Advanced Computer Networks

263-3501-00

Flow Control

Patrick Stuedi

Spring Semester 2017

Last week

- TCP in Datacenters
  - Avoid incast problem
    - Reduce RTO_MIN
  - Avoid buffer overflow
    - DCTCP: Proportional window decrease
  - Make sure flows meet their deadlines
    - D3: Allocate deadline-proportional bandwidth in routers
    - D2TCP: Deadline proportional variation of sender rate
  - Use all the available bandwidth in Fat Trees
    - Multipath TCP

Today

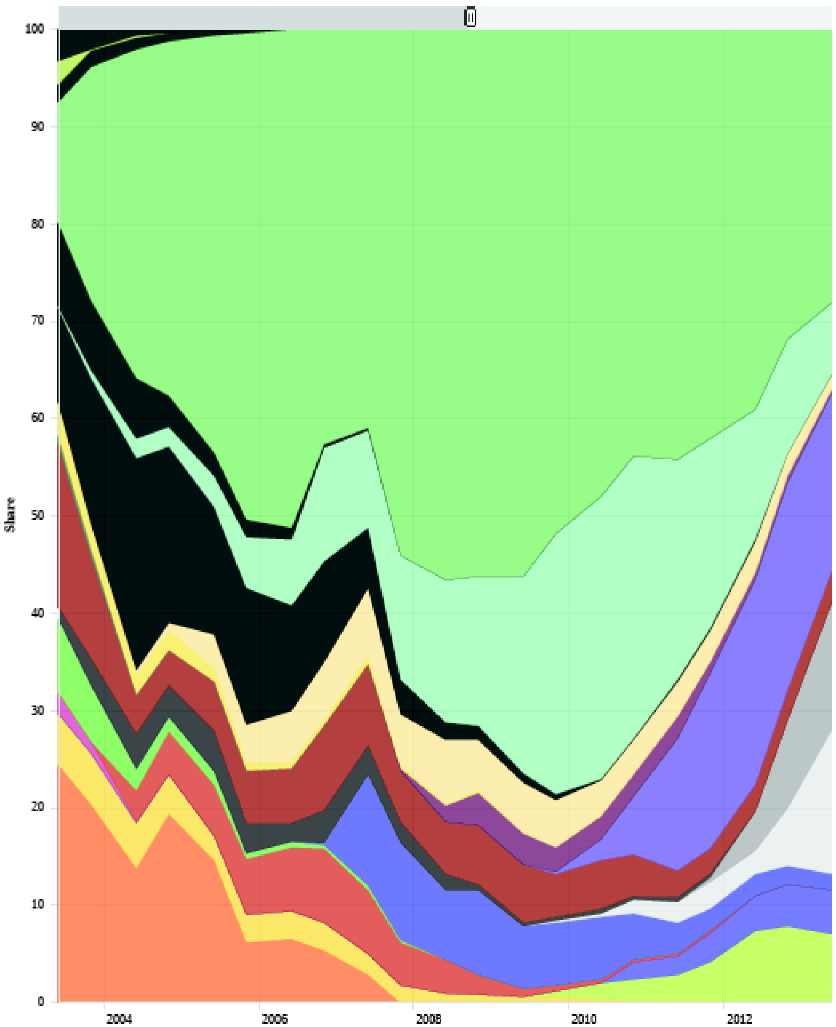
- Flow Control
  - Store-and-forward, cut-through, wormhole
  - Head-of-line blocking
- Infiniband
- Lossless Ethernet
- Flow coordination

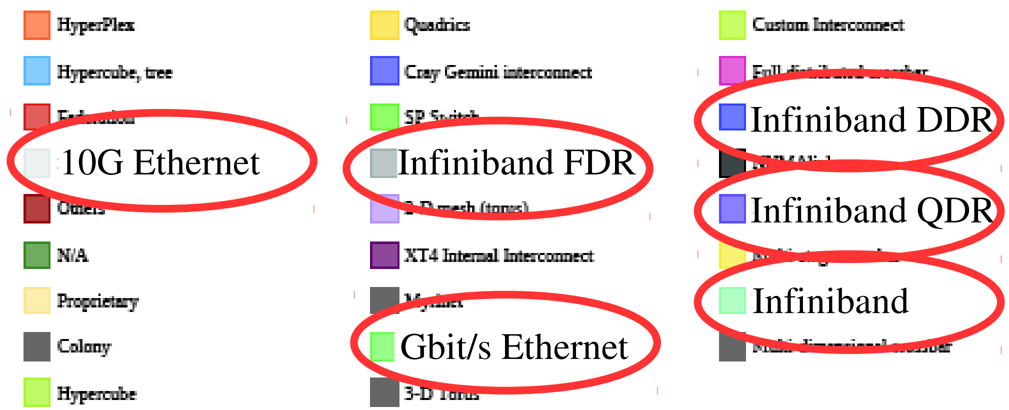
Flow Control Basics

Where to best put flow control

- So far we have discussed the TCP/IP/Ethernet stack
  - TCP flow-control: avoid receiver buffer overflow
  - TCP congestion-control: avoid switch buffer overflow
- TCP's congestion-control is reactive
  - First loose packets, then adjust data rate
- Is reactive congestion-control a good choice at
  - 1 Gbit/s data rate?
  - 10 Gbit/s data rate?
  - 100 Gbit/s data rate?

Supercomputer interconnect technologies (2003-2014)

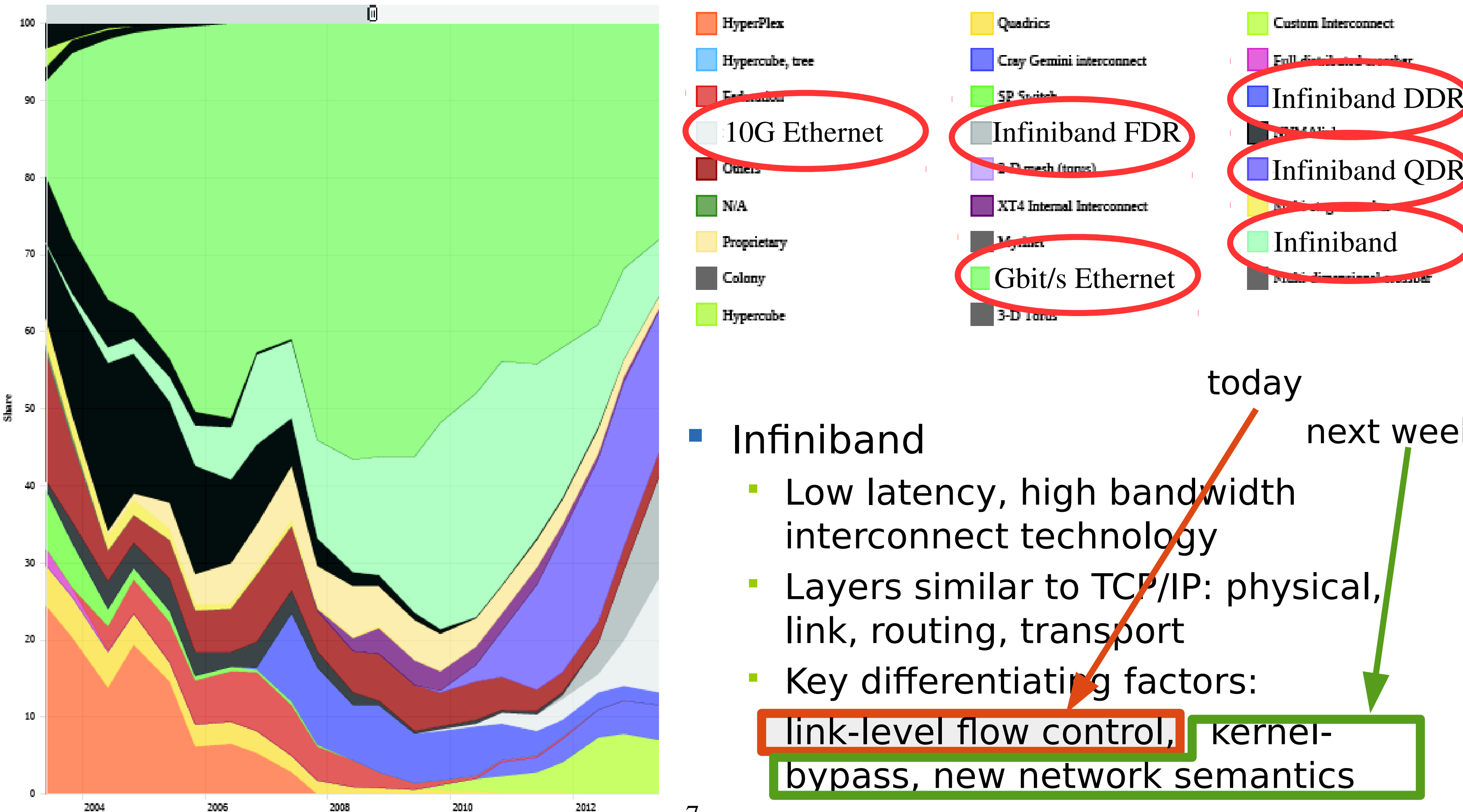




- Infiniband
  - Low latency, high bandwidth interconnect technology
  - Layers similar to TCP/IP: physical, link, routing, transport
  - Key differentiating factors:

    link-level flow control, kernel-bypass, new network semantics

Supercomputer interconnect technologies (2003-2014)

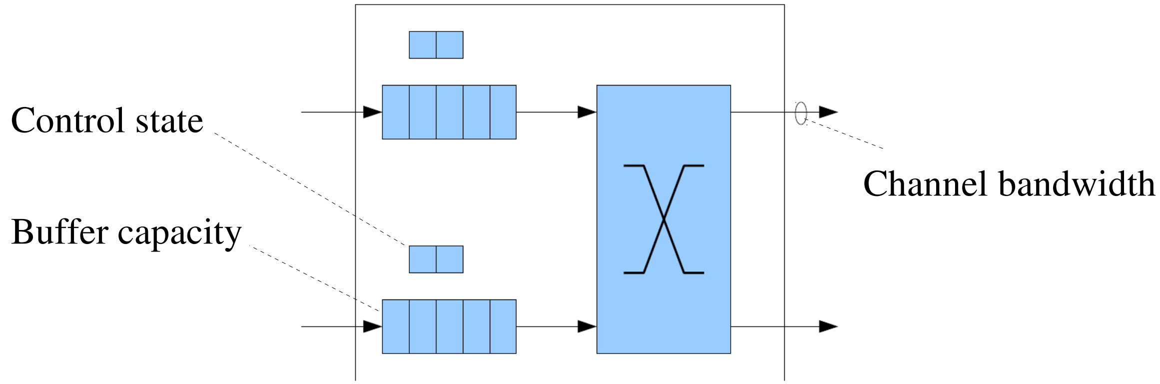


today

next week

- Infiniband
  - Low latency, high bandwidth interconnect technology
  - Layers similar to TCP/IP: physical, link, routing, transport
  - Key differentiating factors: link-level flow control, kernel-bypass, new network semantics

Flow control

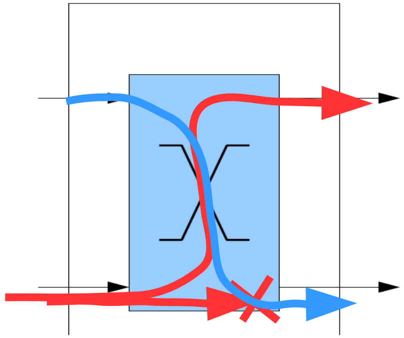


- Determines how a network's resources are allocated to packets traversing the network
- Flow-control network resources:
  - Channel bandwidth
  - Buffer capacity
- Goal:
  - Allocate resources in an efficient manner to achieve a high fraction of the network's ideal bandwidth
  - Deliver packets with low, predictable latency

Flow control mechanisms

- Bufferless:
  - Drop or misroute packets if outgoing channel at switch is blocked

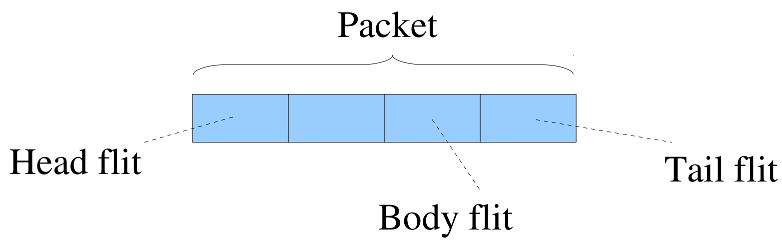
red packet cannot be forward on requested channel because channel is occupied by the **blue** packet



route **red** packet on uppper channel instead and re-route it later

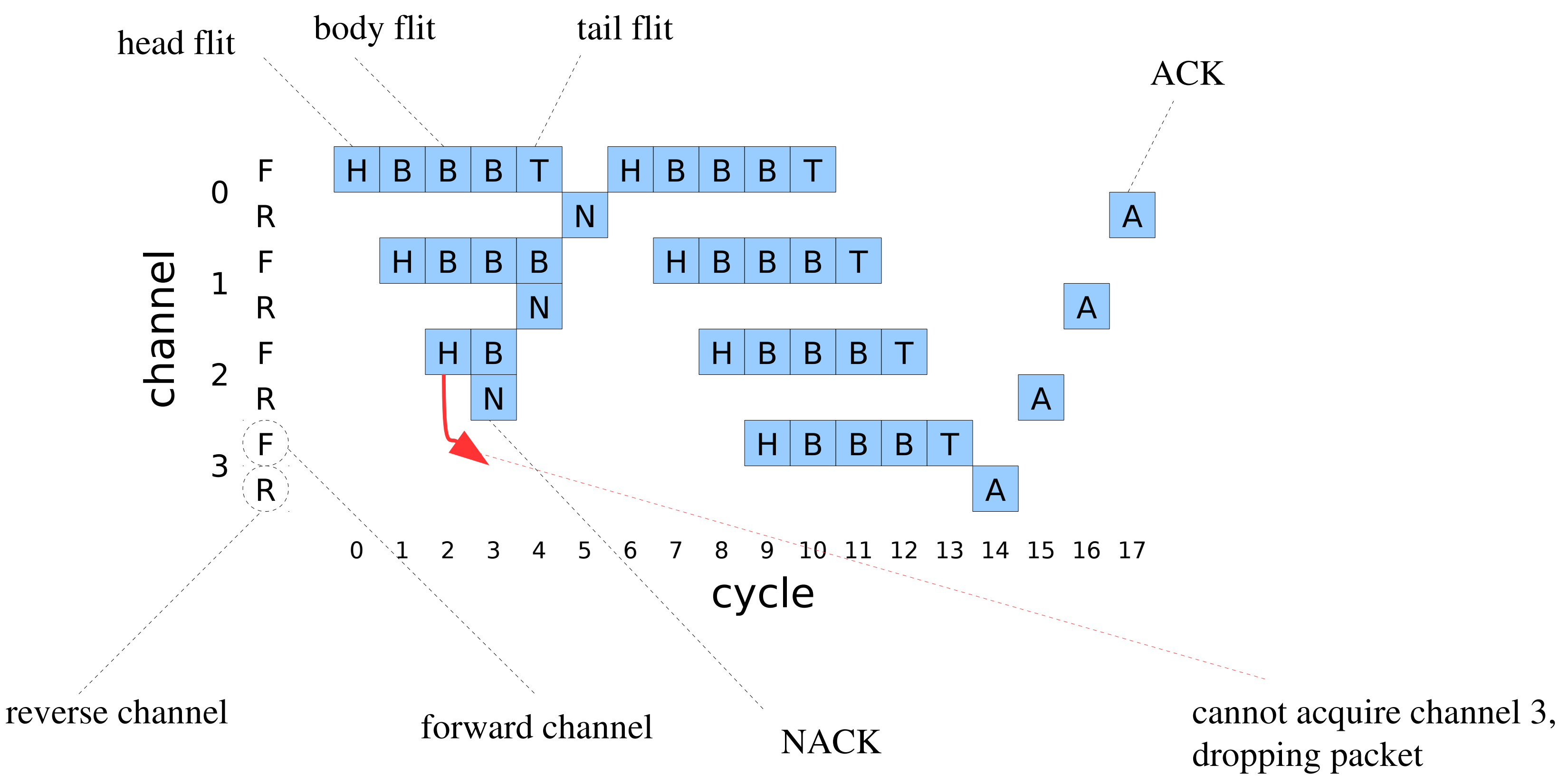
- Circuit switching:
  - Only headers are buffered and reserve path
  - Pause header forwarding if path is not available
- Buffered:
  - Decouples channel allocation in time

Flits

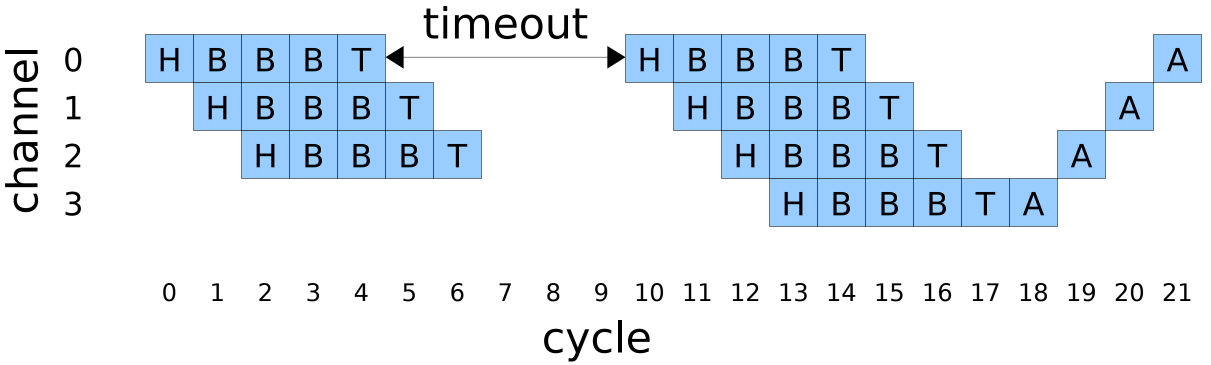


- Packets are MTU-sized
  - Typically several 1000 bytes
  - Switch buffering and forwarding often works at a smaller granularity (several bytes)
- Flit – FLow control unIT
  - Packets are divided into flits by the hardware
  - Typically no extra headers

Time-space diagram: bufferless flow control



Bufferless flow control using timeouts



- Packet is unable to acquire channel 3 in cycle 3 and is dropped
- Preceeding channels continue to transmit the packet until the tail flit is received
- Eventually a timeout triggers retransmission

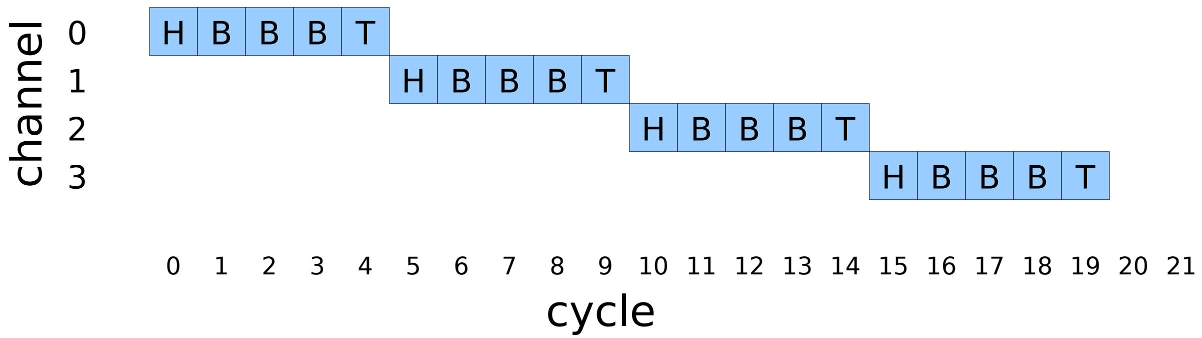
Pros/cons of bufferless flow control

+ Simple
– Inefficient:
  - Valuable bandwidth used for packet that are dropped later
  - Misrouting does not drop packets, but may lead to instability (packet may never reach destination)

Buffered flow control: overview

- Store-and-forward
  - Channel and buffer allocation on a per packet-basis
  - Receives full packets into buffer and forwards them after they have been received
  - High latency (each switch waits for full packet)
- Cut-through
  - Channel and buffer allocation on a per packet-basis
  - Forwards packet as soon as first (header) flit arrives and outgoing resources are available
  - Low latency but still blocks the channel for the duration of a whole packet transmission
- Wormhole
  - Buffers are allocated on a per-flit basis
  - Low latency and efficient buffer usage

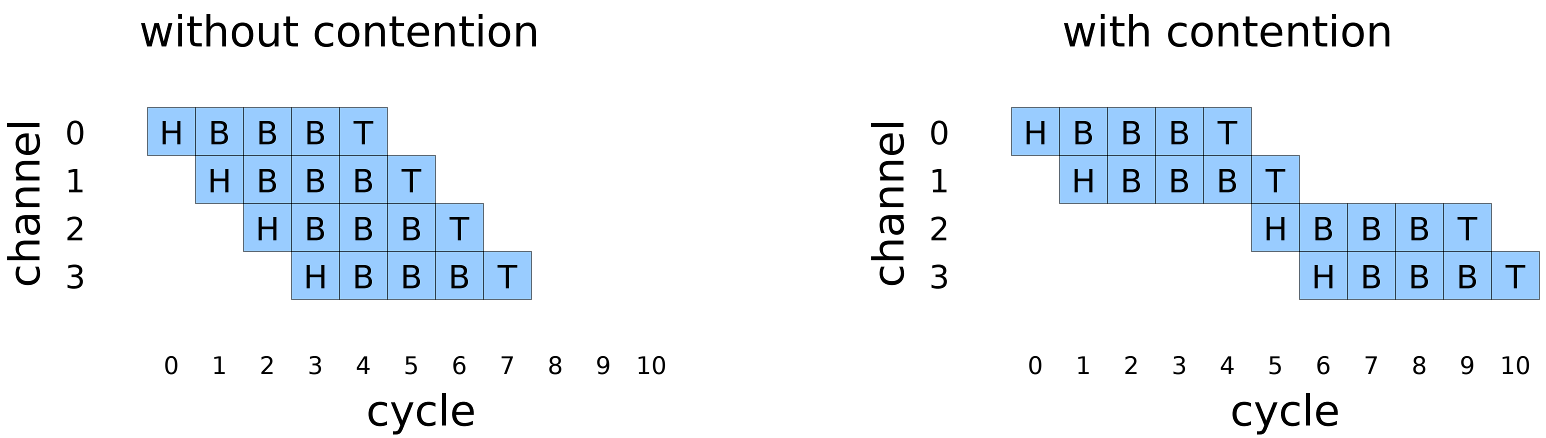
Store-and-forward flow control



- Packet must acquire full packet buffer at next hop, plus channel bandwidth before forwarding
- The entire packet is transmitted over the channel before proceeding to the next channel

– High latency!

Cut-through flow control



- Packet must acquire full packet buffer at next hop, and channel bandwidth before forwarding
- But: each flit of the packet is forwarded as soon as it is received

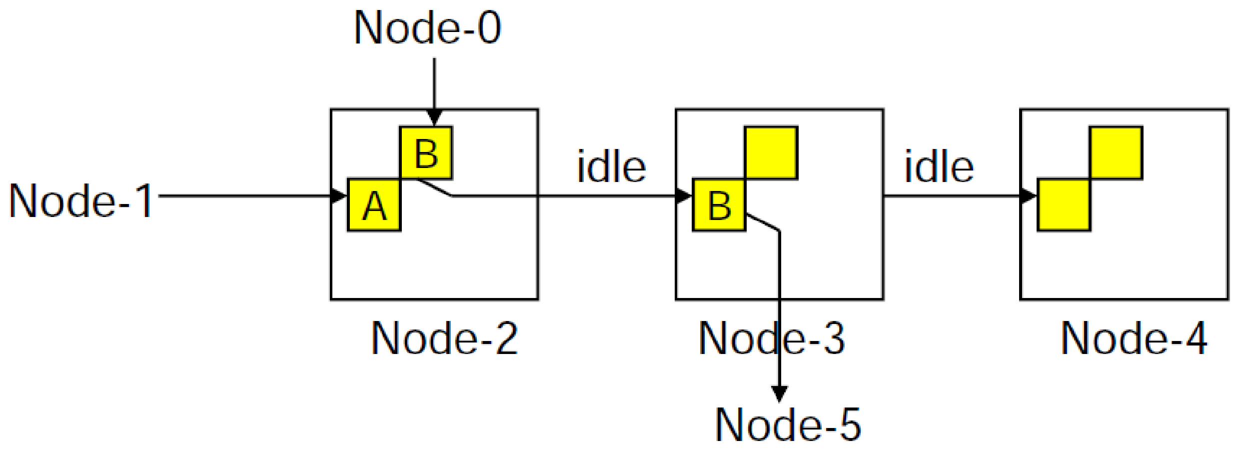
Wormhole routing

- Just like cut-through, but with buffers allocated to flits (not to packets)

- A head must acquire two resources before forwarding
  - One flit buffer at next switch
  - One flit of channel bandwidth

- Consumes much less buffer space than cut-through flow control

Head-of-line (HoL) blocking in Wormhole switching

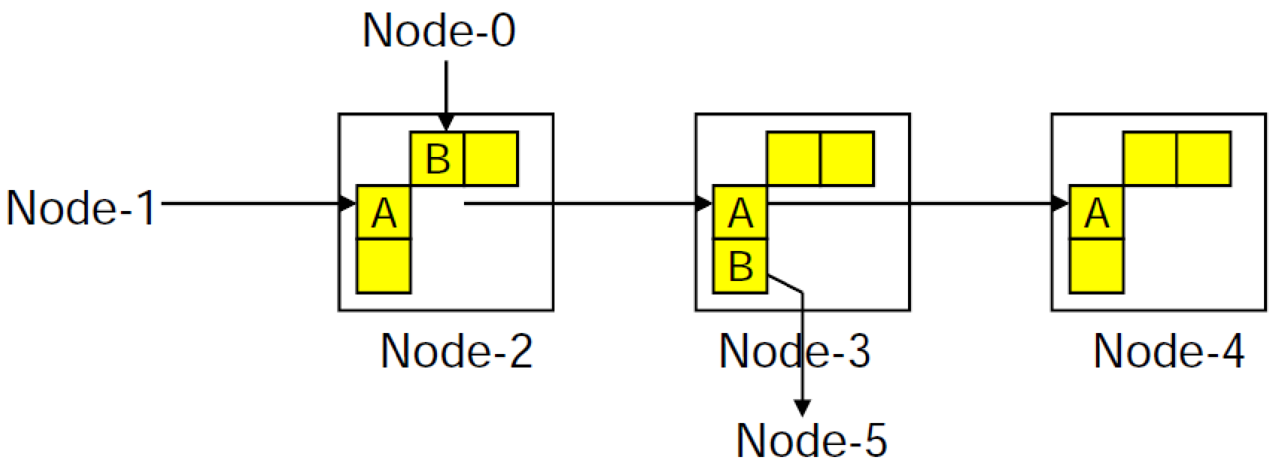


- Intended routing of packets:
  - A is going from Node-1 to Node-4
  - B is going from Node-0 to Node-5
- Situation: B is blocked at Node-3
  - Cannot acquire flit buffer at Node-5
- HoL blocking: A cannot progress to Node-4 even though all channels are idle
  - No intermixing of channels possible in Wormhole
  - B is locking channel from Node-0 to Node-3

Virtual channels

- Each switch has multiple virtual channels per physical channel
- Each virtual channel contains separate buffer space
- A head must acquire two resources before forwarding
  - A virtual channel on the next switch (including buffer space)
  - Channel bandwidth at the switch

Virtual channel flow control

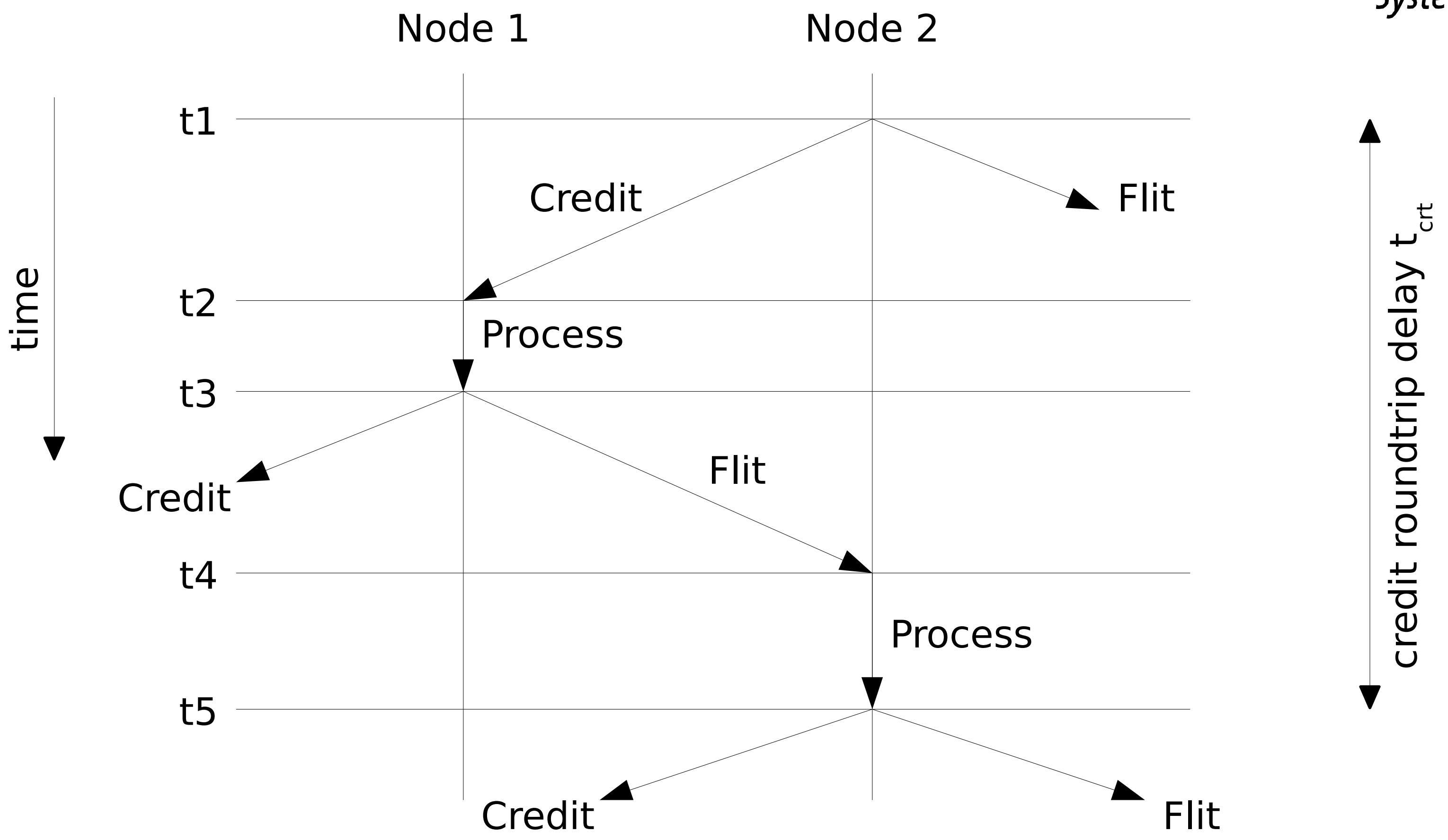


- Example: A can proceed even though B is blocked at Node-3

Buffer management

- How to communicate the availability of buffers between switches?
- Common types used today:
  - Credit-based
    - Switch keeps count of number of free flit buffers per downstream switch (credits)
    - Counter decreased when sending at downstream switch
    - Stop sending when counter reaches zero
    - Downstream switch sends back signal to increment credit counter when buffer is freed (forwarding)
  - One/off
    - Downstream switches send “on” or “off” flag to start and stop incoming flit stream

Credit-based flow control

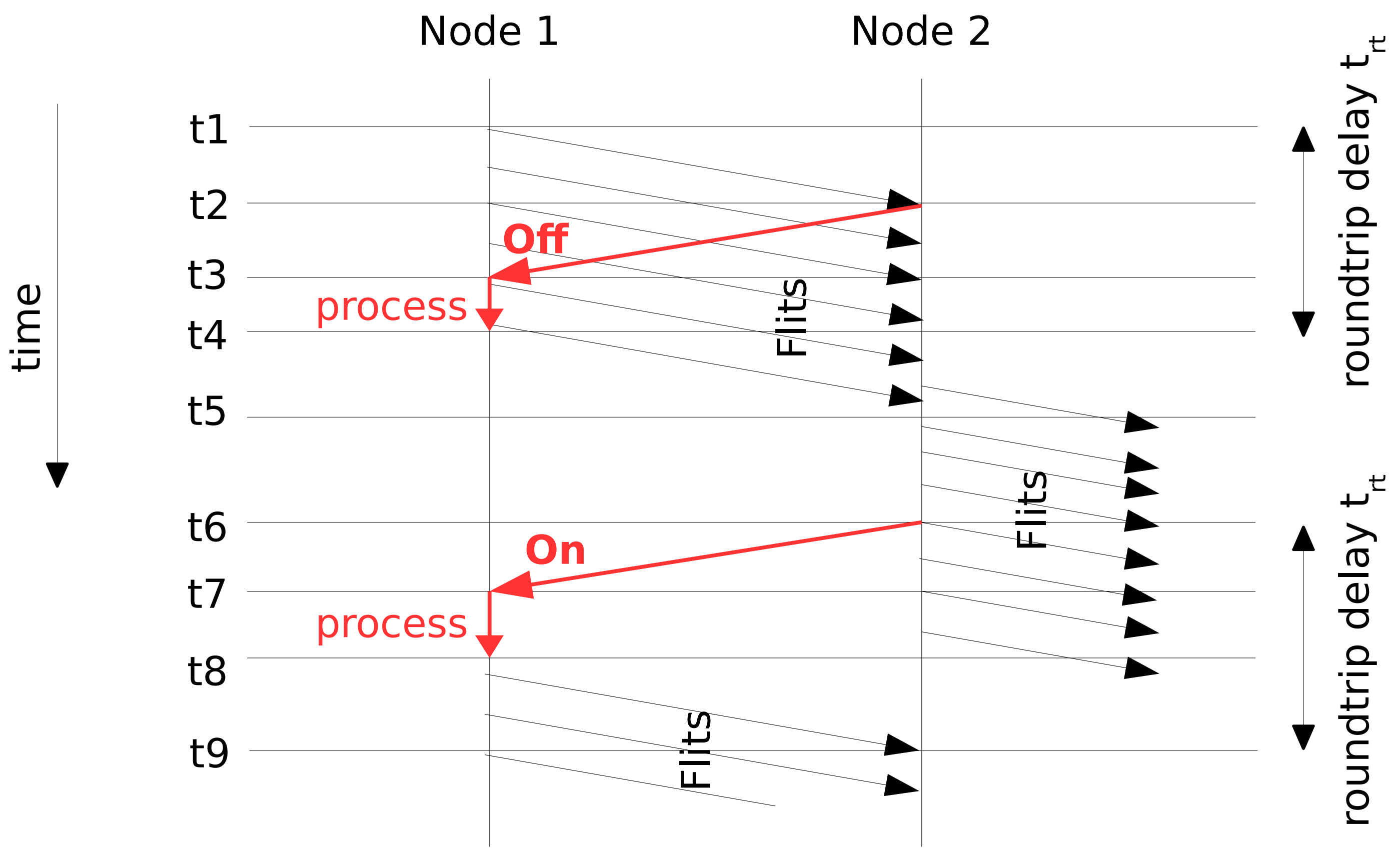


Need enough buffers to hide round trip time:

- With only one flit buffer: throughput = L_f / t_{crt}
- With F buffers: throughput = F*L_f / t_{crt}

L_f: flit length
F: #flits

On/off flow control

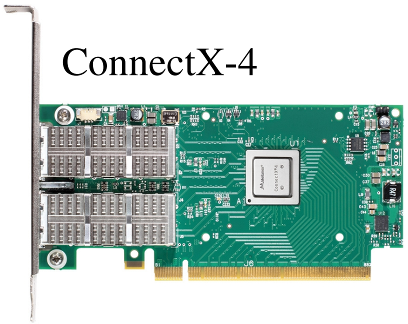


- T2: Switch sends “off” if its free buffer count falls below a limit F_{off}
- T6: Switch send “on” if its free buffer count rises above F_{on}
- Need to prevent additional flits from overflowing: $F_{off} >= t_{rt} * b / L_f$

b: bandwidth

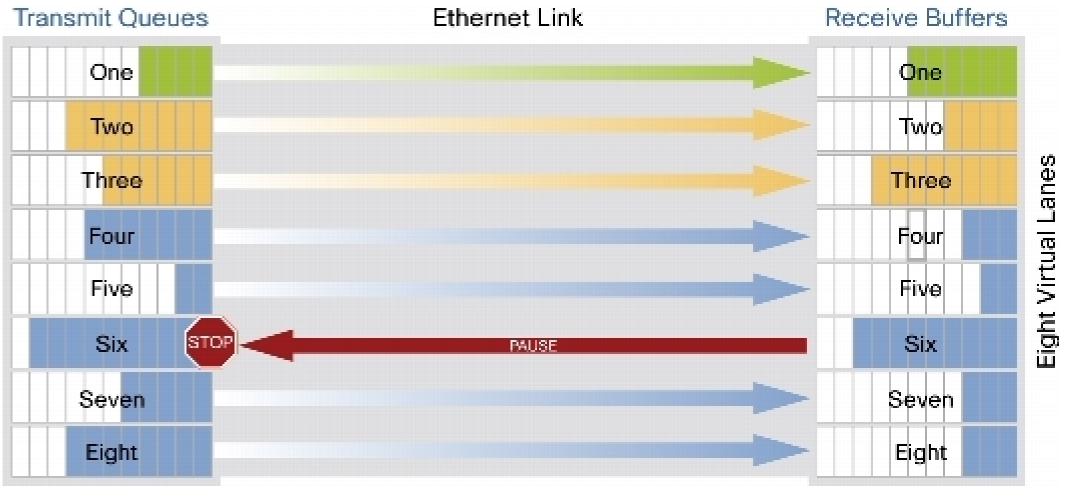
$L_{f:}$ flit length

Infiniband



- Interconnect technology designed for high-bandwidth, low-latency
  - Bandwidth: Up to 100Gbit/s (EDR, Mellanox ConnectX-4)
  - RTT latencies: 1-2 us
- Layered architecture
  - Physical layer
  - Link layer: **credit-based flow control**, **virtual lanes**
  - Network layer
  - Transport: reliable/unreliable, connection/datagram

Ethernet: PFC



- Priority Flow Control (PFC, IEEE 802.1bb):
  - PFC Pause/Start flow control for individual priority classes
  - PFC frame sent to immediate upstream entity (NIC or switch)
- PFC issues:
  - PFC is coarse grained: stops entire priority class
  - Unfair: blocks all flows, even those that didn't cause the congestion

Ethernet: Other Approaches

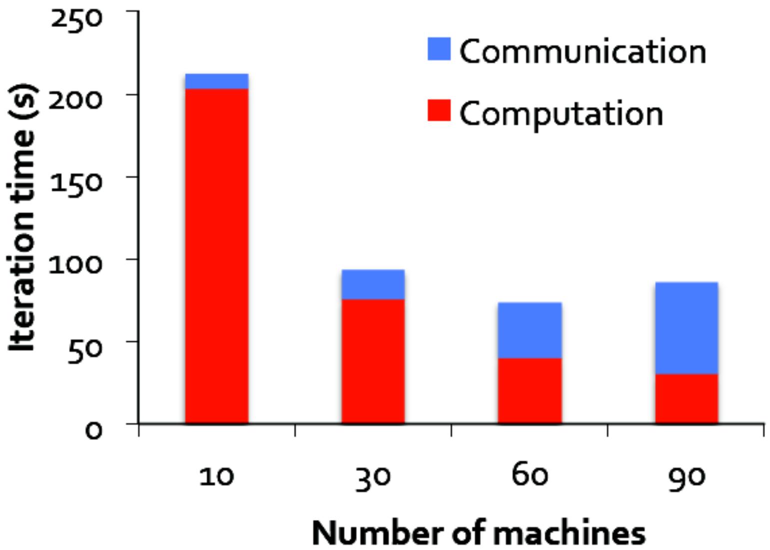
- Quantized Congestion Notification (QCN):
  - Add flow information to MAC packets
  - Detect congestion at switches
  - Send congestion notification message to end host
  - End host reacts and throttles the rate of the flow
  - QCN limits: does not work across subnet boundaries
- Data center QCN:
  - Like QCN but congestion notification messages are sent using UDP
  - Works across IP subnets
- Moral:
  - Ethernet was not designed with built-in flow-control
  - Retro-fitting flow control is difficult
  - Maybe this should not be done at the Ethernet layer then?
  - DCTCP is similar than QCN but is implemented at the transport layer

Limits of Flow-based Congestion Control

- Discussed several techniques to improve networking in partition/aggregate type of applications:
  - **Fine-grain TCP timers**: reduces long tail effects
  - **DCTCP**: reduces queue buildup
  - **D3 and D2DCTCP**: meet deadlines and SLAs
  - **Link-level flow contro**
- These approaches are all working on a per-flow basis
  - None of them looks at the collective behavior of flows by taking job semantics into account
  - No coordination between individual network transfers within a single job

Lack of coordination can hurt the performance

Scalability of Netflix recommendation system

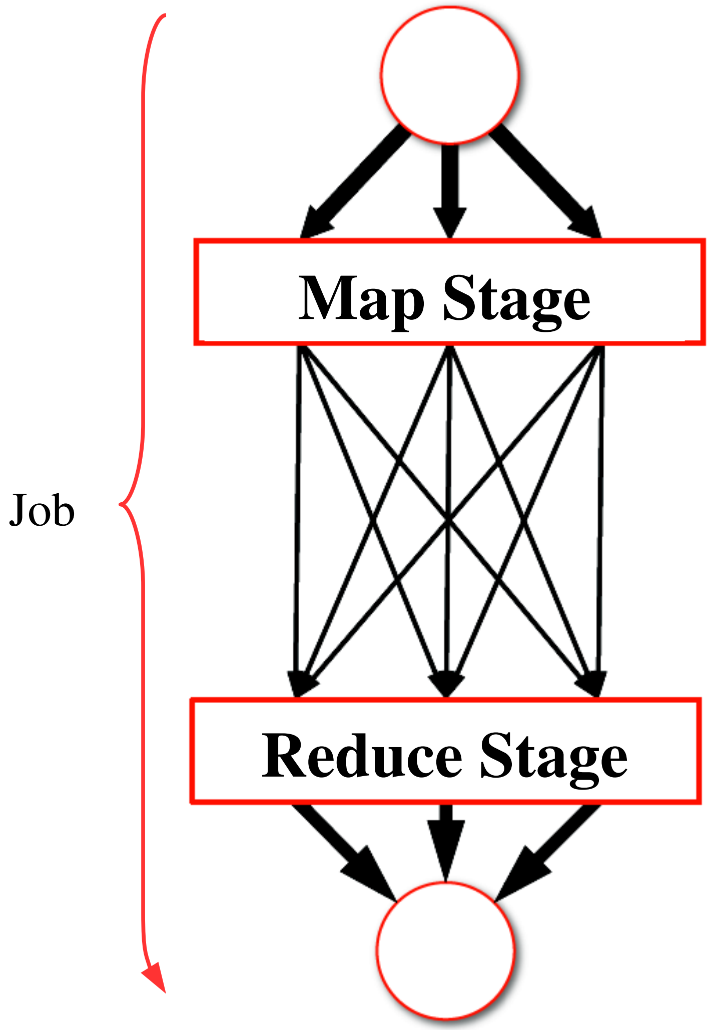


Did not scale beyond 60 nodes

» Comm. time increased faster than comp. time decreased

- Bottlenecked by communication as cluster size increases

Two key traffic patterns in MapReduce

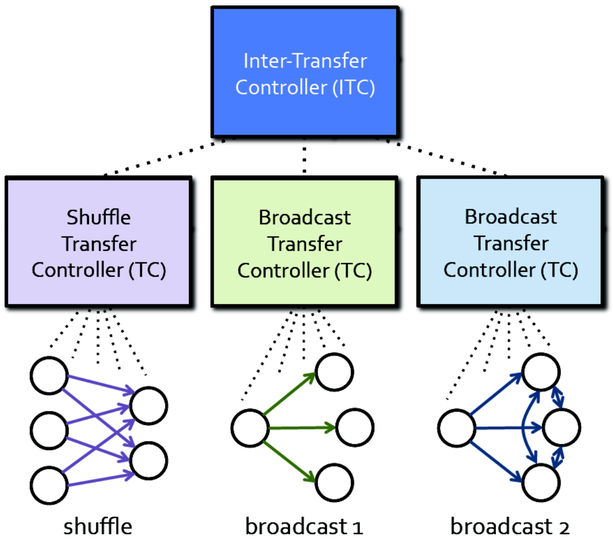


- Broadcast
  - One-to-many
  - Partition work
- Shuffle
  - Many-to-many
  - Aggregate results

Orchestra:

Managing Data Transfers in Computer Clusters

Orchestra: key idea

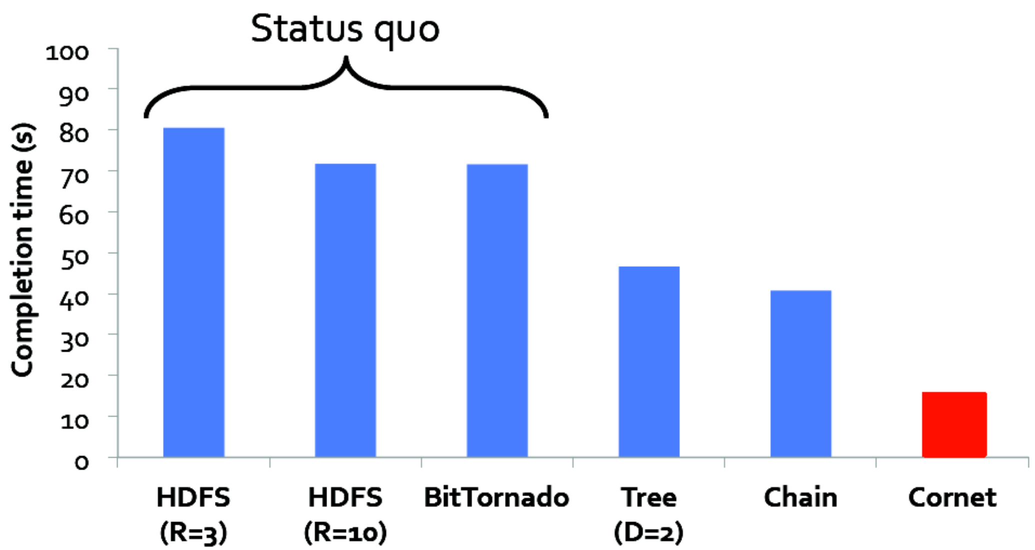


- Optimize at the level of transfers instead of individual flows
- Transfer: set of all flows transporting data between two stages of a job
- Coordination done through three control components
  - **Cornet**: cooperative broadcast
  - **Weighted shuffle scheduling**: shuffle coordination
  - **Inter-transfer controller (ITC)**: global coordination

Cornet: cooperative broadcast

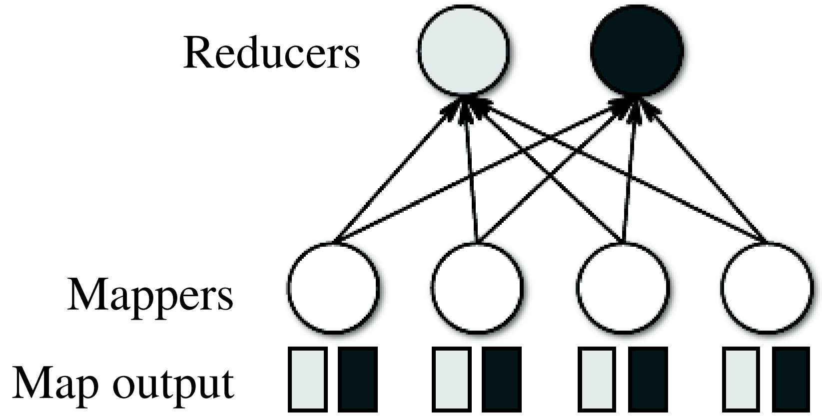
- **Key idea:** Broadcasting in MapReduce is very similar to data distribution mechanisms in the Internet like BitTorrent
- BitTorrent-like protocol optimized for data centers
  - Split data up into blocks and distribute them across nodes in the data center
  - On receiving: request/gather blocks from various nodes
  - Receivers of blocks become part of sender set (BitTorrent)
- Cornet differs from classical BitTorrent:
  - **Blocks are much larger** (4MB)
    - Data center is assumed to have high-bandwidth
  - **No need for incentives**
    - No selfish peers in the data center
  - **Topology aware**
    - Topology of data center is known
    - Receiver chooses sender in the same rack

Cornet performance



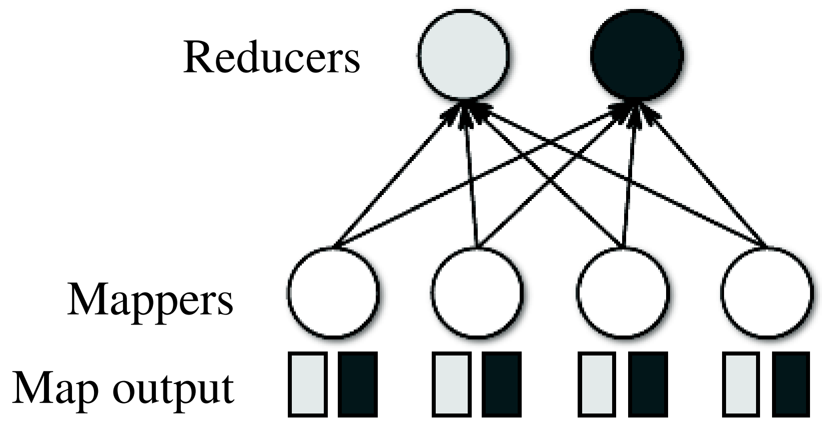
- Experiment: 100GB data to 100 receivers on Amazon EC2 cluster
- Traditional broadcast implementations use distributed file system to store and retrieve broadcast data
- Cornet is about 4-5 times more efficient

Shuffle: Status Quo (1)



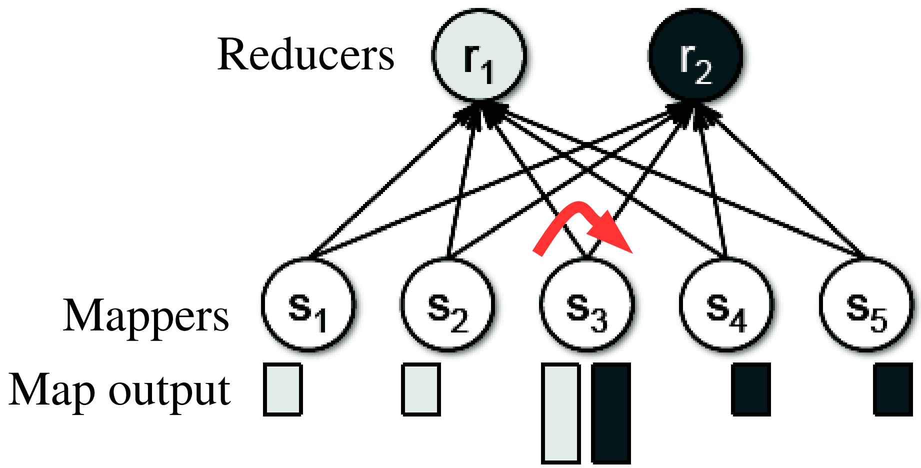
- To receivers (top) need to fetch separate pieces of data from each sender
- If every sender has equal amount of data, all links are equally loaded and utilized

Shuffle: Status Quo (1)



- To receivers (top) need to fetch separate pieces of data from each sender
- If every sender has equal amount of data, all links are equally loaded and utilized
- What if data sizes are unbalanced?

Shuffle: Sender Bottleneck

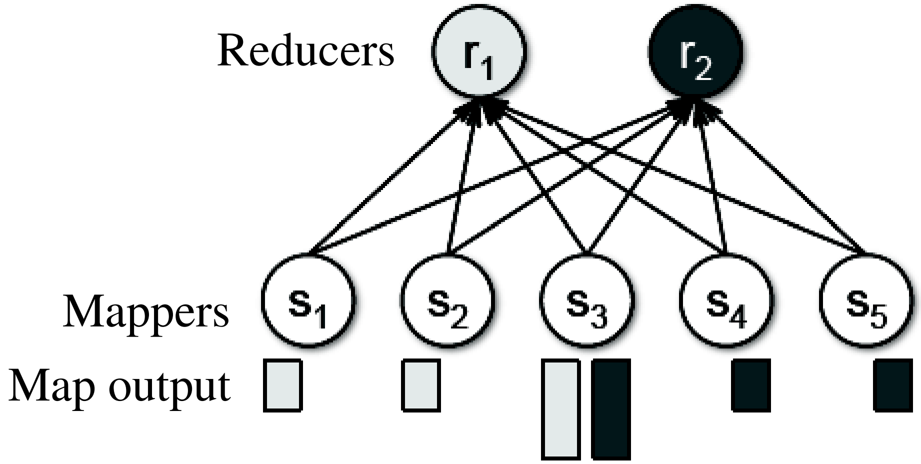


- Senders s1, s2, s4 and s5 have one data unit for each receiver
- Sender s3 has two data units for both receivers
- The link of the sender s3 becomes the bottleneck if flows share bandwidth in fair way

Orchestra: Weighted Shuffle Scheduling (WSS)

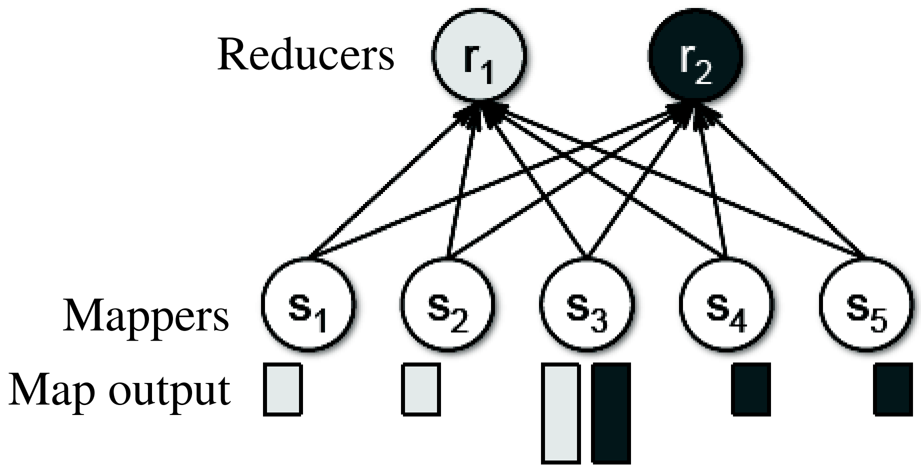
- Key idea:
  - Assign weights to each flow in a shuffle
  - Make the weight proportional to the data that needs to be transported

Example: shuffle with fair **bandwidth sharing**



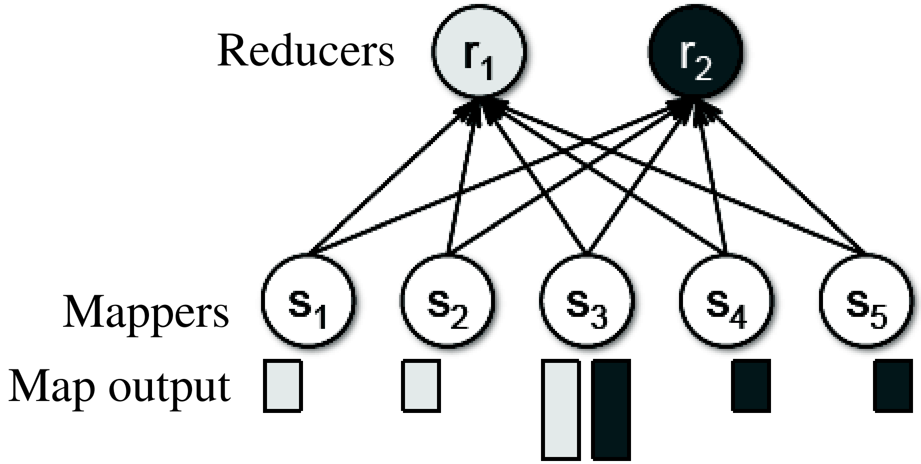
- Each receiver fetches data at 1/3 units/seconds from the three senders (three flows sharing bandwidth at receiver)

Example: shuffle with fair **bandwidth sharing**



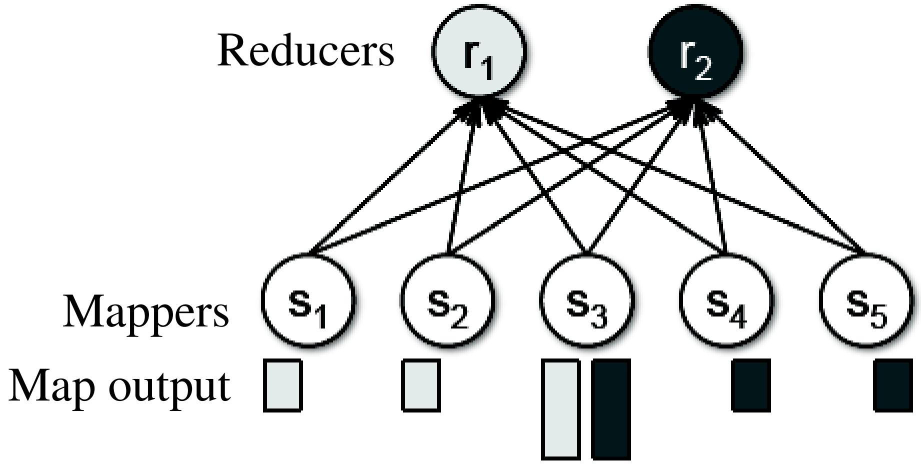
- Each receiver fetches data at 1/3 units/seconds from the three senders (three flows sharing bandwidth at receiver)
- After 3 seconds, all data from s1, s2, s4 and s5 is fetched

Example: shuffle with fair **bandwidth sharing**



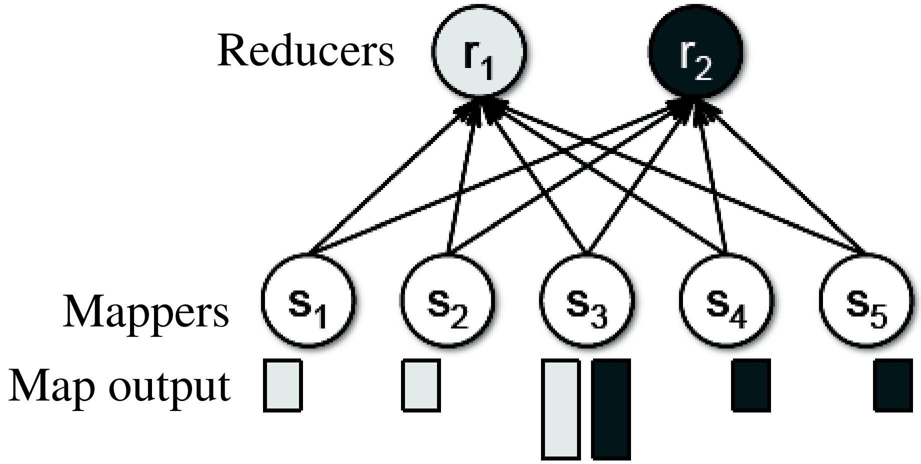
- Each receiver fetches data at 1/3 units/seconds from the three senders (three flows sharing bandwidth at receiver)
- After 3 seconds, all data from s1, s2, s4 and s5 is fetched
- But one unit of data left for both receivers at s3

Example: shuffle with fair **bandwidth sharing**



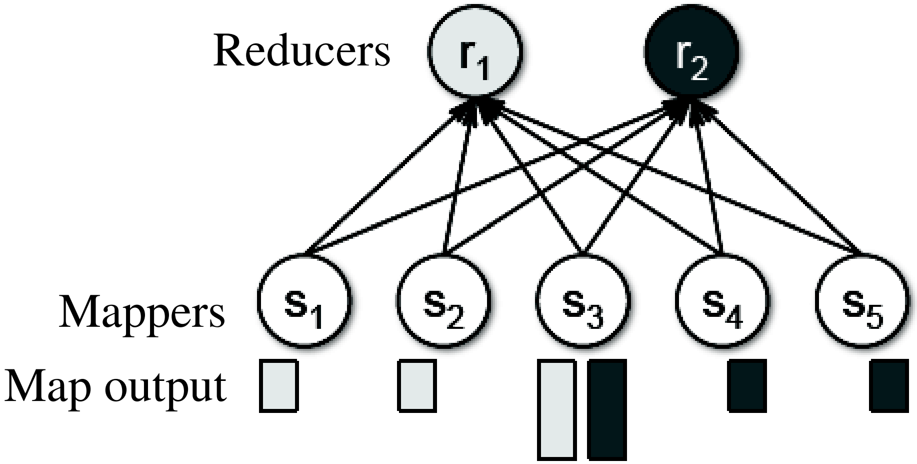
- Each receiver fetches data at 1/3 units/seconds from the three senders (three flows sharing bandwidth at receiver)
- After 3 seconds, all data from s1, s2, s4 and s5 is fetched
- But one unit of data left for both receivers at s3
- s3 transmits the two remaining units at 1/2 units per seconds to each receiver (two flows sharing the bandwidth at sender)

Example: shuffle with fair **bandwidth sharing**



- Each receiver fetches data at 1/3 units/seconds from the three senders (three flows sharing bandwidth at receiver)
- After 3 seconds, all data from s1, s2, s4 and s5 is fetched
- But one unit of data left for both receivers at s3
- s3 transmits the two remaining units at 1/2 units per seconds to each receiver (two flows sharing the bandwidth at sender)
- After two more seconds all units are transferred
- **Total time** = 5 seconds

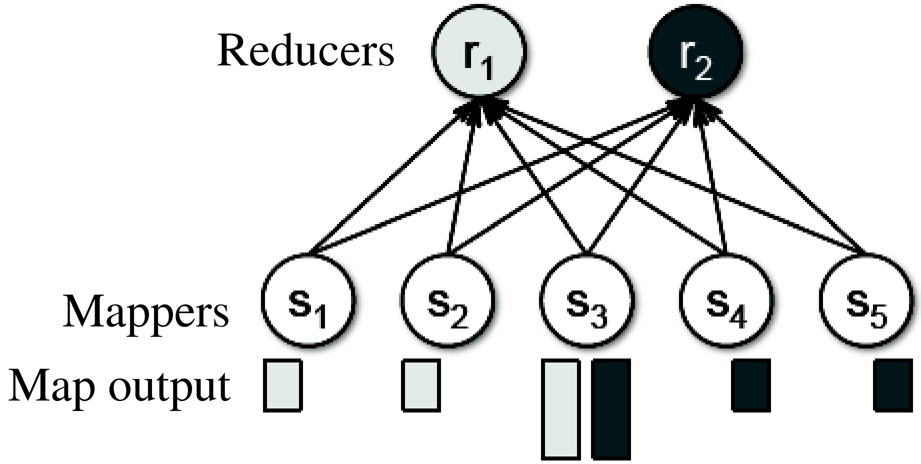
Example: shuffle with **weighted scheduling**



- Receivers fetch data at 1/4 units/seconds from s1, s2, s4 and s5

  ...and: fetch data at 1/2 units/seconds s3

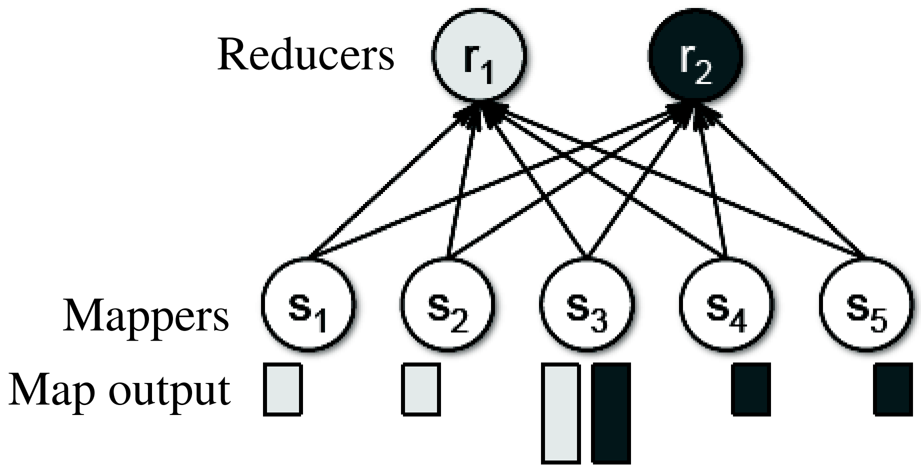
Example: shuffle with **weighted scheduling**



- Receivers fetch data at 1/4 units/seconds from s1, s2, s4 and s5

  ...and: fetch data at 1/2 units/seconds s3
- Fetching data from s1, s2, s4 and s5: 4 seconds
- Fetching data from s3: 4 seconds

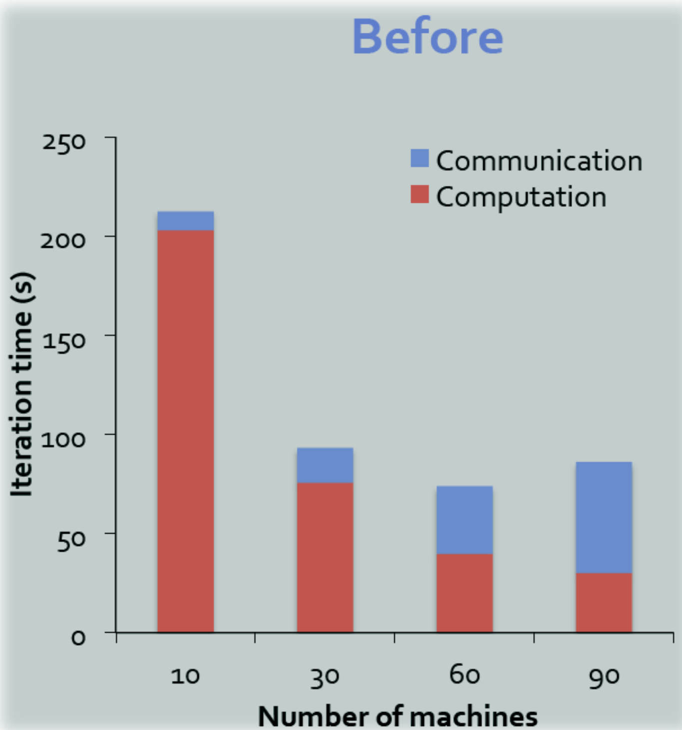
Example: shuffle with **weighted scheduling**

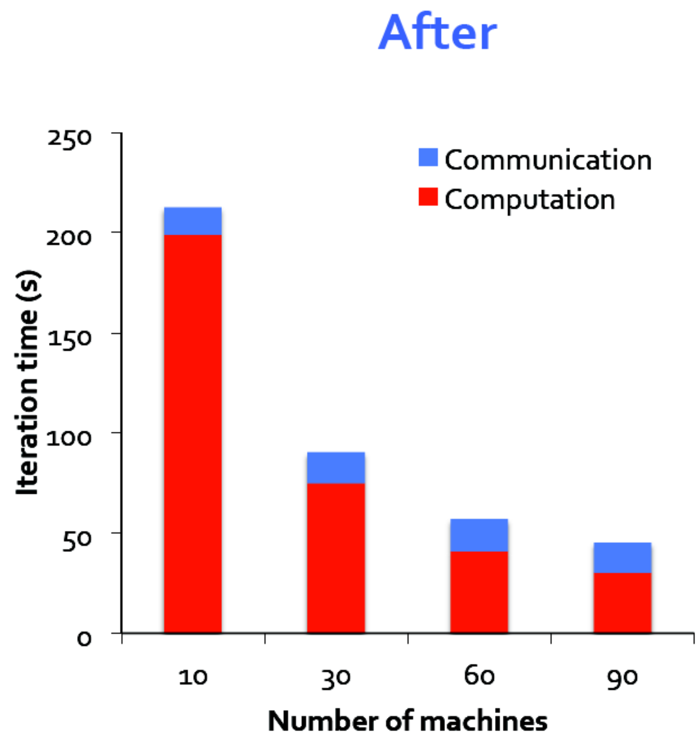


- Receivers fetch data at 1/4 units/seconds from s1, s2, s4 and s5

  ...and: fetch data at 1/2 units/seconds s3
- Fetching data from s1, s2, s4 and s5: 4 seconds
- Fetching data from s3: 4 seconds
- **Total time** = 4 seconds (25% faster than fair sharing)

Orchestra: End-to-end evaluation





- 1.9x faster on 90 nodes

Next time

- End host optimizations
  - User-level networking
  - RDMA