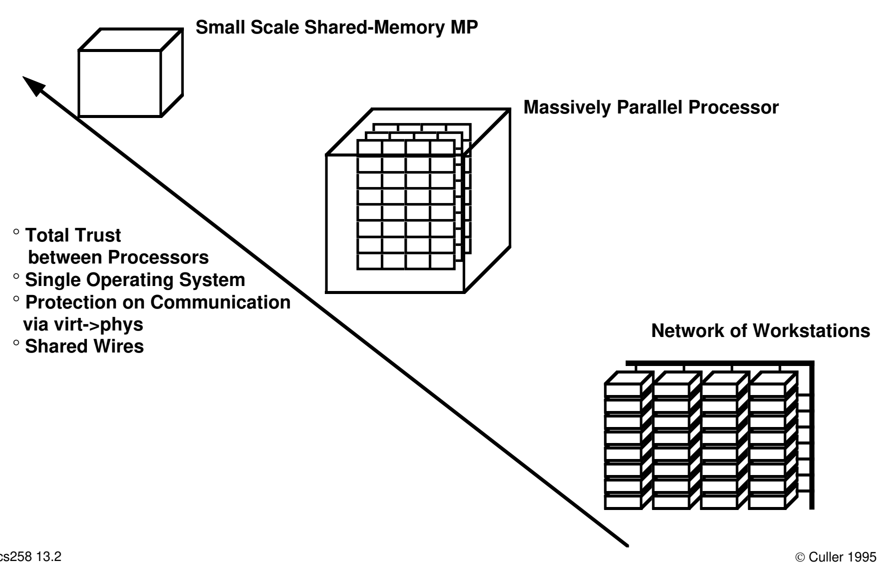
Large-Scale Distributed-Memory Multiprocessors

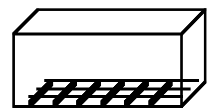
- Scalability
- Network Transactions
- Realizing Programming Models
- Input buffer problem
- Fetch deadlock with request/response

Coupling and Trust

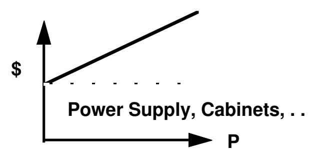


Scalability: what limits bus-based SMPs?

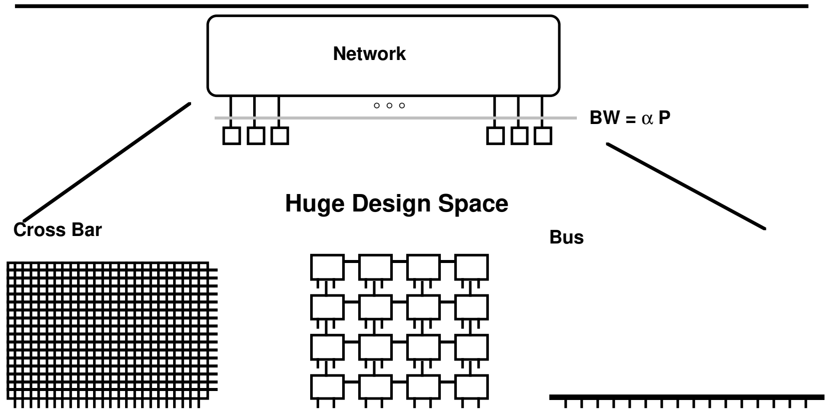
- Single set of wires => fixed bandwidth shared by P
  - **even with no communication, processor performance degrades with P**
    - » **focus on maximizing utility of scarce resource**
  - **BW must increase with P**
    - » **how much?**
- Physical constraints
  - **fixed number of slots**



- Electrical Constraints
  - **bus loading, wire length determine frequency**
  - **power**
- Cost Scaling
- O/S Scaling
  - **scheduling, locking, I/O**

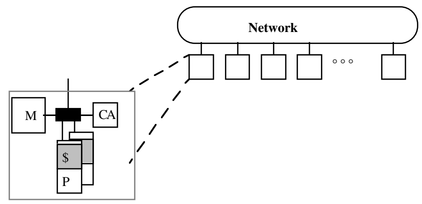


Scalable Interconnection Network



- Bandwidth (P)? Latency (P)? Cost (P) ?
- All increase!
  - **more BW, smaller L <=> greater cost**

Scaling in our Generic Parallel Architecture



- Distributed Memory
  - **=> perfect speedup in trivial case**
  - – **no communication, balanced independent load**
- Physical constraints ?
  - – **modular packaging, stack cabinets and expand networks**
- Electrical constraints ?
  - – **links get slower with length**
  - – **each technology has limits and fixed cost (copper, fiber, ...)**
  - **=> advantages to dense packaging,**
  - **but need to dissipate heat, debug, maintain, and track market**
- Cost ? - typically fixed fraction for P, M, I/O, and Com
- Conceptualization limits ?

Bottom Line

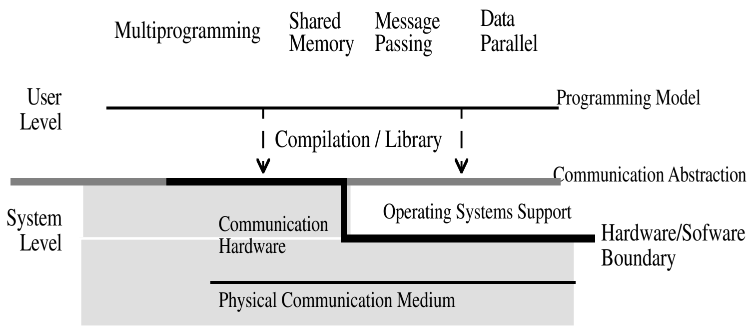
- There is no “truely scalable” machine

- Goal is to design for a given “range of scale”
  - – **one**
  - – **two to tens**
  - – **tens to hundreds**
  - – **hundreds to thousands**
- Techniques at one scale may not be cost effective at another
  - – **example: very dense packaging vs motherboard or box**

- Given that we have met the engineering requirements to physically scale over the range of interest,

  we must ensure that the communication and synchronization operations required to support target programming models also scale and are cost effective.

Realizing Programming Models



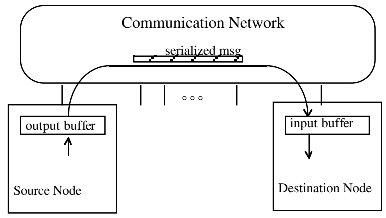
- How is user communication abstraction realized in terms of physical communication primitives
- Is this implemented directly in hardware or via system SW?

Arch. Extention in SMP

- Comm. Abstraction (shared addr) <=> Prog. model (shared var)
- Supported directly in hardware
  - **Load / Store <=> Assignment**
  - **shared VA => shared PA**
- Instruction Set Extensions
  - **atomic read-modify-write, load-locked / store-conditional**
- Memory Organization Extensions
  - **shared (lower portion) of the storage hierarchy**
  - **multiple local caches**

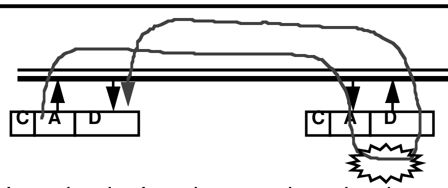
=> Cache Consistency
  - **enhance cache controllers with additional states**
  - **additional tags**
  - **additional transactions and wires on the bus**
    - » **read-exclusive, upgrade, Global-OR sharing, . . .**
  - **additional commands between levels (L2 <-> L1)**

Extensions in Large Scale Machines



- Basic Primitive: Network Transaction
  - **one-way transfer of information for output buffer at source node to input buffer at destination node causing an action**

Recall: Bus Transaction



- Virtual -> physical and protection check occurs prior

• Format:	physical bus wires (cmd, address, data)
• Output buffering:	registers hold information for transaction
• Media Arbitration:	centralized
• Destination:	implicit (broadcast address)
• Input buffering:	Registers ready for any request
• Action:	Simple storage access + state transition
• Ordering:	ordered request / response
• Completion	source & dest. know
• Deadlock	bus deadlock-free, protocols must avoid
• Delivery	guaranteed

Network Transactions

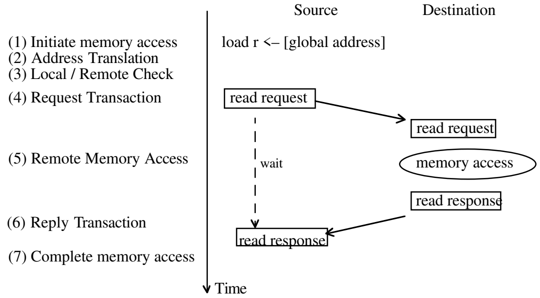
- Many options for protection check and translation
- Format: narrow links => serialization => flexible
  - **information *envelop***

  - **fixed vs. variable sized packets**
- Output buffering: regs, queue, memory + DMA staging regs.
- Media Arbitration: no global arbitration
  - **source controls admission based on limited knowledge**
  - **each transaction places claim on net and dest resources**
  - **incremental arbitration for shared resources**
- Destination explicit
  - **source must provide dest. name or route**
- Input Buffering regs, queue, memory + DMA staging regs.
  - **demanded on arrival**
  - **many sources may demand at once!**
    - » **How is this managed?**

Network Transactions (cont)

- Ordering one-way transfer of information
  - **all further ordering must be derived from a sequence of network transactions**
- Action May be simple to complex
- Completion source knows sent, dest knows recv
  - **must derive completion from a sequence of network transactions**
- Deadlock Networks are deadlock-free if modules accept transactions
  - **issues in routing, etc. for later**
  - **accept while unable to source**
- Delivery Small or no loss due to media errors
  - **key issue is what happens when input buffers are full**
    - » **at destination ?**
    - » **within the network ?**
  - **back-up or loss ?**

Realizing a Shared Address Space



- Fundamentally a two-way request/response protocol
  - **writes have an acknowledgement**
- Issues
  - **fixed or variable length (bulk) transfers**
  - **remote virtual or physical address**
  - **deadlock avoidance and input buffer full**

The Fetch Deadlock Problem

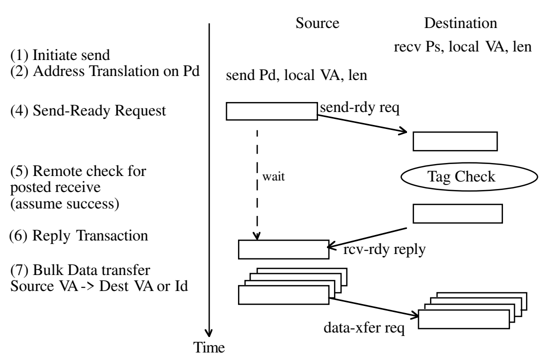
- Even if a node cannot issue a request, it must sink network transactions.
- Incoming transaction may be a request, which will generate a response.
- Closed system (finite buffering)

Key Properties of Shared Address Abstraction

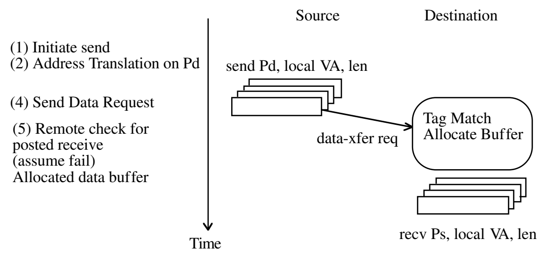
- Source and destination data addresses are specified by the source of the request
  - **a degree of logical coupling and trust**
- no storage logically "outside the address space"
  - » **may employ temporary buffers for transport**
- Operations are fundamentally request response
- Remote operation can be performed on remote memory
  - **logically does not require intervention of the remote processor**

Realizing Message Passing (synchronous)

- Send completes after matching recv and source data sent
- Receive completes after data transfer complete from matching send

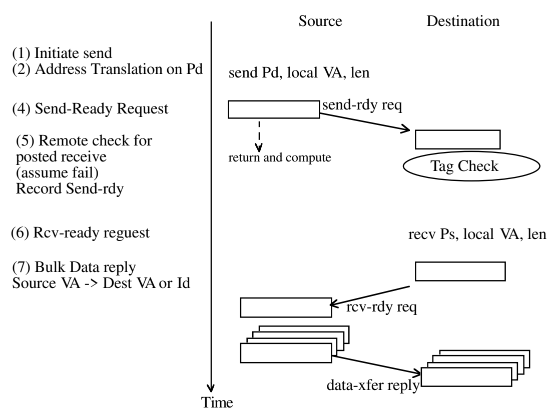


Message Passing (Asynchronous)



- Send variants complete after source area free or after command has been accepted
- Optimistic single-phase protocols assumes the destination can buffer send data with no matching receive *on demand*.
  - **amplified input buffering problem**
  - **fix is to use 3-phase protocol (plus fixed credits for small msgs) and buffer at the source**

Asynchronous (Safer) Msg Passing



Key Features of Msg Passing Abstraction

- Source knows send data address, dest. knows receive data address
  - **after handshake they both know both**
- Arbitrary storage "outside the local address spaces"
  - **may post many sends before any receives**
  - **non-blocking asynchronous sends reduces the requirement to an arbitrary number of descriptors**
    - » **fine print says these are limited too**
- Fundamentally a 3-phase transaction
  - **includes a request / response**
  - **can use optimisitic 1-phase in limited "Safe" cases**
    - » **credit scheme**

Solutions to Fetch Deadlock Problem

- (Logically) Separate Request and Response Networks
  - **request send blocks because buffer full ahead**
  - **reponse input buffer will drain**

  => response output buffer will drain

  => will be able to handle more requests

  => request input buffer will drain

  => request send will eventually succeed
- Reserve sufficient input buffer space
  - **limit k outstanding requests per processor**

  => kp potential incoming requests + k responses
  - **request send can still block for lack of credit**

  => must service incoming transactions while blocked in order to receive own responses and gain credit to send req.
- NAK requests while blocked on send-req
  - **sink respnses**
  - **use response input buffer for naks**
  - **livelock OK, but starvation?**

Challenges in Realizing Prog. Models in the Large

- One-way transfer of information
- No global knowledge, nor global control
  - **barriers, scans, reduce, global-OR give fuzzy global state**
- Very large number of concurrent transactions
- Management of input buffer resources
  - **many sources can issue a request and over-commit destination before any see the effect**
- Latency is large enough that you are tempted to “take risks”
  - **optimistic protocols**
  - **large transfers**
  - **dynamic allocation**
- Many many more degrees of freedom in design and engineering of these system