Recap: 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

Convergence: Generic Parallel Architecture

- Node: processor(s), memory system, plus communication assist
  - Network interface and communication controller
- Scalable network
- Convergence allows lots of innovation, within framework
  - Integration of assist with node, what operations, how efficiently...

Data Parallel Systems

- Programming model
  - Operations performed in parallel on each element of data structure
  - Logically single thread of control, performs sequential or parallel steps
  - Conceptually, a processor associated with each data element
- Architectural model
  - Array of many simple, cheap processors with little memory each
    - Processors don’t sequence through instructions
    - Attached to a control processor that issues instructions
  - Specialized and general communication, cheap global synchronization
- Original motivations
  - Matches simple differential equation solvers
  - Centralize high cost of instruction fetch/sequencing

Application of Data Parallelism

- Each PE contains an employee record with his/her salary
  - If salary > 100K then
    - salary = salary *1.05
  - else
    - salary = salary *1.10
  - Logically, the whole operation is a single step
  - Some processors enabled for arithmetic operation, others disabled
- Other examples:
  - Finite differences, linear algebra, ...
  - Document searching, graphics, image processing, ...
- Some recent machines:
  - Thinking Machines CM-1, CM-2 (and CM-5)
  - Maspar MP-1 and MP-2,
Flynn’s Taxonomy

- # instruction x # Data
  - Single Instruction Single Data (SISD)
  - Single Instruction Multiple Data (SIMD)
  - Multiple Instruction Single Data
  - Multiple Instruction Multiple Data (MIMD)
- Everything is MIMD!

Evolution and Convergence

- SIMD Popular when cost savings of centralized sequencer high
  - 60s when CPU was a cabinet
  - Replaced by vectors in mid-70s
  - More flexible w.r.t. memory layout and easier to manage
  - Revived in mid-80s when 32-bit datapath slices just fit on chip
- Simple, regular applications have good locality
- Programming model converges with SPMD (single program multiple data)
  - need fast global synchronization
  - Structured global address space, implemented with either SAS or MP

CM-5

- Repackaged SparcStation
  - 4 per board
- Fat-Tree network
- Control network for global synchronization

Dataflow Architectures

- Represent computation as a graph of essential dependences
  - Logical processor at each node, activated by availability of operands
  - Message (tokens) carrying tag of next instruction sent to next processor
  - Tag compared with others in matching store; match fires execution

Evolution and Convergence

- Key characteristics
  - Ability to name operations, synchronization, dynamic scheduling
- Problems
  - Operations have locality across them, useful to group together
  - Handling complex data structures like arrays
  - Complexity of matching store and memory units
  - Expose too much parallelism (?)
- Converged to use conventional processors and memory
  - Support for large, dynamic set of threads to map to processors
  - Typically shared address space as well
  - But separation of program model from hardware (like data-parallel)
- Lasting contributions:
  - Integration of communication with thread (handler) generation
  - Tightly integrated communication and fine-grained synchronization
  - Remained useful concept for software (compilers etc.)
Systolic Architectures

- VLSI enables inexpensive special-purpose chips
  
  - Represent algorithms directly by chips connected in regular pattern
  
  - Replace single processor with array of regular processing elements
  
  - Orchestrate data flow for high throughput with less memory access

- Different from pipelining
  
  - Nonlinear array structure, multidirection data flow, each PE may have
    (small) local instruction and data memory

- SIMD? : each PE may do something different

Systolic Arrays (contd.)

Example: Systolic array for 1-D convolution

\[ y(i) = w_1 \times x(i) + w_2 \times x(i+1) + w_3 \times x(i+2) + w_4 \times x(i+3) \]

Architecture

- Two facets of Computer Architecture:
  
  - Defines Critical Abstractions
    
    - especially at HW/SW boundary
    
    - set of operations and data types these operate on
  
  - Organizational structure that realizes these abstractions

- Parallel Computer Arch. = Comp. Arch + Communication Arch.

- Comm. Architecture has same two facets
  
  - communication abstraction
  
  - primitives at user/system and hw/sw boundary

Layered Perspective of PCA

Communication Architecture

User/System Interface + Organization

- User/System Interface:
  
  - Comm. primitives exposed to user-level by hw and system-level sw

- Implementation:
  
  - Organizational structures that 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 programmer
    model and these primitives

- Supported directly by hw, or via OS, or via user sw

- Lot of debate about what to 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 today
  
  - 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
  - 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 perhaps compilers) does translation to physical addresses
  - Operations: Loads, Stores, Arithmetic, Control
  - Ordering: Sequential program order

- Performance Optimizations
  - Compilers and hardware violate program order without getting caught
  - Compiler: reordering and register allocation
  - Hardware: out of order, pipeline bypassing, write buffers
  - Retain dependence order on each “location”
  - Transparent replication in caches

SAS 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 getting caught?
  - 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
  - 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
- Let's see issues across layers
  - How lower layers can support contracts of programming models

### 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 v-to-p mappings, no software layers
- Hardware supports independent physical address spaces
  - Can provide SAS 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/int frequently, then 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
- SAS: How processes see the order of other processes' references defines semantics of SAS
  - Ordering very important and subtle
  - Unprocessors play tricks with ordering to gain parallelism or locality
  - These 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
- Unprocessors: 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
- SAS 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 unprocessors, “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 \( \ll \frac{1}{\text{latency}} \)
  - 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)
  - 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 \( (n) = T_0 + n/B \)
  - useful for message passing, memory access, vector ops etc
- As \( n \) increases, bandwidth approaches asymptotic rate \( B \)
- How quickly it approaches 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
  - \( O_o + O_a + I + n/B + T_c \)
- Overhead and assist occupancy 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

**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