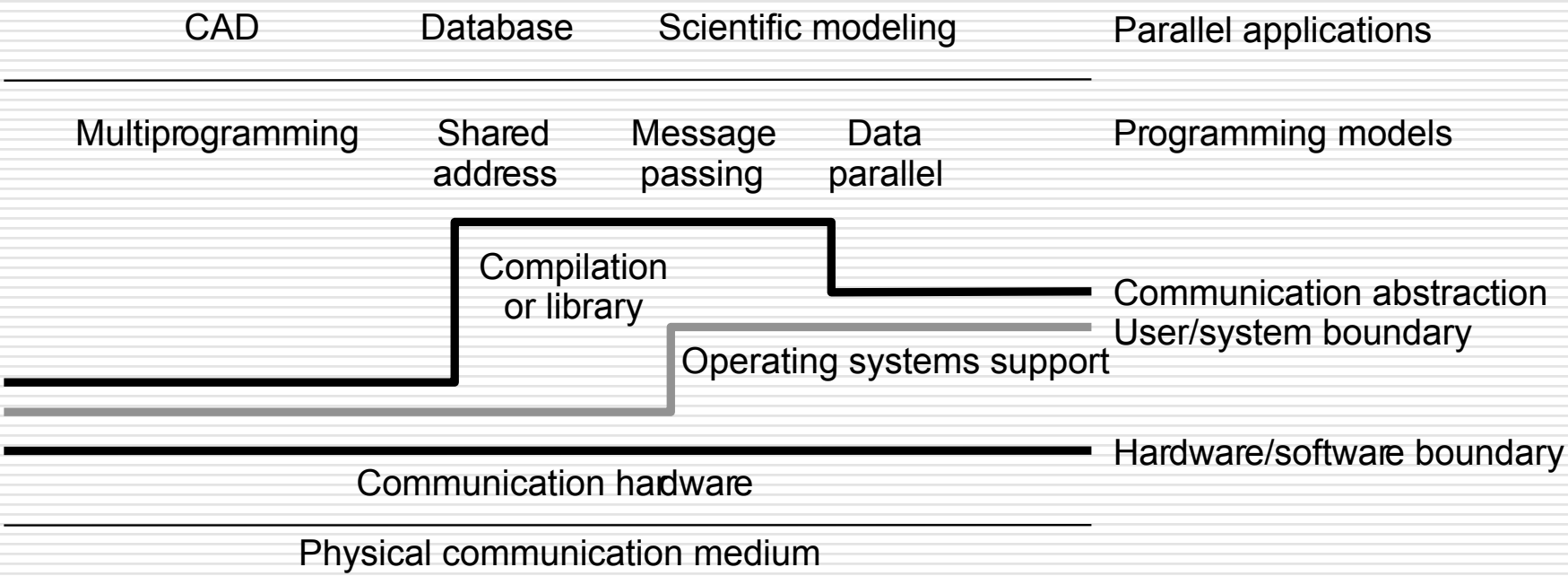
Parallel Machine Design Issues

Zeljko Zilic
McConnell Engineering Building
Room 546

Programming Models

- *Programming Model: Concept of machine for program use*
  - How parts cooperate and coordinate their activities
  - Specifies communication and synchronization operations
- Multiprogramming
  - No communication or synch. at program level
- *Shared memory (shared address space - SAS)*
  - Analogy: bulletin board
- *Message passing*
  - Like letters or phone calls, explicit point to point
- *Data parallel*:
  - More regimented, global actions on data
  - Implemented with shared address space or message passing

Layered Perspective of PCA



Communication Architecture

User/System Interface + Organization

- User/System Interface:
  - Comm. primitives (to user-level) by hw and system-level sw
- Implementation:
  - Implement the primitives: HW or OS
  - How optimized are they? How integrated into processing node?
  - Structure of network
- Goals:
  - Performance
  - Broad applicability
  - Programmability
  - Scalability
  - Low Cost

Communication Abstraction

- User level communication primitives provided
  - Realizes the programming model
  - Mapping exists between language primitives of programming model and these primitives
- Supported directly by hw, or via OS, or via user sw
- Debate about support in sw and gap between layers
- Today:
  - Hw/sw interface tends to be flat, i.e. complexity roughly uniform
  - Compilers and software play important roles as bridges
  - Technology trends exert strong influence
- Result is convergence in organizational structure
  - Relatively simple, general purpose communication primitives

Understanding Parallel Architecture

- Traditional taxonomies not very useful
- Programming models not enough, nor hardware structures
  - Same one can be supported by radically different architectures

=> *Architectural distinctions that affect software*

  - Compilers, libraries, programs
- Design of user/system and hardware/software interface
  - Constrained from above by progr. models and below by technology
- Guiding principles provided by layers
  - What primitives are provided at communication abstraction
  - How programming models map to these
  - How they are mapped to hardware

Fundamental Design Issues

- At any layer, interface (contract) aspect and performance aspects
  - *Naming*: How are logically shared data and/or processes referenced?
  - *Operations*: What operations are provided on these data
  - *Ordering*: How are accesses to data ordered and coordinated?
  - *Replication*: How are data replicated to reduce communication?
  - *Communication Cost*: Latency, bandwidth, overhead, occupancy

Sequential Programming Model

- Contract
  - Naming: Can name any variable (in virtual address space)
    - Hardware (and compilers) does translation to physical addresses
  - Operations: Loads, Stores, Arithmetic, Control
  - Ordering: Sequential program order
    - Always: read the last write to memory location

- Performance Optimizations
  - Compilers and hardware violate program order with impunity
    - Compiler: reordering and register allocation
    - Hardware: out of order, pipeline bypassing, write buffers
  - Retain dependence order on each "location"
  - Transparent replication in caches

Shared Memory Programming Model

- Naming: Any process can name any variable in shared space
- Operations: loads and stores, plus those needed for ordering
- Simplest Ordering Model:
  - Within a process/thread: sequential program order
  - Across threads: some interleaving (as in time-sharing)
  - Additional ordering through explicit synchronization
  - Can compilers/hardware weaken order without causing trouble?
    - Different, more subtle ordering models also possible (discussed later)

Synchronization

- Mutual exclusion (locks)
  - Ensure certain operations on certain data can be performed by only one process at a time
    - Room that only one person can enter at a time
  - No ordering guarantees
- Event synchronization
  - Ordering of events to preserve dependences
    - e.g. producer —> consumer of data
  - 3 main types:
    - point-to-point
    - global
    - group

Message Passing Programming Model

- Naming: Processes can name private data directly.
  - No shared address space
- Operations: Explicit communication through *send* and *receive*
  - Send transfers data from private address space to another process
  - Receive copies data from process to private address space
  - Must be able to name processes
- Ordering:
  - Program order within a process
  - Send and receive can provide pt to pt synch between processes
  - Mutual exclusion inherent + conventional optimizations legal
- Can construct global address space:
  - Process number + address within process address space
    - But no direct operations on these names

Design Issues Apply at All Layers

- Prog. model's position provides constraints/goals for system
- In fact, each interface between layers supports or takes a position on:
  - Naming model
  - Set of operations on names
  - Ordering model
  - Replication
  - Communication performance
- Any set of positions can be mapped to any other by software
- Common issues across layers
  - How lower layers can support contracts of programming models
  - Performance issues

Naming and Operations

- Naming and operations in programming model can be directly supported by lower levels, or translated by compiler, libraries or OS
- Example: Shared virtual address space in programming model
  - Hardware interface supports shared physical address space
    - Direct support by hardware through virtual-to-phy mapping, no software layers
  - Hardware supports independent physical address spaces
    - Can provide SM through OS, so in system/user interface
      - v-to-p mappings only for data that are local
      - remote data accesses incur page faults; brought in via page fault handlers
    - Compilers or runtime, so above sys/user interface

Naming and Operations: Msg Passing

- Direct support at hardware interface
  - But match and buffering benefit from more flexibility
- Support at sys/user interface or above in software
  - Hardware interface provides basic data transport (well suited)
  - Send/receive built in sw for flexibility (protection, buffering)
  - Choices at user/system interface:
    - OS each time: expensive
    - OS sets up once/infrequently, little sw involvement each time
  - Or lower interfaces provide SAS, and send/receive built on top with buffers and loads/stores
- Need to examine the issues and tradeoffs at every layer
  - Frequencies and types of operations, costs

Ordering

- Message passing: no assumptions on orders across processes except those imposed by send/receive pairs
- SM: How processes see the order of other processes' references defines semantics of SM
  - Ordering very important and subtle
  - Uniprocessors violate ordering to gain parallelism or locality
    - These goals are more important in multiprocessors
  - Need to understand which old tricks are valid, and learn new ones
  - How programs behave, what they rely on, and hardware implications

Replication

- Reduces data transfer/communication
  - depends on naming model
- Uniprocessor: caches do it automatically
  - Reduce communication with memory
- Message Passing naming model at an interface
  - Receive replicates, giving a new name
  - Replication is explicit in software above that interface
- SM naming model at an interface
  - A load brings in data, and can replicate transparently in cache
  - OS can do it at page level in shared virtual address space
  - No explicit renaming, many copies for same name: coherence problem
    - In uniprocessors, "coherence" of copies is natural in memory hierarchy

Communication Performance

- Performance characteristics determine usage of operations at a layer
  - Programmer, compilers etc make choices based on this
- Fundamentally, three characteristics:
  - *Latency*: time taken for an operation
  - *Bandwidth*: rate of performing operations
  - *Cost*: impact on execution time of program
- If processor does one thing at a time: bandwidth $\propto$ 1/latency
  - But actually more complex in modern systems
- Characteristics apply to overall operations, as well as individual components of a system

Simple Example

- Component performs an operation in 100ns
- Simple bandwidth: 10 Mops
- Internally pipeline depth 10 => bandwidth 100 Mops
  - Rate determined by slowest stage of pipeline, not overall latency
- Delivered bandwidth on application depends on initiation frequency
- Suppose application performs 100 M operations. What is cost?
  - op count * op latency gives 10 sec (upper bound)
  - op count / peak op rate gives 1 sec (lower bound)
    - assumes full overlap of latency with useful work, so just issue cost
  - If application can do 50 ns of useful work before depending on result of op, cost to application is the other 50ns of latency

Linear Model of Data Transfer Latency

- *Transfer time (n)* $= T_0 + n/B$
  - True for message passing, memory access, vector ops …
- As *n* increases, bandwidth approaches asymptotic rate *B*
  - Convergence speed depends on T_0
- Size needed for half bandwidth (half-power point):
  - $n_{1/2} = T_0 / B$
- But linear model not enough
  - When can next transfer be initiated? Can cost be overlapped?
  - Need to know how transfer is performed

Communication Cost Model

- Comm Time per message= Overhead + Assist Occupancy + Network Delay + Size/Bandwidth + Contention
  - $= o_v + o_c + l + n/B + T_c$
- Overhead and assist occupancy (service time) may be *f(n)* or not
- Each component along the way has occupancy and delay
  - Overall delay is sum of delays
  - Overall occupancy (1/bandwidth) is biggest of occupancies
- Comm Cost = frequency * (Comm time - overlap)
- General model for data transfer: applies to cache misses too

Diminishing Role of Topology

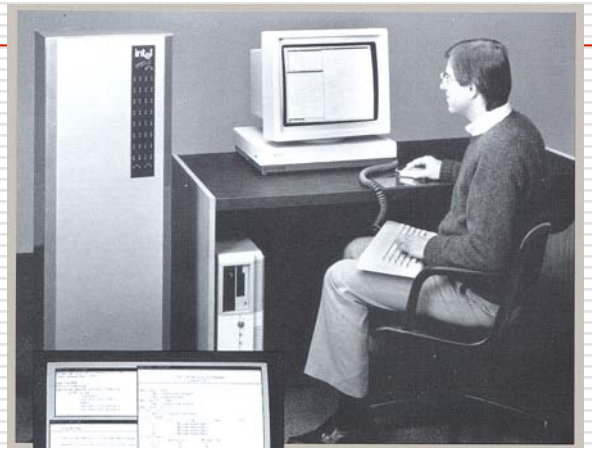
- Shift to general links
  - DMA, enabling non-blocking ops
    - Buffered by system at destination until recv
  - Store&forward routing
- Diminishing role of topology
  - Any-to-any pipelined routing
  - Node-network interface dominates communication time

$$H \times (T_0 + n/B)$$

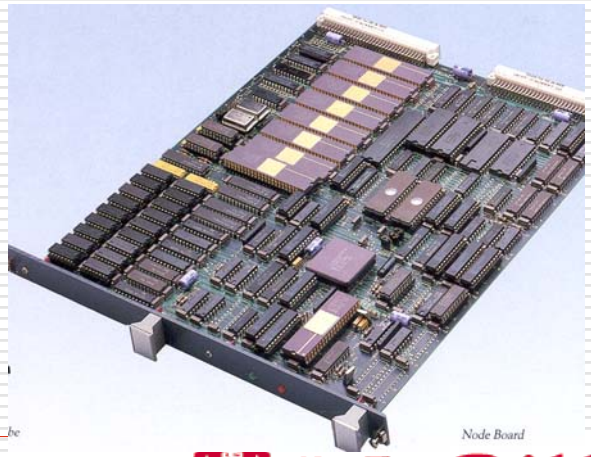
vs

$$T0 + H\Delta + n/B$$

  - Simplifies programming
  - Allows richer design space
    - grids vs hypercubes



Intel iPSC/1 -> iPSC/2 -> iPSC/860



Toward Architectural Convergence

- Evolution and role of software have blurred boundary
  - Send/recv supported on SAS machines via buffers
  - Can construct global address space on MP (GA -> P | LA)
  - Page-based (or finer-grained) shared virtual memory
- Hardware organization converging too
  - Tighter NI integration even for MP (low-latency, high-bandwidth)
  - Hardware SAS passes messages
- Even clusters of workstations/SMPs are parallel systems
  - Emergence of fast system area networks (SAN)
- Programming models distinct, but organizations converging
  - Nodes connected by general network and communication assists
  - Implementations also converging, at least in high-end machines

Summary of Design Issues

- Functional and performance issues apply at all layers
- Functional: Naming, operations and ordering
- Performance: Organization
  - latency, bandwidth, overhead, occupancy
- Replication and communication are deeply related
  - Management depends on naming model
- Goal of architects: design against frequency and type of operations that occur at communication abstraction, constrained by tradeoffs from above or below
  - Hardware/software tradeoffs