18-742 Fall 2012
Parallel Computer Architecture
Lecture 3: Programming Models and Architectures

Prof. Onur Mutlu
Carnegie Mellon University
9/12/2012

Reminder: Assignments for This Week

1. Review two papers from ISCA 2012 – due September 11, 11:59pm.

2. Attend NVIDIA talk on September 10 – write an online review of the talk; due September 11, 11:59pm.

3. Think hard about
   - Literature survey topics
   - Research project topics

4. Examine survey and project topics from Spring 2011

5. Find your literature survey and project partner

Late Review Assignments

- Even if you are late, please submit your reviews
- You will benefit from this

Reminder: Reviews Due Sunday

- Sunday, September 16, 11:59pm.
- Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS 2009.
- Suleman et al., "Data Marshaling for Multi-core Architectures," ISCA 2010.
- Joao et al., "Bottleneck Identification and Scheduling in Multithreaded Applications," ASPLOS 2012.

Programming Models vs. Architectures

What Will We Cover in This Lecture?

- Hill, Jouppi, Sohi, "Multiprocessors and Multicomputers," pp. 551-560, in Readings in Computer Architecture.

- Culler, Singh, Gupta, Chapter 1 (Introduction) in "Parallel Computer Architecture: A Hardware/Software Approach."

Programming Models vs. Architectures

- Five major models
  - (Sequential)
  - Shared memory
  - Message passing
  - Data parallel (SIMD)
  - Dataflow
  - Systolic

- Hybrid models?

Shared Memory vs. Message Passing

- Are these programming models or execution models supported by the hardware architecture?
- Does a multiprocessor that is programmed by "shared memory programming model" have to support a shared address space processors?
- Does a multiprocessor that is programmed by "message passing programming model" have to have no shared address space between processors?

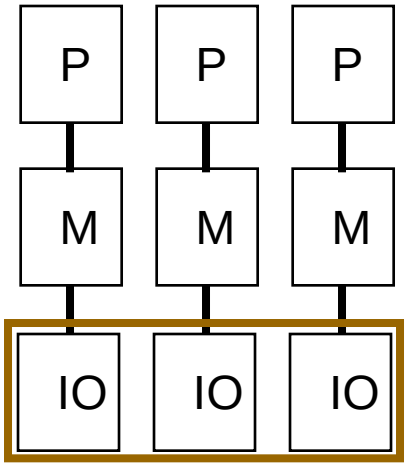
Programming Models: Message Passing vs. Shared Memory

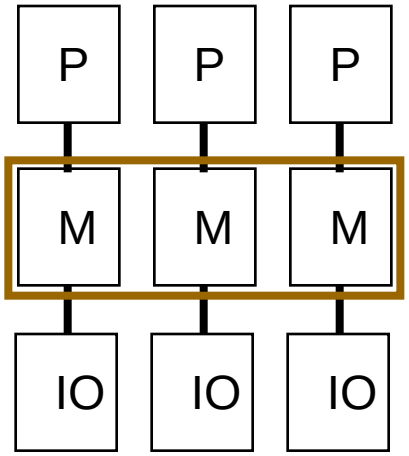
- Difference: how communication is achieved between tasks
- Message passing programming model
  - Explicit communication via messages
  - Loose coupling of program components
  - Analogy: telephone call or letter, no shared location accessible to all
- Shared memory programming model
  - Implicit communication via memory operations (load/store)
  - Tight coupling of program components
  - Analogy: bulletin board, post information at a shared space
- Suitability of the programming model depends on the problem to be solved. Issues affected by the model include:
  - Overhead, scalability, ease of programming, bugs, match to underlying hardware, …

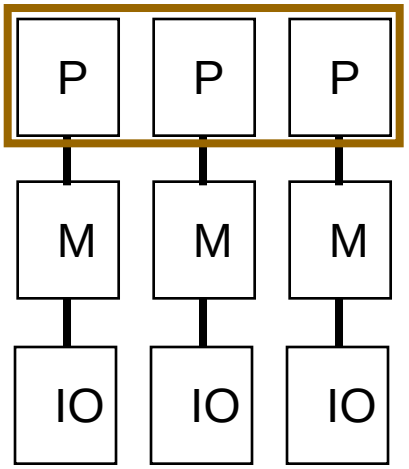
Message Passing vs. Shared Memory Hardware

- Difference: how task communication is supported in hardware
- Shared memory hardware (or machine model)
  - All processors see a global shared address space
    - Ability to access all memory from each processor
  - A write to a location is visible to the reads of other processors
- Message passing hardware (machine model)
  - No global shared address space
  - Send and receive variants are the only method of communication between processors (much like networks of workstations today, i.e. clusters)
- Suitability of the hardware depends on the problem to be solved as well as the programming model.

Message Passing vs. Shared Memory Hardware







Join At:	I/O (Network)	Memory	Processor
Program With:	Message Passing	Shared Memory	(Dataflow/Systolic), Single-Instruction Multiple-Data (SIMD) ==> Data Parallel

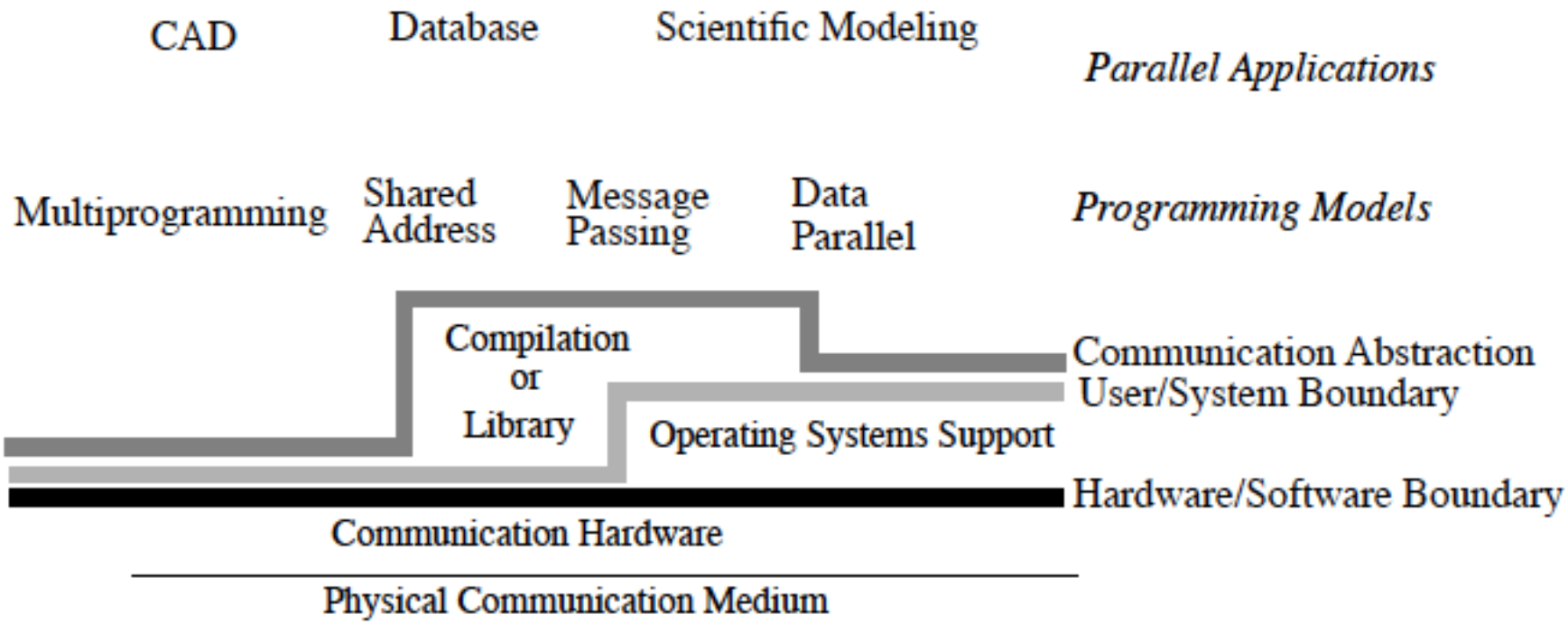
Programming Model vs. Hardware

- Most of parallel computing history, there was no separation between programming model and hardware
  - Message passing: Caltech Cosmic Cube, Intel Hypercube, Intel Paragon
  - Shared memory: CMU C.mmp, Sequent Balance, SGI Origin.
  - SIMD: ILLIAC IV, CM-1

- However, any hardware can really support any programming model
- Why?
  - Application → compiler/library → OS services → hardware

Layers of Abstraction

- Compiler/library/OS map the communication abstraction at the programming model layer to the communication primitives available at the hardware layer



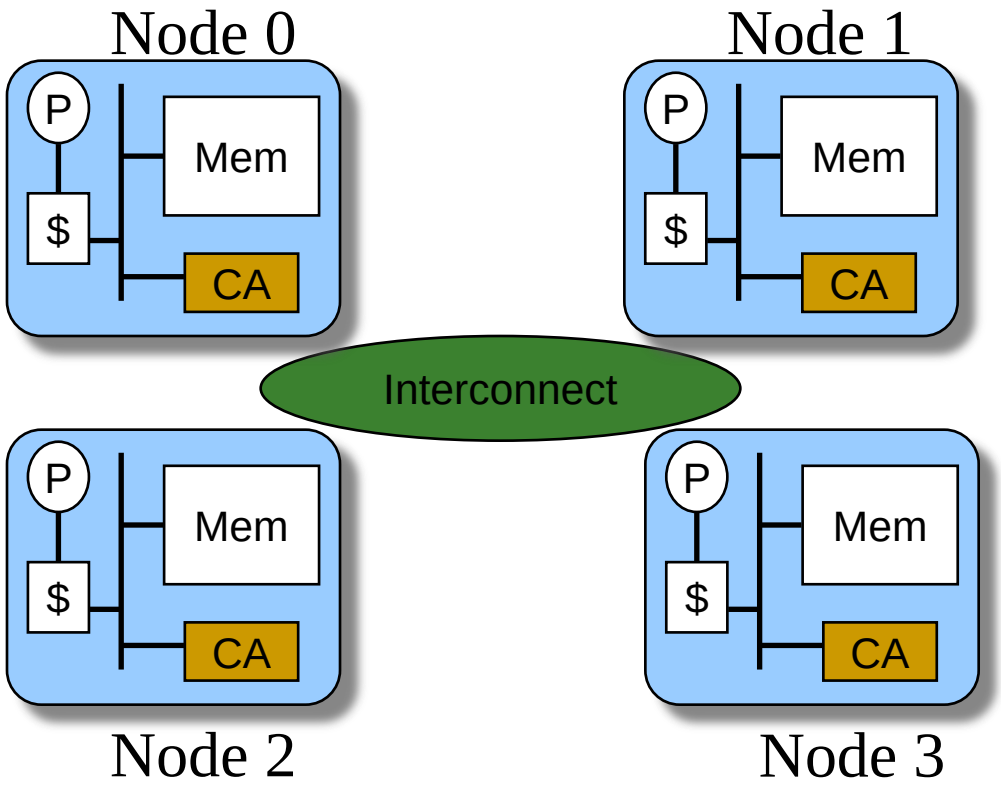
Programming Model vs. Architecture

- Machine → Programming Model
  - Join at network, so program with message passing model
  - Join at memory, so program with shared memory model
  - Join at processor, so program with SIMD or data parallel

- Programming Model → Machine
  - Message-passing programs on message-passing machine
  - Shared-memory programs on shared-memory machine
  - SIMD/data-parallel programs on SIMD/data-parallel machine

- Isn't hardware basically the same?
  - Processors, memory, interconnect (I/O)
  - Why not have generic parallel machine and program with model that fits the problem?

A Generic Parallel Machine



- Separation of programming models from architectures
- All models require communication
- Node with processor(s), memory, communication assist

Simple Problem

```
for i = 1 to N
        A[i] = (A[i] + B[i]) * C[i]
        sum = sum + A[i]
```

- How do I make this parallel?

Simple Problem

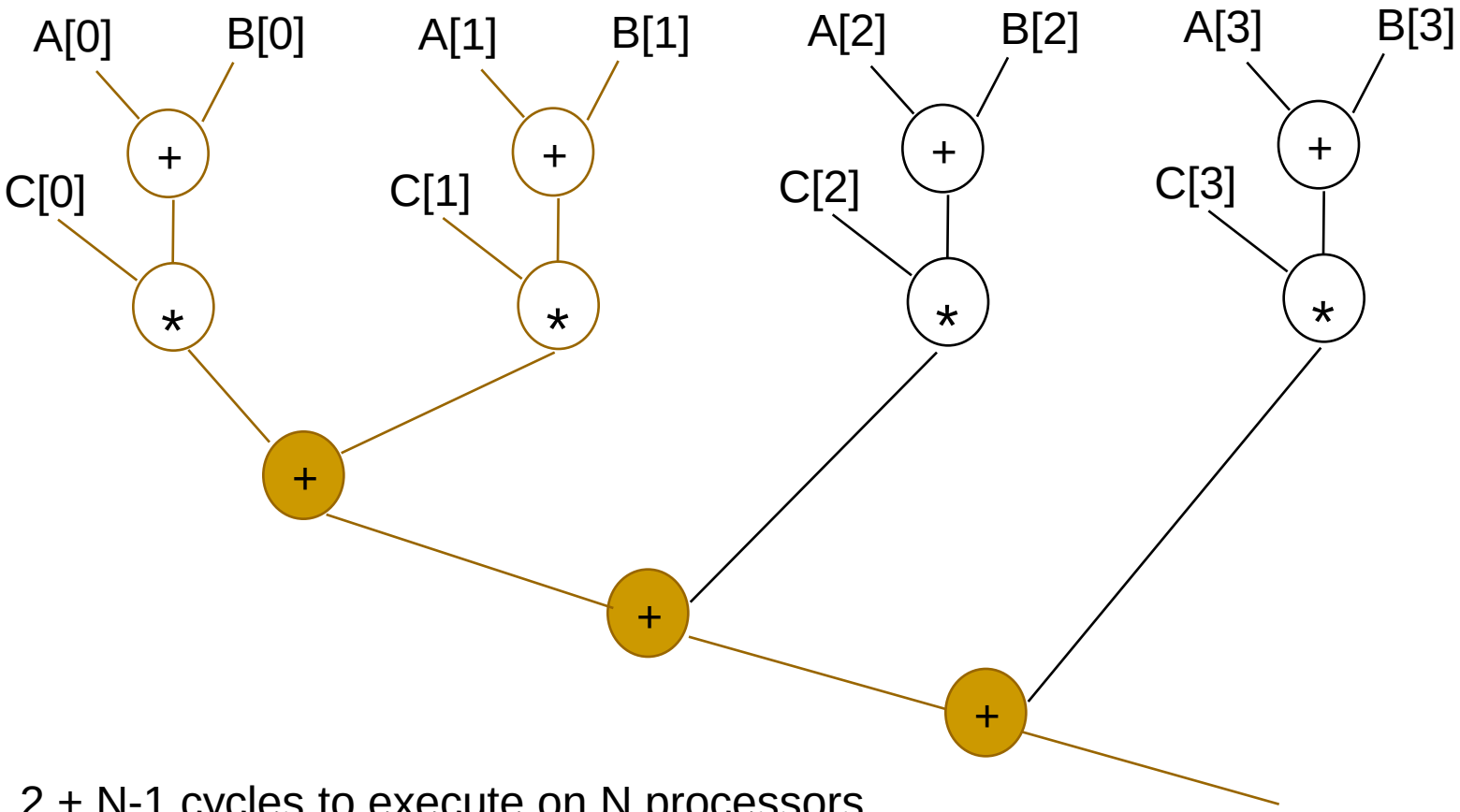
```
for i = 1 to N
        A[i] = (A[i] + B[i]) * C[i]
        sum = sum + A[i]
```

- Split the loops → Independent iterations

```
for i = 1 to N
        A[i] = (A[i] + B[i]) * C[i]
for i = 1 to N
        sum = sum + A[i]
```

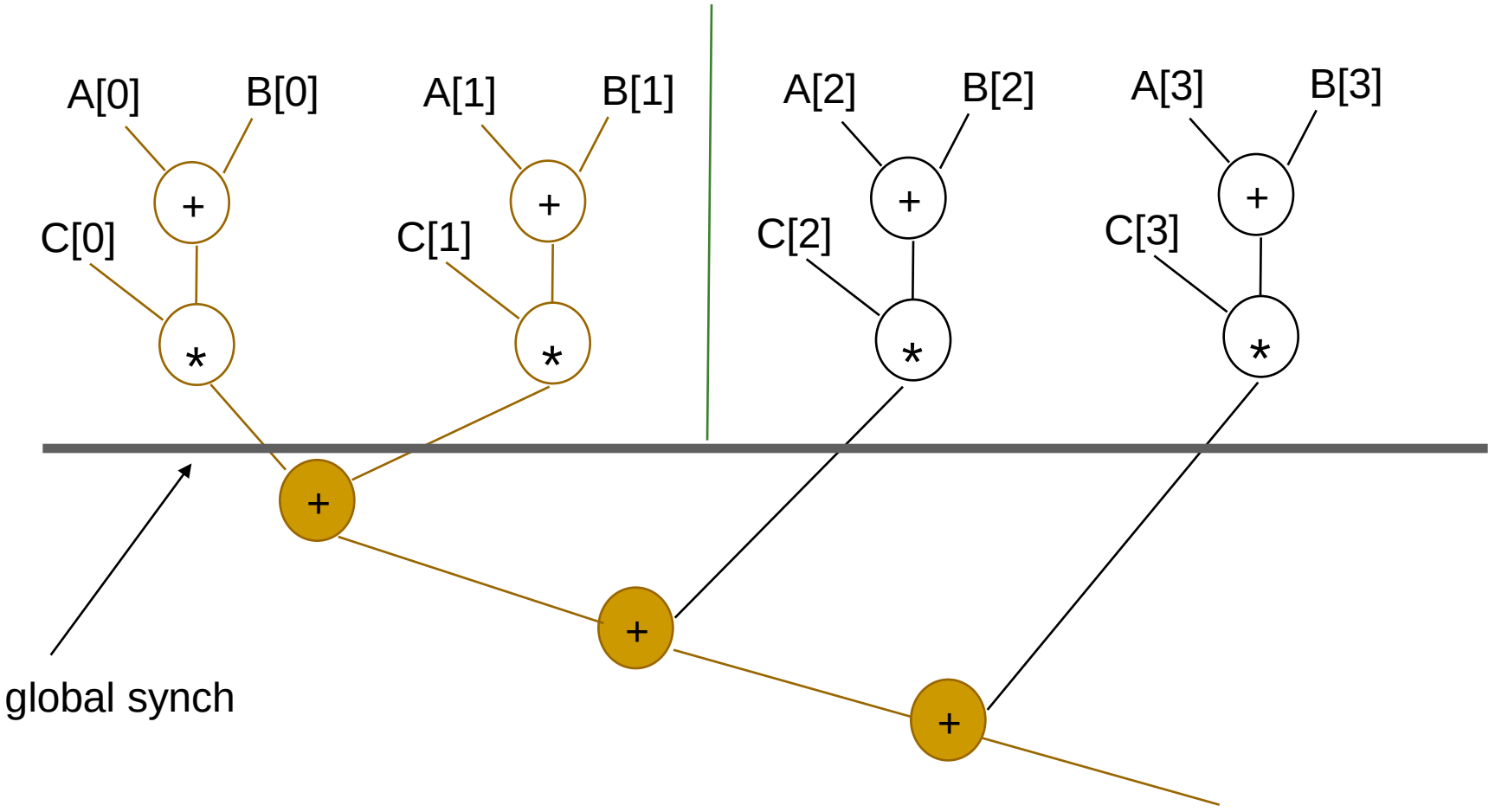
- Data flow graph?

Data Flow Graph

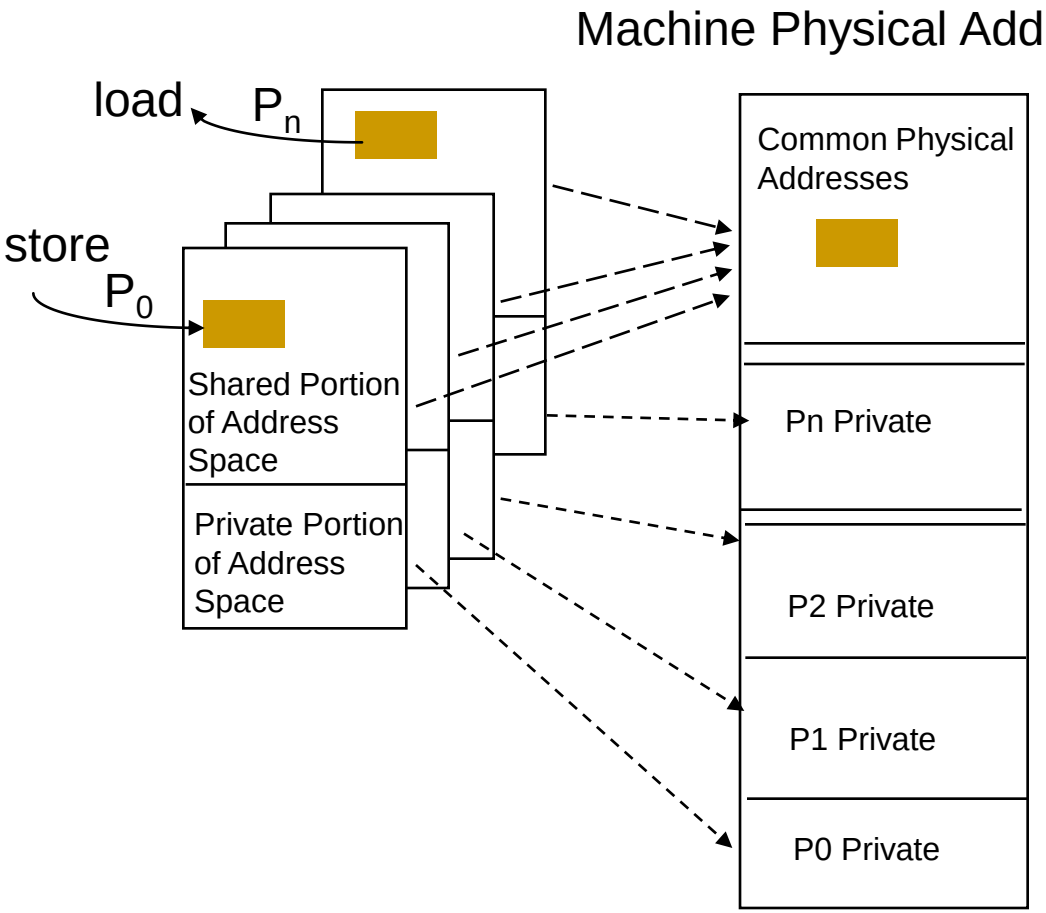


2 + N-1 cycles to execute on N processors
what assumptions?

Partitioning of Data Flow Graph

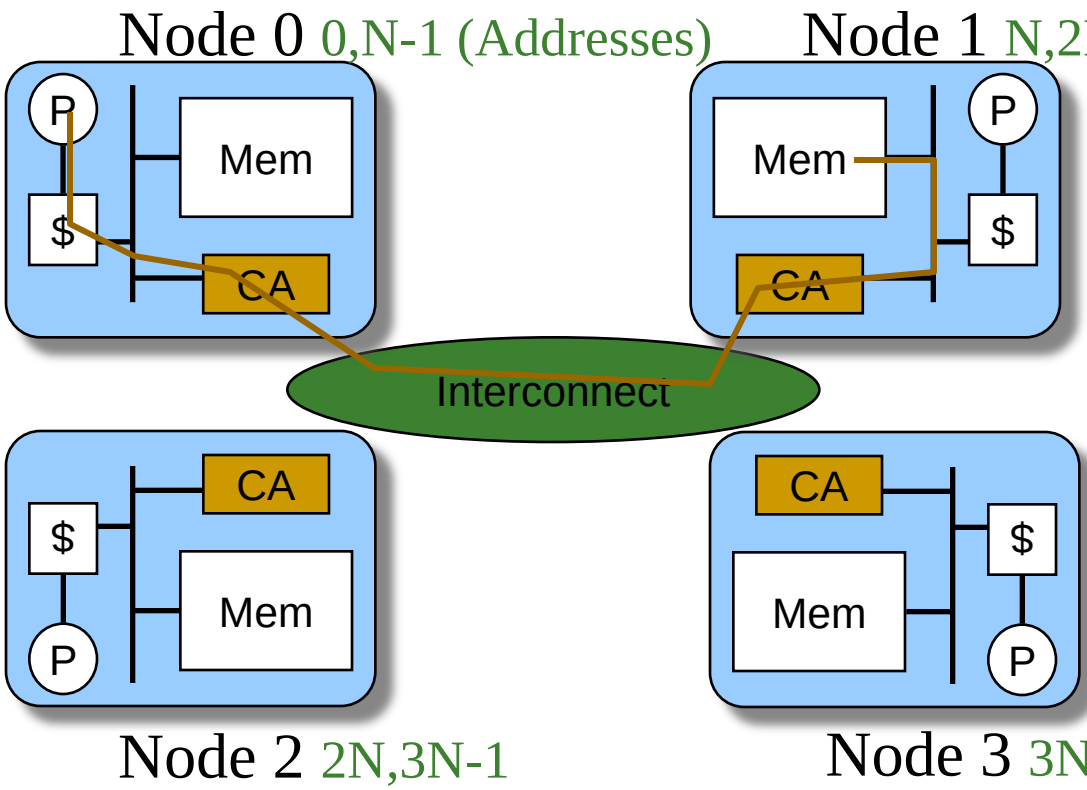


Shared (Physical) Memory



- Communication, sharing, and synchronization with store / load on shared variables
- Must map virtual pages to physical page frames
- Consider OS support for good mapping

Shared (Physical) Memory on Generic MP



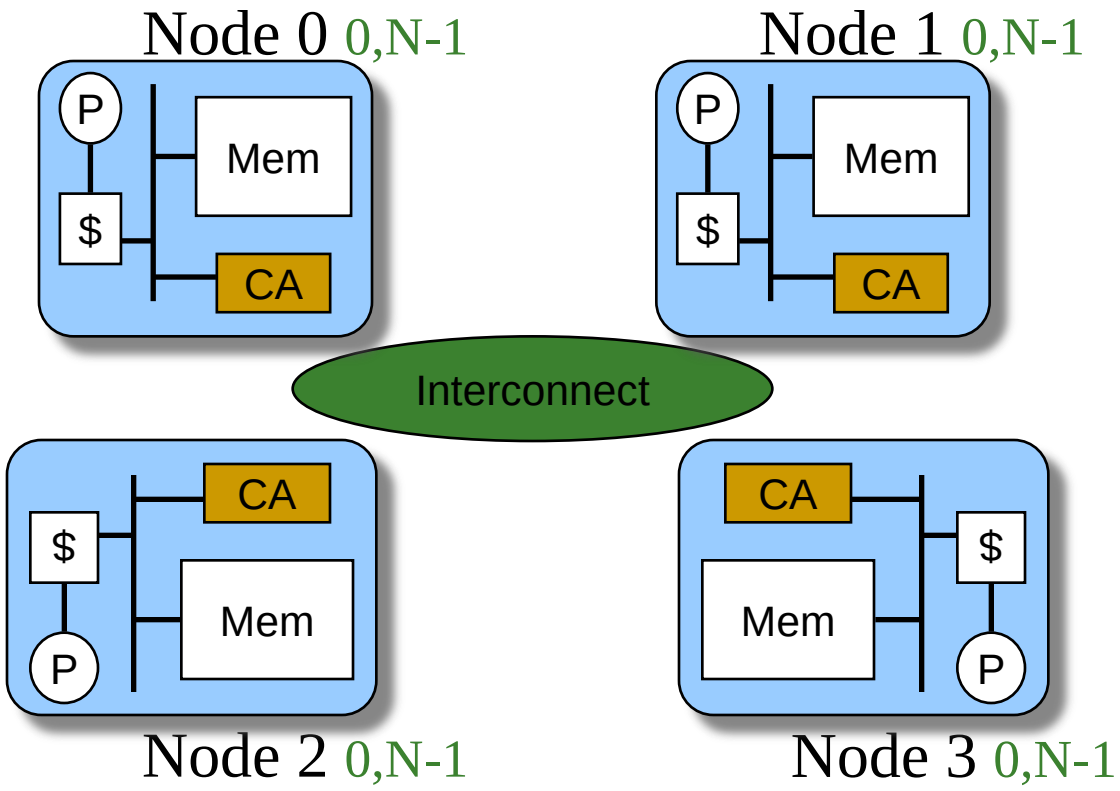
Keep private data and frequently used shared data on same node as computation

Return of The Simple Problem

```
private int i, my_start, my_end, mynode;
shared float A[N], B[N], C[N], sum;
for i = my_start to my_end
        A[i] = (A[i] + B[i]) * C[i]
GLOBAL_SYNCH;
if (mynode == 0)
        for i = 1 to N
                sum = sum + A[i]
```

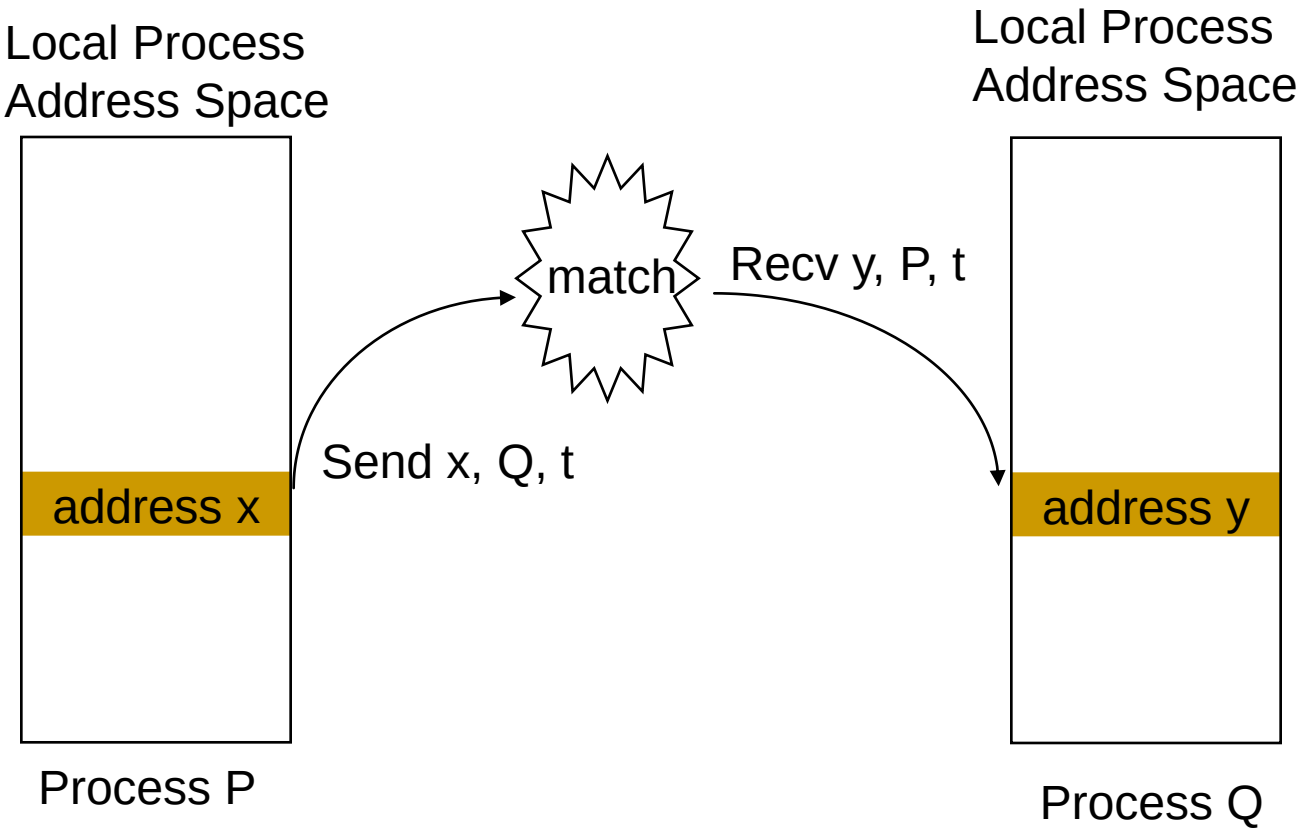
- Can run this on any shared memory machine

Message Passing Architectures



- Cannot directly access memory on another node
- IBM SP-2, Intel Paragon
- Cluster of workstations

Message Passing Programming Model



- User level send/receive abstraction
  - local buffer (x,y), process (Q,P) and tag (t)
  - naming and synchronization

The Simple Problem Again

```
int i, my_start, my_end, mynode;
float A[N/P], B[N/P], C[N/P], sum;
for i = 1 to N/P
        A[i] = (A[i] + B[i]) * C[i]
        sum = sum + A[i]
if (mynode != 0)
        send (sum,0);
if (mynode == 0)
        for i = 1 to P-1
                recv(tmp,i)
                sum = sum + tmp
```

- Send/Recv communicates and synchronizes
- P processors

Separation of Architecture from Model

- At the lowest level shared memory model is all about sending and receiving messages
  - HW is specialized to expedite read/write messages using load and store instructions
- What programming model/abstraction is supported at user level?
- Can I have shared-memory abstraction on message passing HW? How efficient?
- Can I have message passing abstraction on shared memory HW? How efficient?

Challenges in Mixing and Matching

- Assume prog. model same as ABI (compiler/library → OS → hardware)
- Shared memory prog model on shared memory HW
  - How do you design a scalable runtime system/OS?
- Message passing prog model on message passing HW
  - How do you get good messaging performance?
- Shared memory prog model on message passing HW
  - How do you reduce the cost of messaging when there are frequent operations on shared data?
  - Li and Hudak, "Memory Coherence in Shared Virtual Memory Systems," ACM TOCS 1989.
- Message passing prog model on shared memory HW
  - Convert send/receives to load/stores on shared buffers
  - How do you design scalable HW?

Data Parallel Programming Model

- Programming Model
  - Operations are performed on each element of a large (regular) data structure (array, vector, matrix)
  - Program is logically a single thread of control, carrying out a sequence of either sequential or parallel steps

- The Simple Problem Strikes Back

  $A = (A + B) * C$

  sum = global_sum (A)

- Language supports array assignment

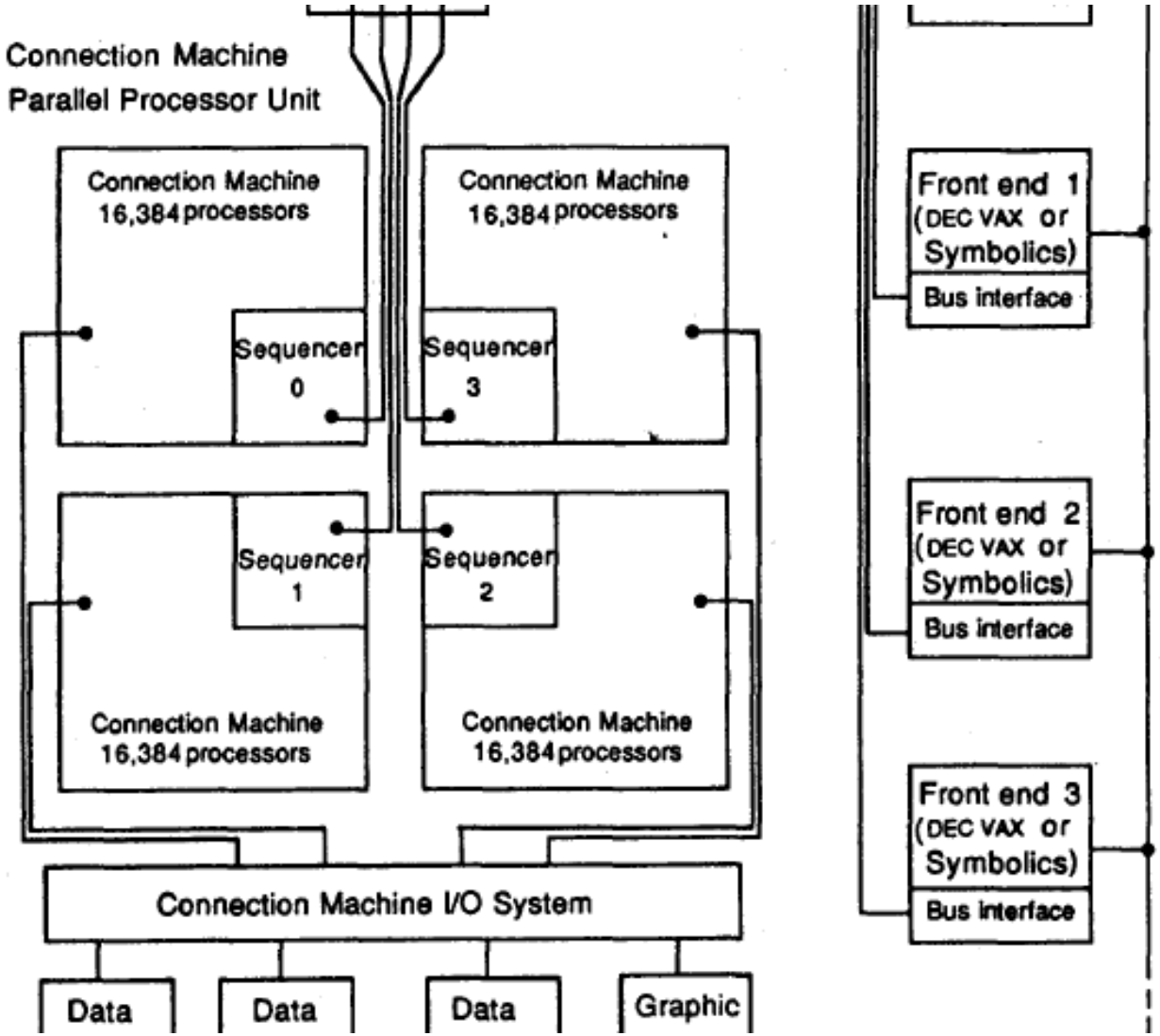
Data Parallel Hardware Architectures (I)

- Early architectures directly mirrored programming model

- Single control processor (broadcast each instruction to an array/grid of processing elements)
  - Consolidates control
- Many processing elements controlled by the master

- Examples: Connection Machine, MPP
  - Batcher, "Architecture of a massively parallel processor," ISCA 1980.
    - 16K bit serial processing elements
  - Tucker and Robertson, "Architecture and Applications of the Connection Machine," IEEE Computer 1988.
    - 64K bit serial processing elements

Connection Machine



Data Parallel Hardware Architectures (II)

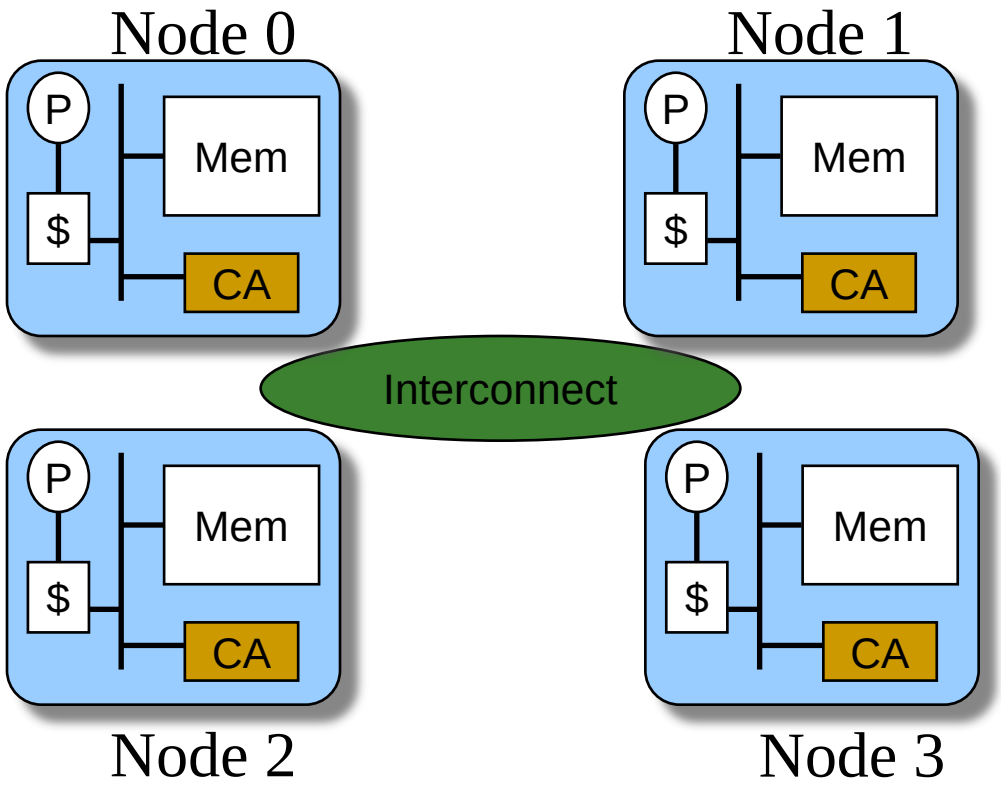
- Later data parallel architectures
  - Higher integration → SIMD units on chip along with caches
  - More generic → multiple cooperating multiprocessors with vector units
  - Specialized hardware support for global synchronization
    - E.g. barrier synchronization

- Example: Connection Machine 5
  - Hillis and Tucker, "The CM-5 Connection Machine: a scalable supercomputer," CACM 1993.
  - Consists of 32-bit SPARC processors
  - Supports Message Passing and Data Parallel models
  - Special control network for global synchronization

Review: Separation of Model and Architecture

- Shared Memory
  - Single shared address space
  - Communicate, synchronize using load / store
  - Can support message passing
- Message Passing
  - Send / Receive
  - Communication + synchronization
  - Can support shared memory
- Data Parallel
  - Lock-step execution on regular data structures
  - Often requires global operations (sum, max, min...)
  - Can be supported on either SM or MP

Review: A Generic Parallel Machine



- Separation of programming models from architectures
- All models require communication
- Node with processor(s), memory, communication assist

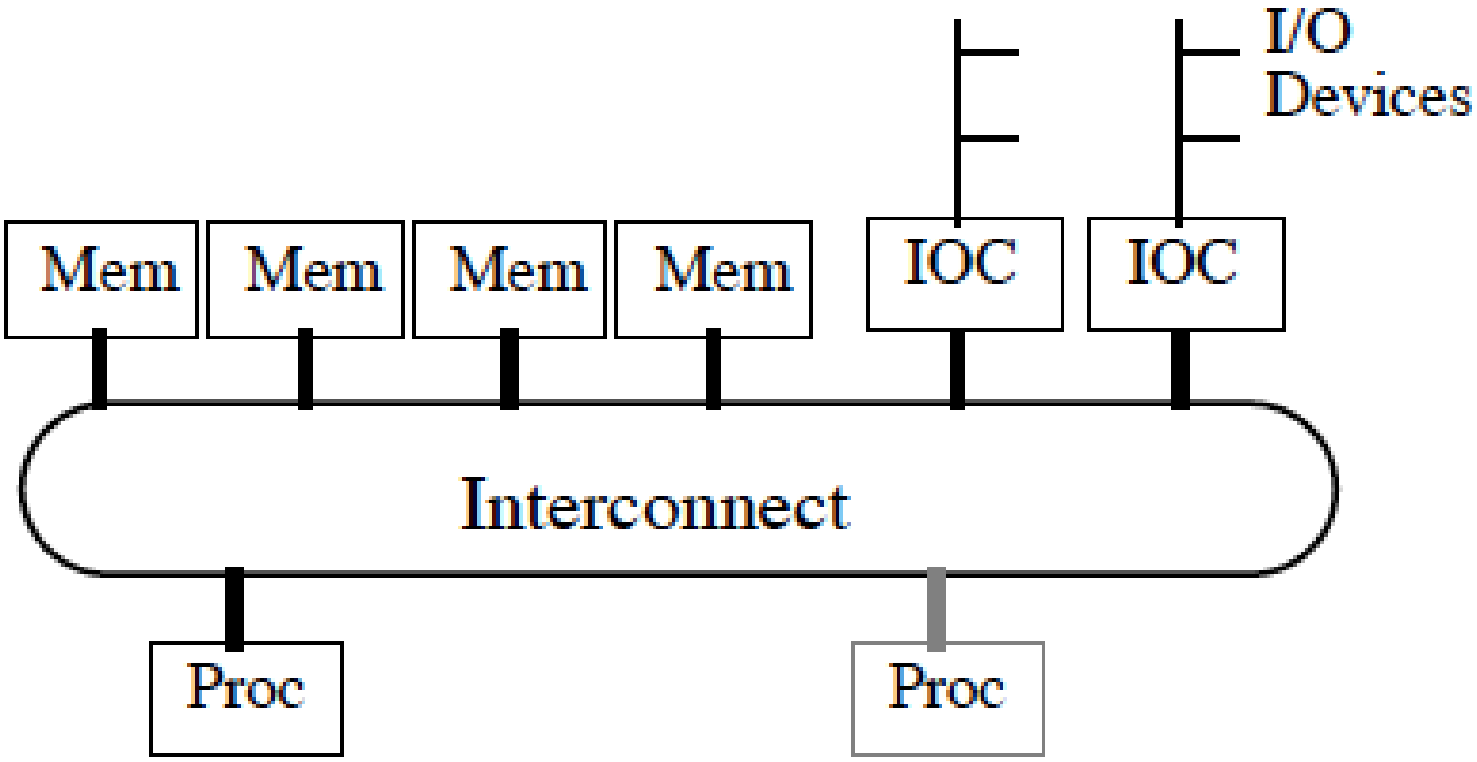
Data Flow Programming Models and Architectures

- A program consists of data flow nodes
- A data flow node fires (fetched and executed) when all its inputs are ready
  - i.e. when all inputs have tokens
- No artificial constraints, like sequencing instructions
- How do we know when operands are ready?
  - Matching store for operands (remember OoO execution?)
  - large associative search!

- Later machines moved to coarser grained dataflow (threads + dataflow across threads)
  - allowed registers and cache for local computation
  - introduced messages (with operations and operands)

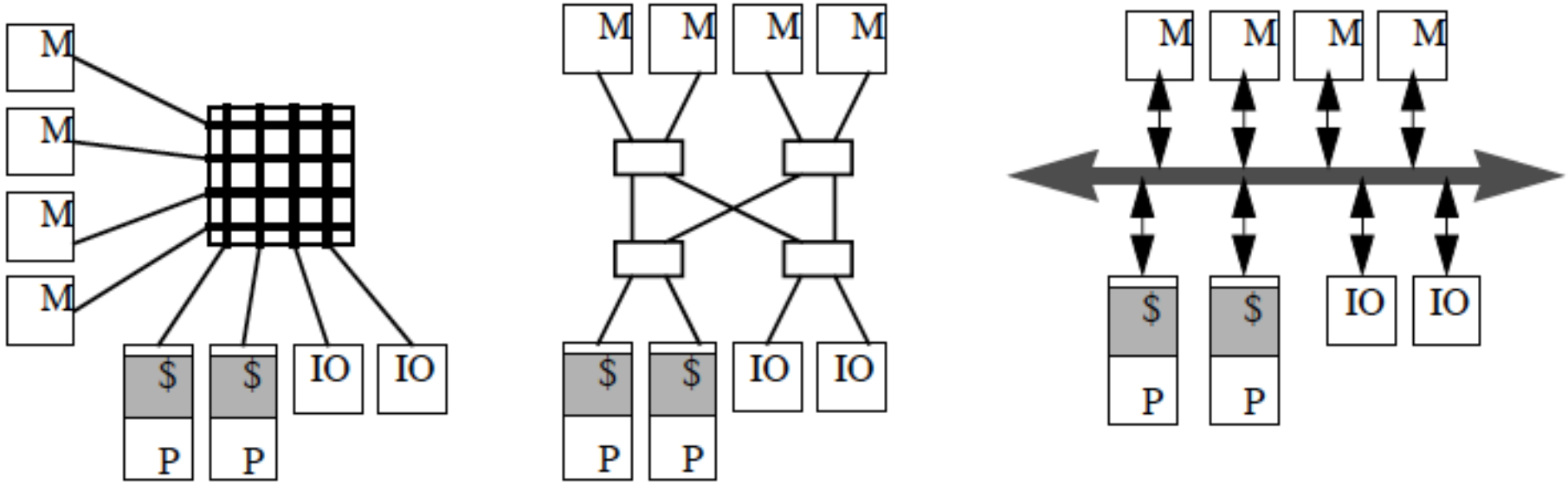
Scalability, Convergence, and Some Terminology

Scaling Shared Memory Architectures



Interconnection Schemes for Shared Memory

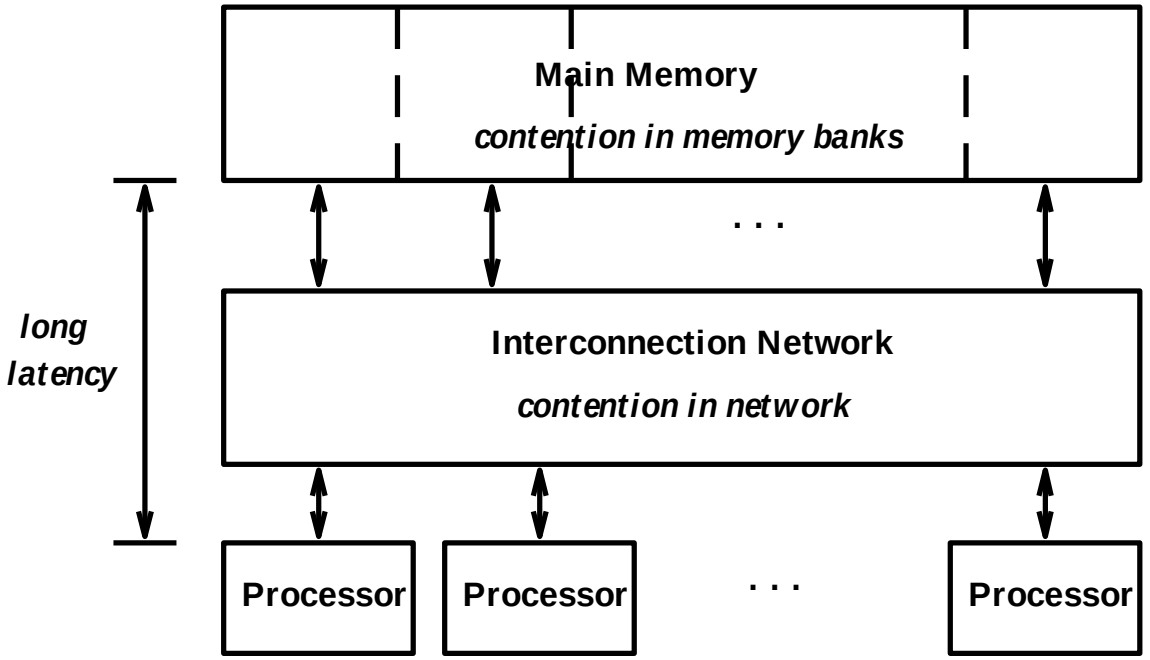
- Scalability dependent on interconnect



(a) Cross-bar Switch (b) Multistage Interconnection Network (c) Bus Interconnect

UMA/UCA: Uniform Memory or Cache Access

- All processors have the same uncontended latency to memory
- Latencies get worse as system grows
- Symmetric multiprocessing (SMP) ~ UMA with bus interconnect

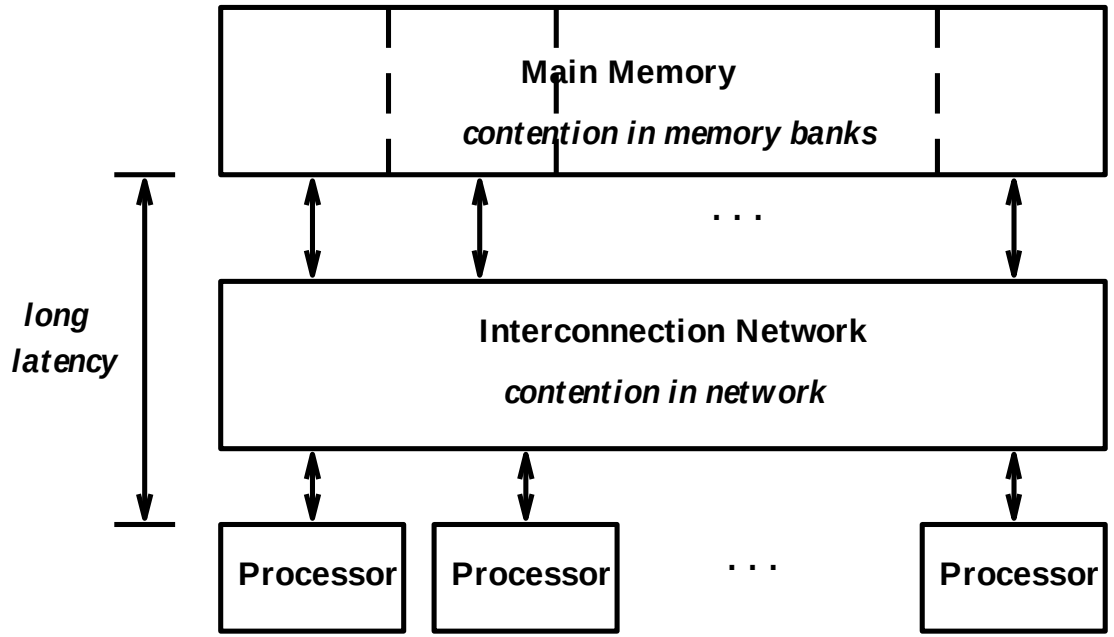


Uniform Memory/Cache Access

+ Data placement unimportant/less important (easier to optimize code and make use of available memory space)

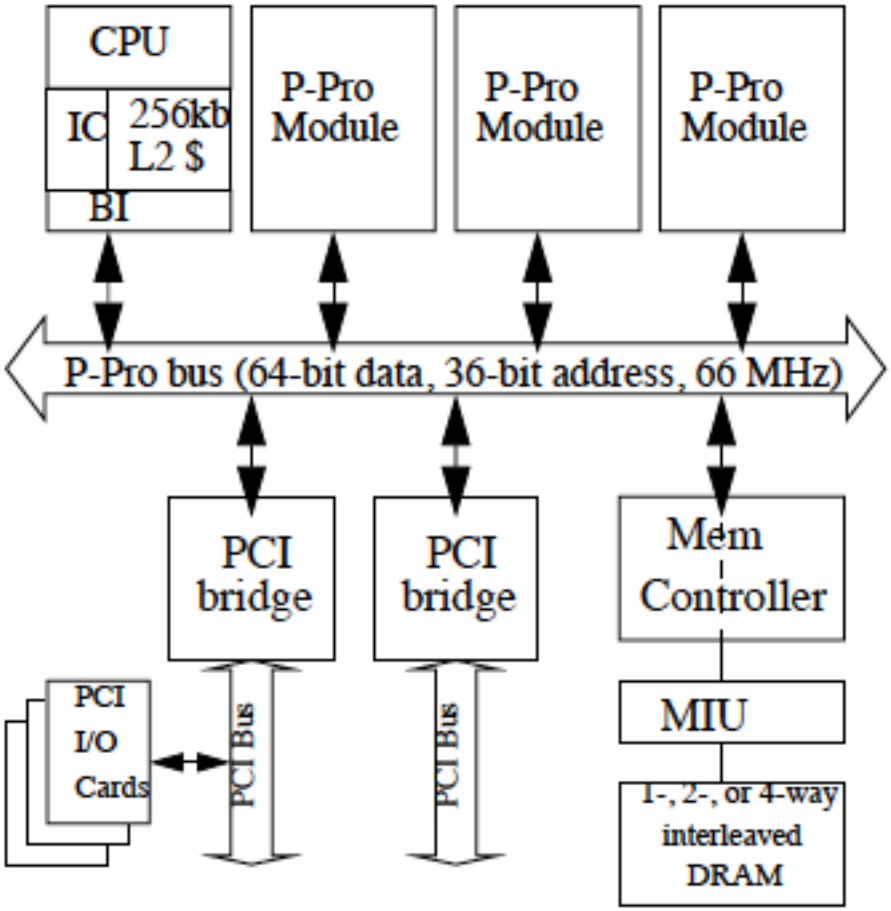
- Scaling the system increases **all** latencies

- Contention could restrict bandwidth and increase latency



Example SMP

- Quad-pack Intel Pentium Pro



How to Scale Shared Memory Machines?

- Two general approaches

- Maintain UMA
  - Provide a scalable interconnect to memory
  - Downside: Every memory access incurs the round-trip network latency

- Interconnect complete processors with local memory
  - NUMA (Non-uniform memory access)
    - Local memory faster than remote memory
  - Still needs a scalable interconnect for accessing remote memory
    - Not on the critical path of local memory access

NUMA/NUCA: NonUniform Memory/Cache Access

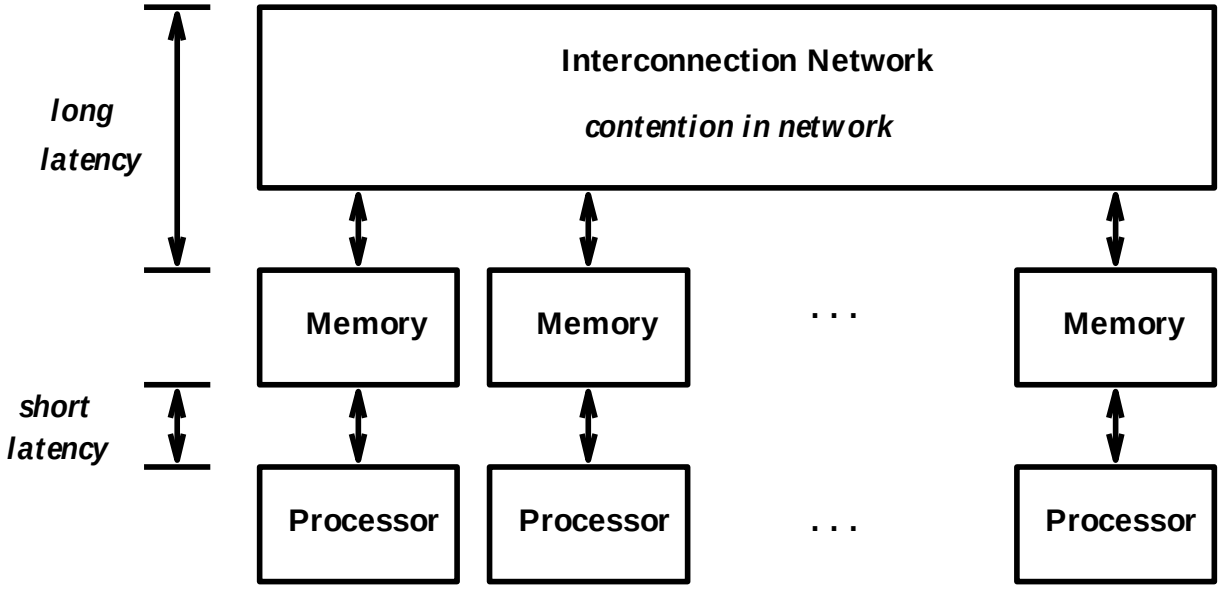
- Shared memory as local versus remote memory

+ Low latency to local memory

- Much higher latency to remote memories

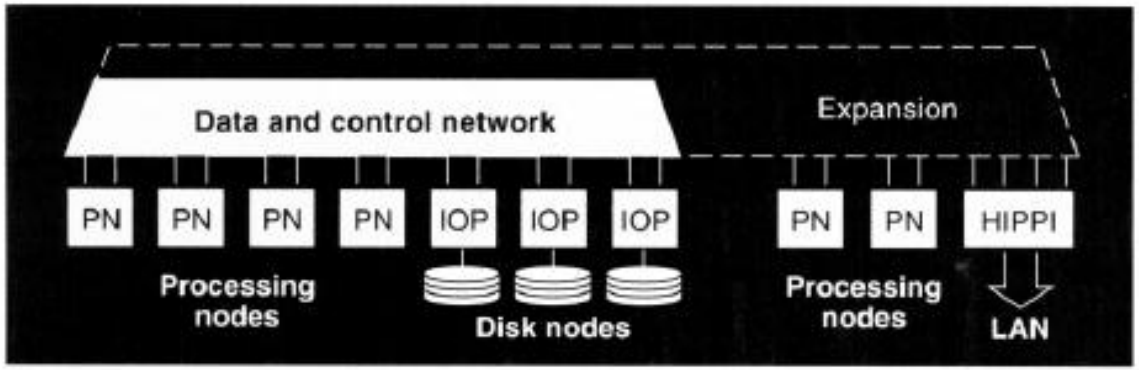
+ Bandwidth to local memory may be higher

- Performance very sensitive to data placement



Example NUMA Machines (I) – CM5

- CM-5
- Hillis and Tucker, "The CM-5 Connection Machine: a scalable supercomputer," CACM 1993.



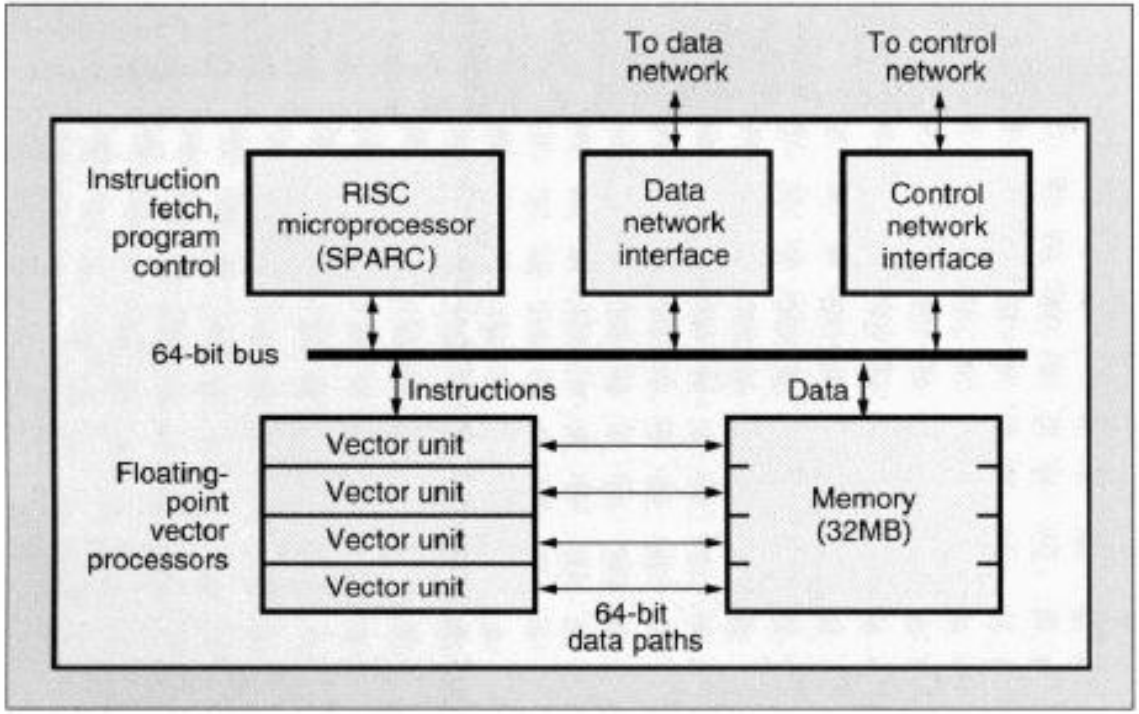


Figure 2.
CM-5 processing node with vector units
Each processing node of the current implementation of the CM-5 consists of a RISC microprocessor, 32MB of memory, and an interface to the control and data interconnection networks. The processor also includes four independent vector units, each with a direct 64-bit path to an 8MB bank of memory. This gives a processing node with a double-precision floating-point rate of 128MFLOPS, and a memory bandwidth of 512MB/sec.

Example NUMA Machines (I) – CM5

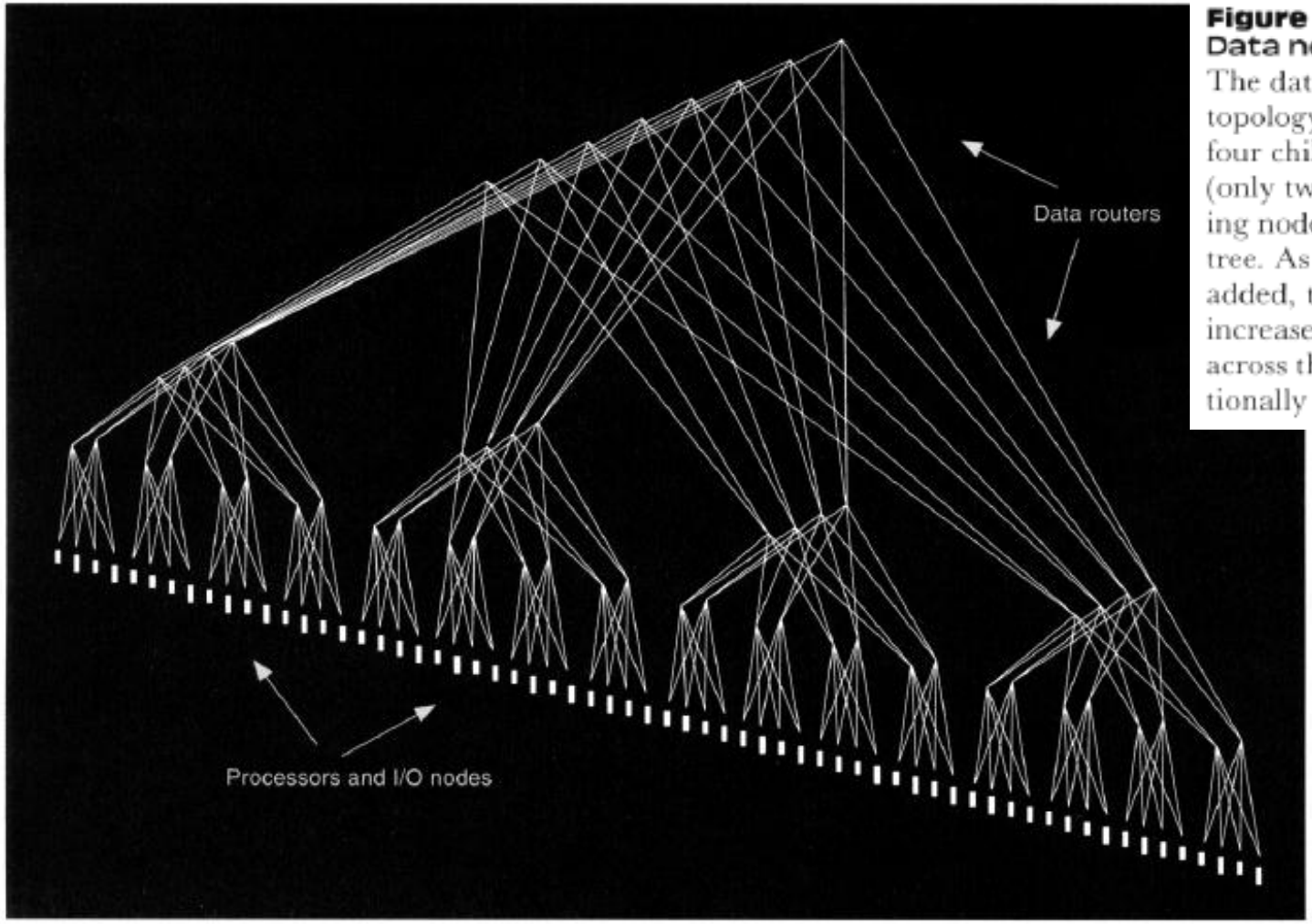
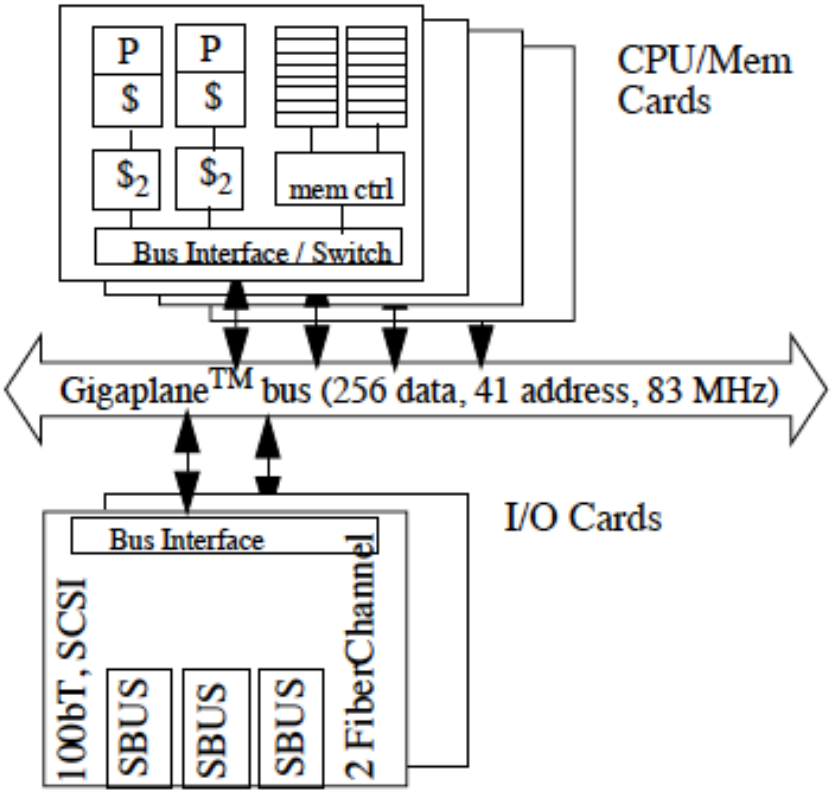
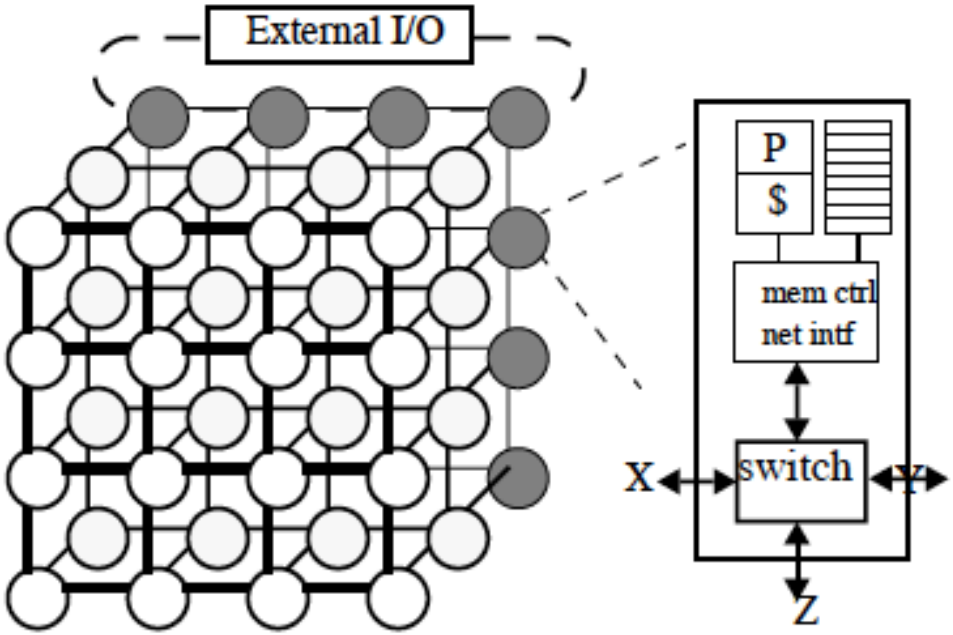


Figure 4.
Data network topology
The data network uses a fat tree topology. Each interior node connects four children to two or more parents (only two parents are shown). Processing nodes form the leaf nodes of the tree. As more processing nodes are added, the number of levels of the tree increases, allowing the total bandwidth across the network to increase proportionally to the number of processors.

Example NUMA Machines (II)

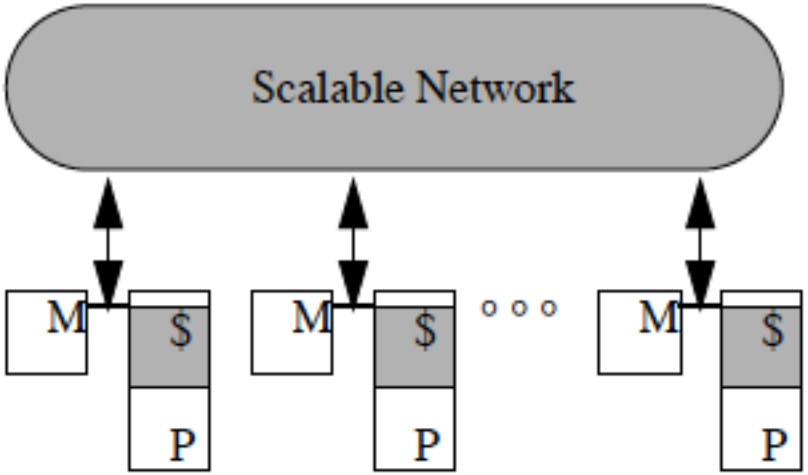
- Sun Enterprise Server
- Cray T3E





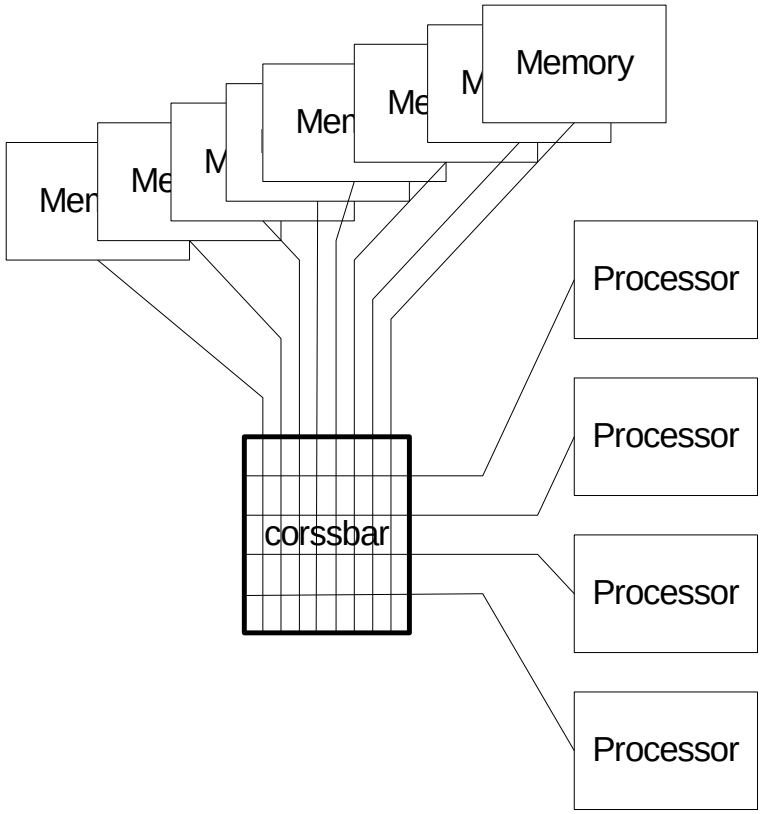
Convergence of Parallel Architectures

- Scalable shared memory architecture is similar to scalable message passing architecture
- Main difference: is remote memory accessible with loads/stores?



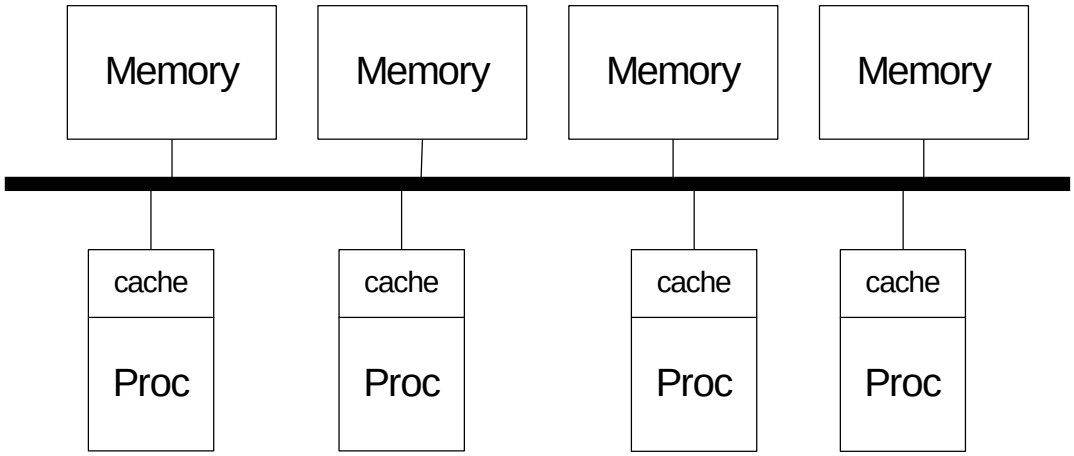
Historical Evolution: 1960s & 70s

- **Early MPs**
  - **Mainframes**
  - **Small number of processors**
  - **crossbar interconnect**
  - **UMA**



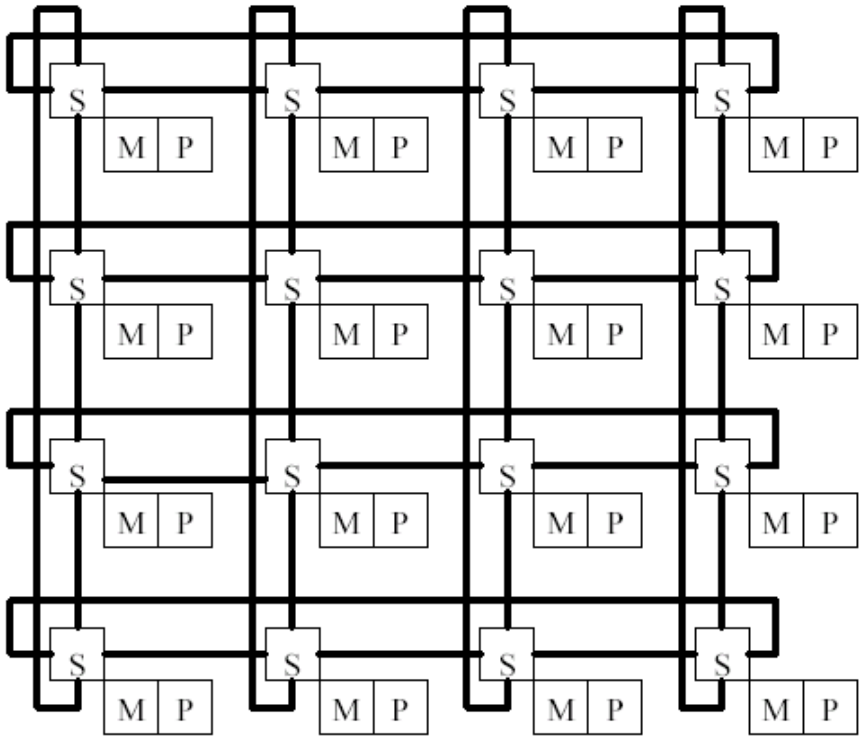
Historical Evolution: 1980s

- **Bus-Based MPs**
  - **enabler: processor-on-a-board**
  - **economical scaling**
  - **precursor of today's SMPs**
  - **UMA**



Historical Evolution: Late 80s, mid 90s

- **Large Scale MPs (Massively Parallel Processors)**
  - **multi-dimensional interconnects**
  - **each node a computer (proc + cache + memory)**
  - **both shared memory and message passing versions**
  - **NUMA**
  - **still used for "supercomputing'**



Historical Evolution: Current

- Chip multiprocessors (multi-core)
- Small to Mid-Scale multi-socket CMPs
  - One module type: processor + caches + memory
- Clusters/Datacenters
  - Use high performance LAN to connect SMP blades, racks

- Driven by economics and cost
  - Smaller systems => higher volumes
  - Off-the-shelf components
- Driven by applications
  - Many more throughput applications (web servers)
  - … than parallel applications (weather prediction)
  - Cloud computing

Historical Evolution: Future

- Cluster/datacenter on a chip?
- Heterogeneous multi-core?
- Bounce back to small-scale multi-core?
- ???

Required Readings

- Hillis and Tucker, "The CM-5 Connection Machine: a scalable supercomputer," CACM 1993.

- Seitz, "The Cosmic Cube," CACM 1985.