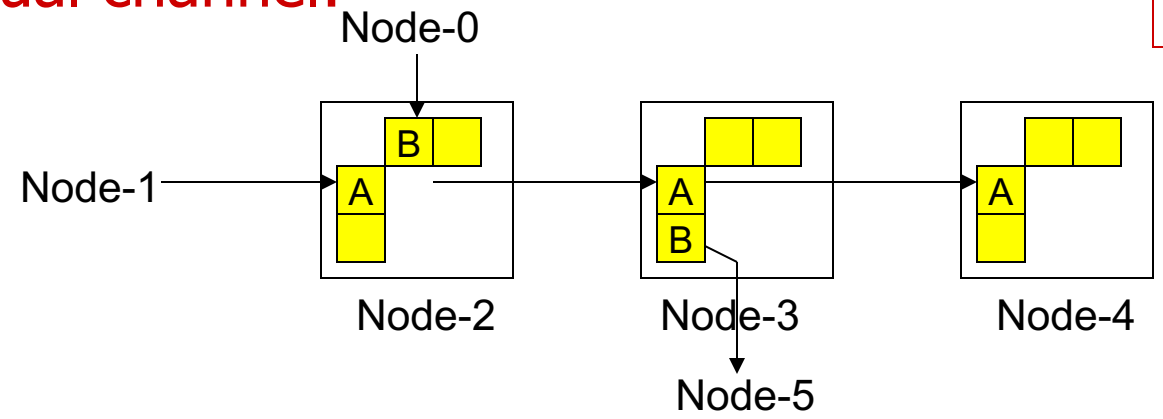
A is going from Node-1 to Node-4; B is going from Node-0 to Node-5



(blocked, no free VCs/buffers)

Traffic Analogy:
B is trying to make a left turn; A is trying to go straight; there is no left-only lane with wormhole, but there is one with VC

- Virtual channel:



(blocked, no free VCs/buffers)

Virtual Channel Flow Control

- Incoming flits are placed in buffers

- For this flit to jump to the next router, it must acquire three resources:
  - A free virtual channel on its intended hop
    - We know that a virtual channel is free when the tail flit goes through
  - Free buffer entries for that virtual channel
    - This is determined with credit or on/off management
  - A free cycle on the physical channel
    - Competition among the packets that share a physical channel

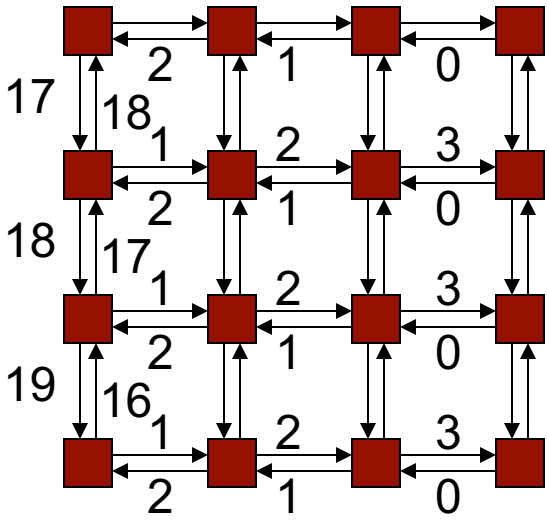
Buffer Management

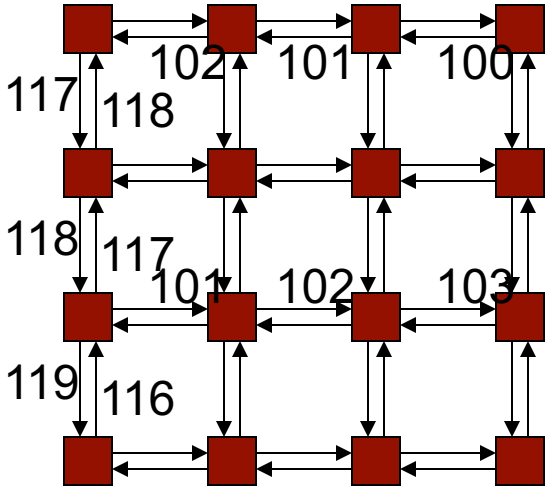
- Credit-based: keep track of the number of free buffers in the downstream node; the downstream node sends back signals to increment the count when a buffer is freed; need enough buffers to hide the round-trip latency

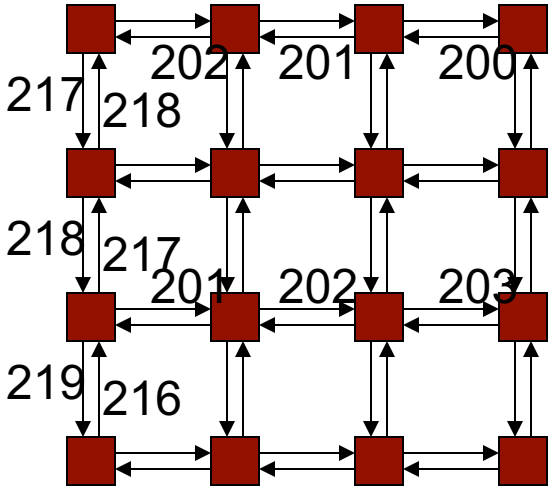
- On/Off: the upstream node sends back a signal when its buffers are close to being full – reduces upstream signaling and counters, but can waste buffer space

Deadlock Avoidance with VCs

- VCs provide another way to number the links such that a route always uses ascending link numbers







- Alternatively, use West-first routing on the 1st plane and cross over to the 2nd plane in case you need to go West again (the 2nd plane uses North-last, for example)

Acknowledgements

- EPFL, Onur Mutlu, Digital Design and Computer Architecture, 2020, 2023
- Utah University, Rajeev, CS/ECE 6810, Computer Architecture
- 计算机体系结构:量化研究方法(中文版第六版)
- UCSD CSE 240
- Washington University, CSE3 78
- 国科大，胡伟武，计算机体系结构
- CMU, CS 447 Computer Architecture
- UC Berkeley, CS 152 and CS 61C
- 国科大南京学院，安学军等，计算机体系结构
- 浙江大学，计算机体系结构
- 南京大学，计算机体系结构