Diversity and Convergence of Parallel Architectures

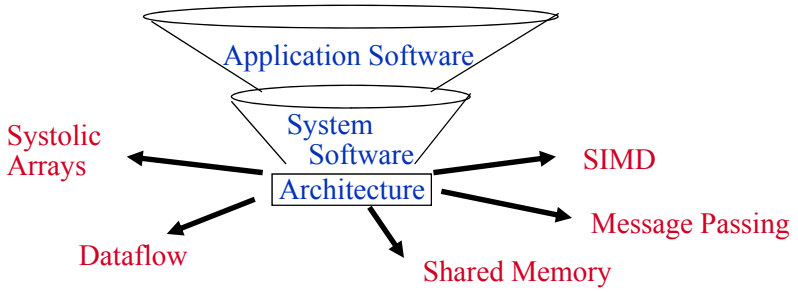
Muhamed Mudawar

Computer Engineering Department

King Fahd University of Petroleum and Minerals

Divergent Parallel Architectures

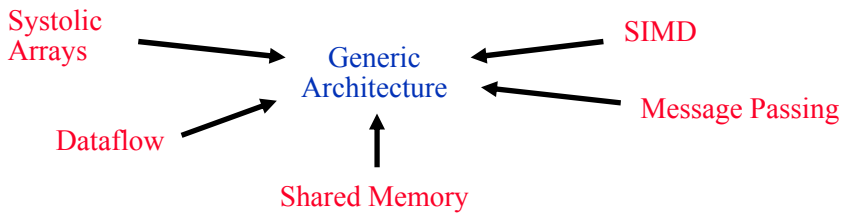
- Parallel architectures were tied closely to programming models
- Divergent architectures, with no predictable pattern for growth



- Uncertainty of direction paralyzed parallel software development

Plan of this Presentation

- Look at major programming models
  - From where they came? what do they provide?
- Understand the diversity of parallel architectures
- Understand how parallel architectures have converged
- Understand fundamental design issues



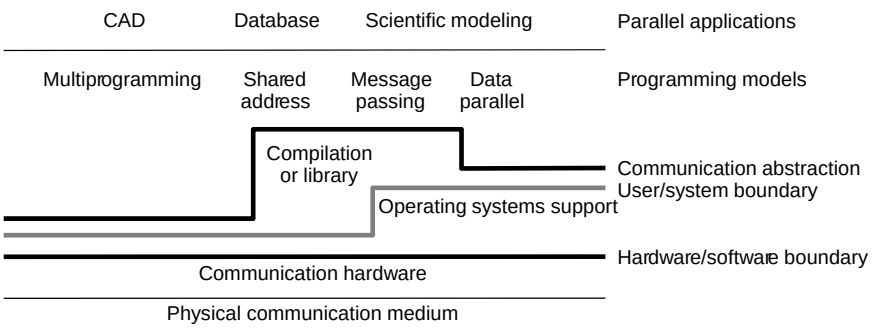
Flynn's Taxonomy

- SISD (Single Instruction Single Data)
  - Uniprocessors
- MISD (Multiple Instruction Single Data)
  - Single data stream operated by successive functional units
- SIMD (Single Instruction Multiple Data)
  - Instruction stream executed by multiple processors on different data
    - Simple programming model, low overhead
  - Example: Vector Processor
- MIMD (Multiple Instruction Multiple Data) is the most general
  - Flexibility for parallel applications and multi-programmed systems
    - Processors can run independent processes or applications
    - Processors can also run threads belonging to one parallel application
  - Uses off-the-shelf microprocessors and components

Major MIMD Styles

- Symmetric Multiprocessors (SMP)
  - Main memory is shared and equally accessible by all processors
    - Called also Uniform Memory Access (UMA)
    - Bus based or interconnection network based
- Distributed memory multiprocessors
  - Distributed Shared Memory (DSM) multiprocessors
    - Distributed memories are shared and can be accessed by all processors
    - Non-uniform memory access (NUMA)
    - Latency varies between local and remote memory access
  - Message-Passing multiprocessors, multi-computers, and clusters
    - Distributed memories are NOT shared
    - Each processor can access its own local memory
    - Processors communicate by sending and receiving messages

Layers of Abstraction

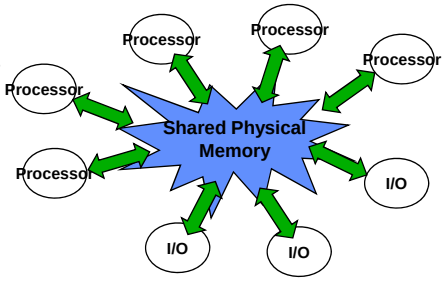


Programming Model

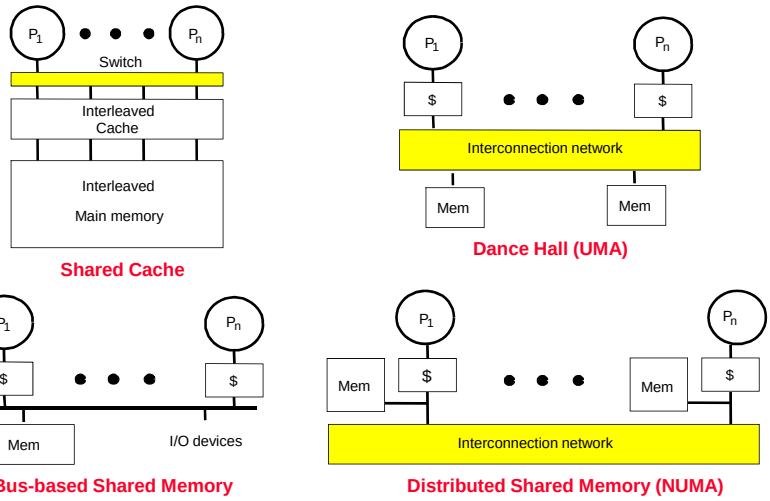
- **Concepts that programmers use in coding applications**
  - Specifies parallelism
  - How tasks cooperate and coordinate their activities
  - Specifies communication and synchronization operations
- Multiprogramming
  - No communication or synchronization at the program level
- Shared memory
  - Shared address space is used for communication and synchronization
- Message passing
  - Explicit messages are sent and received, explicit point to point
- Data parallel
  - Operations are performed in parallel on all elements of a data structure

Shared Memory Architecture

- Any processor can directly reference any physical memory
- Any I/O controller to any physical memory
- Operating system can run on any processor
  - OS uses shared memory to coordinate
- Communication occurs implicitly as result of loads and stores
- Wide range of scale
  - Few to hundreds of processors
  - Memory may be physically distributed among processors
- History dates to early 1960s

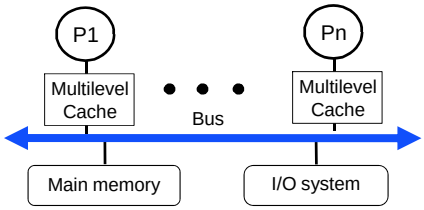


Shared Memory Organizations



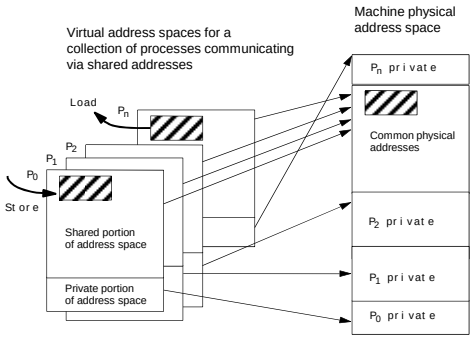
Bus-Based Symmetric Multiprocessors

- Symmetric access to main memory from any processor
- Dominate the server market
  - Building blocks for larger systems
- Attractive as throughput servers and for parallel programs
  - Uniform access via loads/stores
  - Automatic data movement and coherent replication in caches
  - Cheap and powerful extension to uniprocessors
  - Key is extension of memory hierarchy to support multiple processors

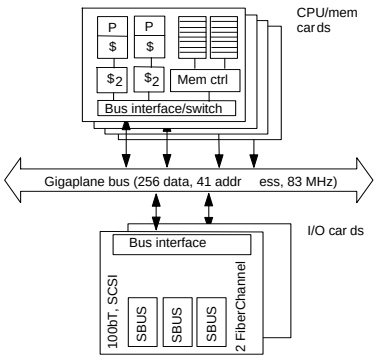


Shared Address Space Programming Model

- A process is a virtual address space
  - With one or more threads of control
- Part of the virtual address space is shared by processes
  - Multiple threads share the address space of a single process
- All communication is through shared memory
  - Achieved by loads and stores
  - Writes by one process/thread are visible to others
  - Special atomic operations for synchronization
  - OS uses shared memory to coordinate processes

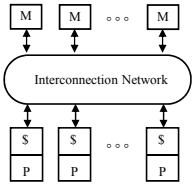


Engineering: SUN Enterprise

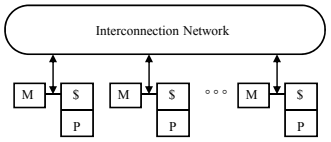


- CPU/Mem cards and I/O cards
  - 16 cards of either type
  - All memory accessed over bus
  - Symmetric multiprocessor, symmetric access to all memory locations
  - Highly pipelined memory bus, 2.5 GB/sec
  - Two UltraSparc processors per card, each with level-1 and level-2 caches

Medium and Large Scale Multiprocessors



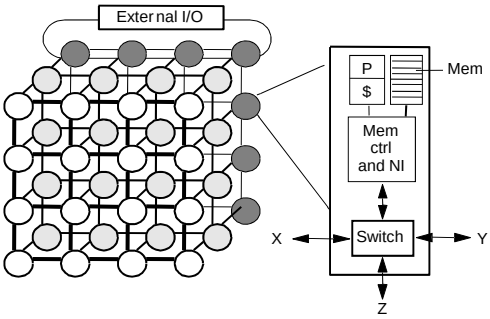
Centralized Memory "Dance hall"



Distributed Shared Memory

- Problem is interconnect: high cost (crossbar) or bandwidth (bus)
- Centralized memory or uniform memory access (**UMA**) – Less common today
  - Latencies to memory uniform, but uniformly large
  - Interconnection network: crossbar or multi-stage
- Distributed shared memory (**DSM**) or non-uniform memory access (**NUMA**)
  - Distributed memory forms one global shared physical address space
  - Access to local memory is faster than access to remote memory
- Caching shared (particularly nonlocal) data is an issue

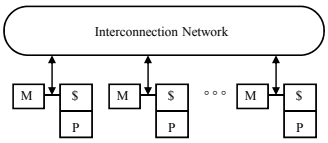
Engineering: Cray T3E



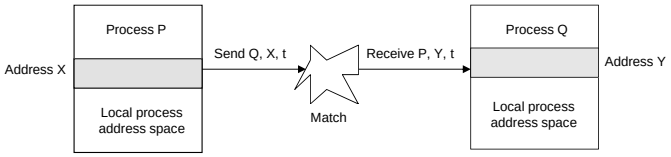
- Scales up to 1024 processors
- Supports a global shared address space with non-uniform memory access
- Each node contains Alpha processor, memory, network interface and switch
- Interconnection network is organized as a 3D cube with 650 MB/s links
- Memory controller generates request message for non-local references
- No hardware mechanism for cache coherence – Remote data is not cached

Message Passing Architectures

- Complete computer as a building block
  - Includes processor, memory, and I/O system
- Easier to build and scale than shared memory architectures
- Communication via explicit I/O operations
  - Communication integrated at I/O level, Not into memory system
- Much in common with networks of workstations
  - However, tight integration between processor and network
  - Network is of higher capability than a local area network
- Programming model
  - Direct access only to local memory (private address space)
  - Communication via explicit send/receive messages (library or system calls)



Message-Passing Abstraction



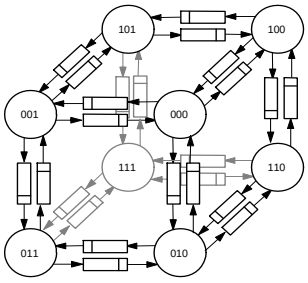
- Send specifies receiving process and buffer to be transmitted
- Receive specifies sending process and buffer to receive into
- Optional tag on send and matching rule on receive
  - Matching rule: match a specific tag *t* or any tag
- Combination of send and matching receive achieves …
  - Pairwise synchronization event
  - Memory to memory copy of message
- Overheads: copying, buffer management, protection

Variants of Send and Receive

- Parallel programs using send and receive are quite structured
  - Most often, all nodes execute identical copies of a program
  - Processes can name each other using a simple linear ordering
- Blocking send:
  - Sender sends a request and waits until the reply is returned
- Non-blocking send:
  - Sender sends a message and continues without waiting for a reply
- Blocking receive:
  - Receiver blocks if it tries to receive a message that has not arrived
- Non-blocking receive:
  - Receiver simply posts a receive without waiting for sender
  - Requires completion detection

Evolution of Message-Passing Machines

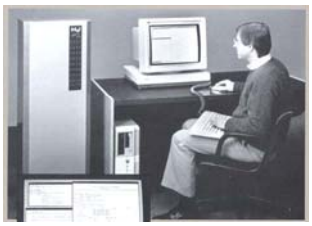
- Early machines: FIFO on each link
  - Hardware close to programming model
  - Synchronous send/receive operations
  - Topology central (hypercube algorithms)



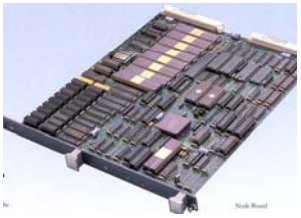
CalTech Cosmic Cube (Seitz), 1981
64 Intel 8086/87 processors
6-dimensional hypercube

Diminishing Role of Topology

- Any-to-any pipelined routing
  - Better performance than store&forward
  - Node-network interface dominates communication time
- Shift to general links
  - DMA, enabling non-blocking operations
  - Non-blocking send
    - Sender initiates send and continues
    - Data is buffered by system at destination until receive
- Simplifies programming
- Richer design space of interconnect
  - Grids, Hypercubes, etc.



Intel iPSC (1985) → iPSC/860 (1990)

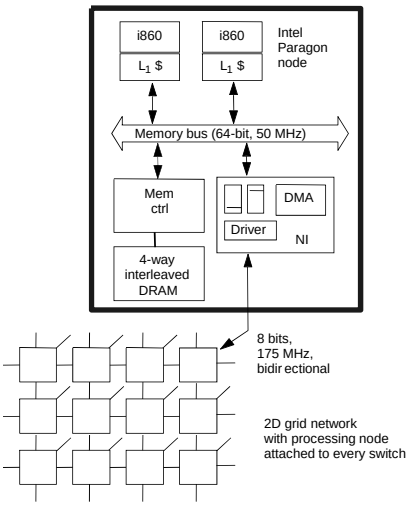


Intel Paragon (1992)

Sandia's Intel Paragon XP/S-based SuperComputer

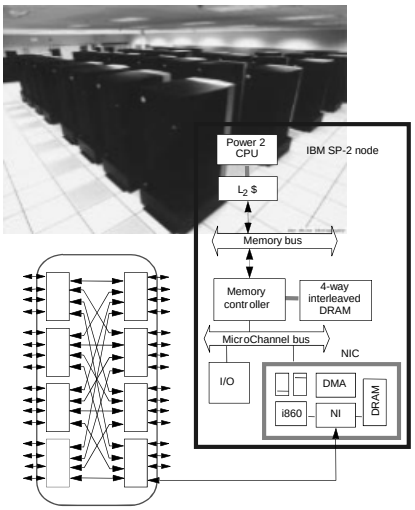
Each card is an SMP with two or more i860 processors and a network interface chip connected to the cache-coherent memory bus.

Each node has a DMA engine to transfer contiguous chunks of data to and from the network at a high rate.



IBM SP (1994 – 2002)

- Made out of essentially complete RS6000 workstations
- Packaged workstations into standing racks
- Network interface card connected to I/O bus
  - Contains drivers for the actual link
  - Memory to buffer message data
  - DMA engine
  - i860 processor to move data between host and network
- Network
  - Butterfly-like structure
  - Cascaded 8 × 8 crossbar switches

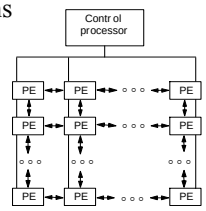


Toward Architectural Convergence

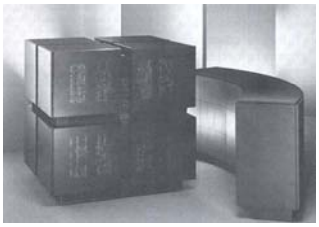
- Evolution and role of software has blurred boundary
  - Send/receive supported on shared memory machines via shared buffers
  - Global address space on message passing using process and local address
  - Shared virtual address space (page level) on message passing architectures
- Hardware organization converging too
  - Tighter NI integration even for message passing
    - Lower latency, higher bandwidth
  - Remote memory access in shared memory converted to network 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

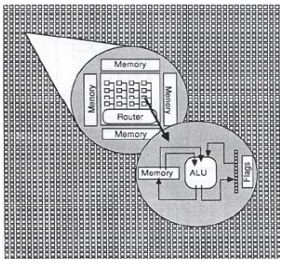
Data Parallel Systems

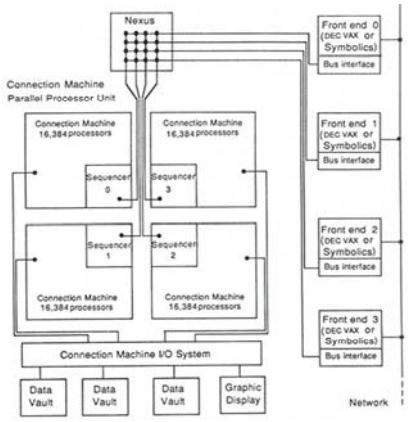
- Programming model
  - Operations performed in parallel on each element of data structure
  - Logically single thread of control, performs sequential or parallel steps
  - Conceptually, a processor associated with each data element
- Architectural model: Single Instruction Multiple Data (SIMD)
  - Array of many simple, cheap processors with little memory to 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 important scientific computations
  - Communication is often with neighboring elements



Engineering: The Connection Machine





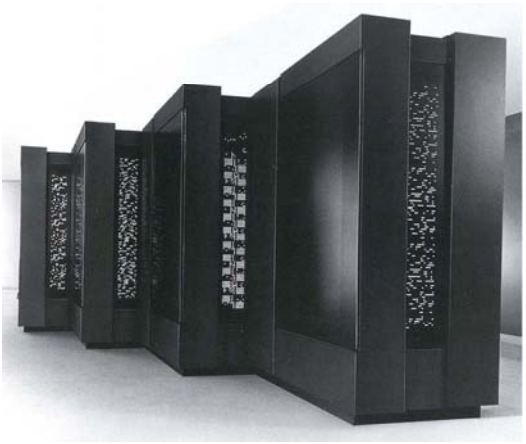


(Tucker, IEEE Computer, August 1988)

Evolution and Convergence

- Rigid control structure (SIMD in Flynn's taxonomy)
  - SISD = uniprocessor, MIMD = multiprocessor
- Replaced by vectors in mid-1970s
  - More flexible w.r.t. memory layout and easier to manage
- Revived in mid-1980s when 32-bit datapath slices just fit on chip
- Other reasons for demise
  - Simple, regular applications have good locality, can do well anyway
  - Loss of applicability due to hardwiring data parallelism
  - MIMD machines as effective for data parallelism and more general
- Programming model evolved to SPMD (Single Program Multiple Data)
  - Contributes need for fast global synchronization
  - Structured global address space
    - Implemented with either shared memory or message passing

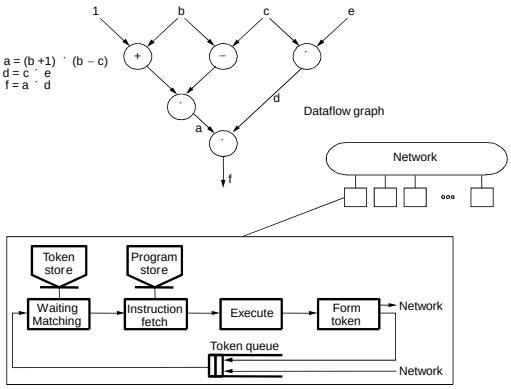
Evolution of Connection Machine: CM-5



- Repackaged SparcStation
  - 4 per board
- Fat-Tree network
- Control network for global synchronization
- Supports data parallel and message passing programming models

Dataflow Architectures

- Represent computation as a graph of essential dependences
- Node execution activated by availability of operands
- Tokens (messages) carry data, tag, and instruction address
- Tag compared with others in matching store; match fires execution
- Graph can have multiple dynamic activations
- Ability to name operations
- Synchronization and dynamic scheduling at each node

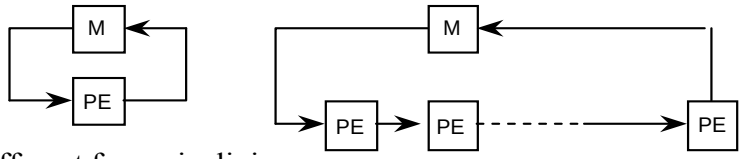


Evolution and Convergence

- 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
  - Separation of programming model from hardware (like data-parallel)
- Lasting contributions
  - Integration of communication with thread (handler) generation
  - Tightly integrated communication and fine-grained synchronization
  - Remains as a useful concept for compilers and software

Systolic Architectures

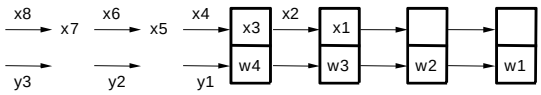
- Special purpose architectures driven by advances in VLSI
  - Implement algorithms directly on chips connected in regular pattern
  - Replace single processor with array of regular processing elements
  - Orchestrate data flow for high throughput with less memory access



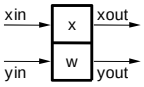
- Different from pipelining
  - Nonlinear array structure, multi-direction data flow, each PE may have (small) local instruction and data memory
- Different from SIMD: each PE may do something different

Example of a Systolic Array Computation

$$y(i) = w1 \times x(i) + w2 \times x(i+1) + w3 \times x(i+2) + w4 \times x(i+3)$$



Example:
1-D convolution

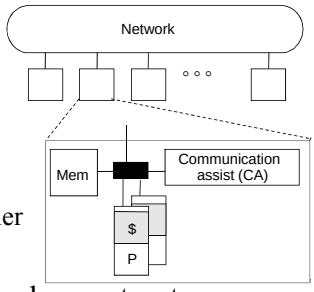


xout = x
x = xin
yout = yin + w × xin

- Practical realizations (e.g. iWARP) use quite general processors
  - Enable variety of algorithms on same hardware
- But dedicated interconnect channels
  - Data transfer directly from register to register across channel
- Communication patterns
  - Regular and highly localized

Convergence: Generic Parallel Architecture

- Generic modern multiprocessor
  - Collection of essentially complete computers
- Node
  - One or more processors
  - Memory system plus *communication assist*
  - Network interface and communication controller
- Scalable network
- Separation of programming model from hardware structure
  - But programming model place some requirements on network and CA
  - What operations should be supported, how efficient, etc.
    - Shared memory: access to and caching of remote and shared memory
    - Message Passing: quick message initiation and response to incoming ones
    - Data Parallel and Systolic: emphasis on fast global synchronization
    - Dataflow: fast dynamic scheduling based on incoming messages



Fundamental Design Issues

- Key concepts that affect the design of software/hardware layers
  - **Naming:** How are logically shared data and/or processes referenced?
  - **Operations:** What operations are provided on named objects or data
  - **Ordering:** How are accesses to data ordered and coordinated?
  - **Replication:** How are data replicated to reduce communication?
  - **Communication Cost and Performance:**
    - Latency: time taken to complete an operation
    - Bandwidth: rate of performing operations
    - Cost: impact on execution time of a program
      - How to reduce the overhead of initiating a transfer?
      - How to overlap useful computation with communication?
- Design issues apply at all layers

Sequential Programming Model

- **Naming:** can name any variable in virtual address space
  - Operating System allocates and maps to physical addresses
  - Hardware does translation
- **Operations:** loads and stores
- **Ordering:** sequential program order
- **Replication:** transparently done in caches
- **Performance:**
  - Compilers and hardware violate sequential program order
    - But maintain dependences to ensure correctness
  - Compiler: instruction reordering and register allocation
  - Hardware: out of order execution, pipeline bypassing, write buffers

Shared Address Programming Model

- **Naming:** any thread can name any variable in shared address space
- **Operations:** loads, stores, and atomic read-modify-write
- **Ordering:** various models, here is a simple one
  - Within a process/thread: sequential program order
  - Across threads: some interleaving (as in time-sharing)
    - No assumption is made on relative thread speed
  - Additional ordering through synchronization
  - Again, compilers/hardware can violate sequential ordering
    - But without violating dependences and memory access ordering requirements

Synchronization in Shared Address

- Required when implicit ordering is not enough
- **Mutual exclusion (locks)**
  - Operations on shared data can be performed by only one thread at a time
  - Similar to a room that only one person can enter at a time
  - No required ordering: threads can enter in any order
  - Serialization of access which degrades performance
- **Event synchronization**
  - Ordering of events to preserve dependences
    - Example: producer → consumer type of synchronization
  - Three main types:
    - Point-to-point: involving a pair of threads
    - Global: involving all threads
    - Group: involving a subset of threads

Message Passing Programming Model

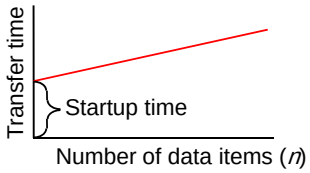
- **Naming:** processes can name other processes
  - No shared address space but a named process space
- **Operations:** explicit communication through *send* and *receive*
  - Send transfers data from private address space to another process
  - Receive copies data from process to private address space
- **Ordering:**
  - Program order within a process
  - Send and receive can provide point to point synch between processes
  - Mutual exclusion inherent because there is no shared address space
- Can construct global address space:
  - Process number + address within process address space
  - But no direct operations on these names

Communication and Replication

- Inherently related
- Replication reduces data transfer/communication
  - Depending on naming model
- Uniprocessor: caches reduce communication with memory
  - Coherence problem: multiple copies in memory hierarchy
- Message Passing communication and replication
  - Receive replicates message in the destination address space
  - Replication is explicit in software
- Shared Address Space communication and replication
  - A load brings in data, that can replicate transparently in cache
  - Hardware caches do replicate blocks in shared physical address space
  - No explicit renaming, many copies for same name: coherence problem

Linear Model of Data Transfer Time

- **Transfer time (n)** $= T_0 + n/B$
  - T_0: Startup cost to initiate transfer
  - n: amount of data (number of bytes)
  - B: Transfer rate (bytes per second)



- Useful for message passing, memory access, vector operations
- As n increases, bandwidth approaches asymptotic rate B
- Size at half peak bandwidth: **half-power point**

$$n_{1/2} = T_0 B$$

- But linear model is not enough
  - When can next transfer be initiated?
  - Can useful work be overlapped?

Simple Example

- IBM SP-2 with MPI is quoted to have the following:
  - Startup time = 35 µs = 35000 ns
  - Transfer rate = 1 double (8 bytes) per 230 ns
  - Clock cycle time = 4.2 ns
- Normalizing the startup time and transfer rate to clock cycles
  - Startup time = 35000 ns / 4.2 ns = 8333 cycles
  - Transfer rate = 1 double / 55 cycles
- Conclusions:
  - In many cases, the startup time dominates the communication time
  - Startup time cannot be ignored unless n is very large
  - A lot of computation cycles are needed to hide communication latency
  - Linear model is not always accurate due to contention delays

Communication Cost Model

- Communication Time per message =
  Overhead + Occupancy + Network Delay + Contention
- Overhead: time to initiate the transfer (tell the assist to start)
- Occupancy: time to pass through the slowest component along path
- Occupancy = n / B to transfer n bytes through the slowest component that has a bandwidth of B bytes per second.
- Network delay: sum of delays along the network path
- Contention: time waiting for resource, increases communication time
- Communication Cost = Frequency×(Communication time – Overlap)
- Overlapping the communication time with useful computation reduces the communication cost

Summary of Design Issues

- Functional and performance design issues apply at all layers
- Functional design issues:
  - Naming, operations, and ordering
- Performance issues:
  - Organization, latency, bandwidth, overhead, and occupancy
- Replication and communication are deeply related
- Goals of architect:
  - Design against frequency and type of operations at communication abstraction.
  - Hardware/software tradeoffs – what primitives at the hardware layer, what primitives at the software layer.