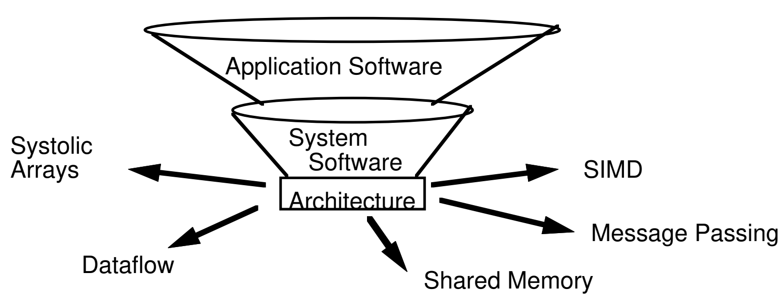
Evolution of Parallel Machines

History

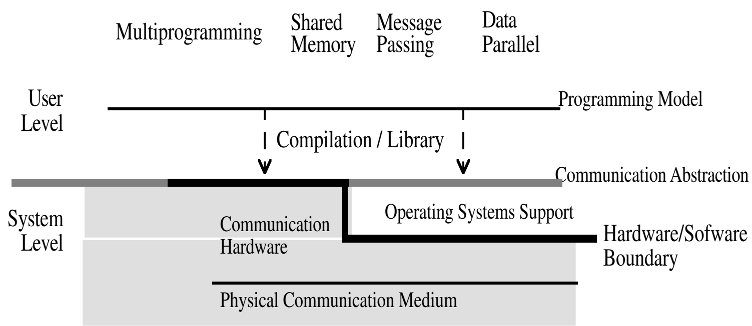
Historically, parallel computing consisted of competing models and divergent architectures, with no predictable pattern of growth.



Uncertainty of direction paralyzed parallel software development!

Today

- Extension of "computer architecture" to support communication and cooperation
  - **OLD: Instruction Set Architecture**
  - **NEW: Communications Architecture**



Communications Architecture

= Communication Absraction + Implementation

Abstraction:

- Comm. primitives exposed to user-level by hardware and system-level software.
- Examples are shared memory model, message-passing model, etc.
- While most models are universal, the implicit assumption is that the primitive operations are efficiently implemented.

Implementation:

- Organizational structures that implement the primitives: hardware or OS
- How optimized are they? How well integrated into the processing node?
- Structure of network

- *Goals*:
  - Performance
  - Broad applicability
  - Programmability
  - Scalability
  - Low Cost

Scalability Considerations

- Both small-scale and large-scale machines have their place.
  - today almost everything beyond desktop is multiprocessor
  - tomorrow desktops may be parallel, servers larger

- Cost-performance-complexity issues, however, can be different depending on the scale.

- Volume market is small-scale, but large-scale often drives advances

Design Issues

- **Naming:** How are logically shared data and/or comm. partner named?

- **Latency:** What is the latency of communicating in a protected fashion?

- **Bandwidth:** How much data can be communicated per second?

- **Synchronization:** How can producers and consumers of data synchronize?

- **Node Granularity:** How to split silicon between processors and memory?

- **Applicability:** How general-purpose or special-purpose?

- ...

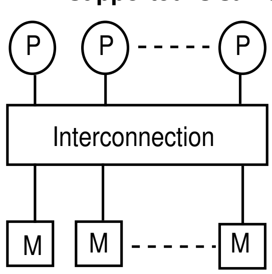
Architectural Models

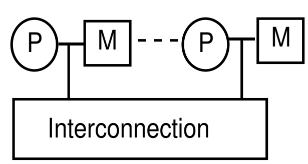
- Shared Address Space
- Message Passing
- Data Parallel (SIMD)
- Dataflow
- Systolic Arrays

Shared Address Space Architectures

•Key Feature: All processors in the system can directly reference all memory locations in the system, thus providing a convenient and fast (?) mechanism for processors to communicate. Bulletin board model

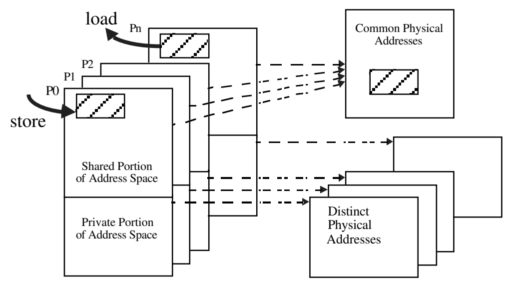
- **convenient: (i) location transparency; (ii) data access abstraction supported is same as that on uniprocessors.**





- Memory either centralized (small-scale) or distributed (large-scale).
- Popularly known as *shared memory* machines (ambiguous name).

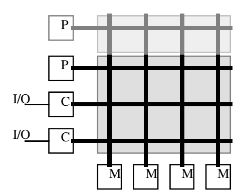
Shared Address Space



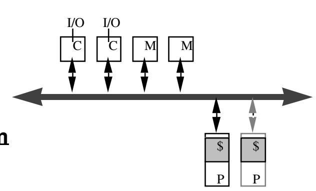
- Communication implicit through load/store on shared variables
- Nice programming model (uniprocessor model + synchronization)
- Potential disadvantage is scalability and hardware cost

History

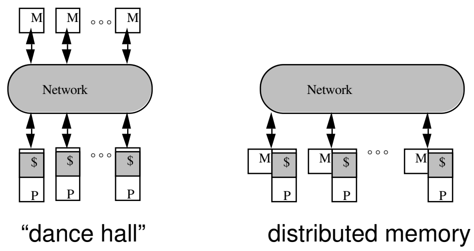
- "Mainframe" approach
  - motivated by multiprogramming
  - extends crossbar
  - latency
  - bandwidth scales with p
  - high incremental cost



- "Minicomputer" approach
  - motivated by multiprogramming and transaction processing
  - used heavily for parallel comp.
  - latency larger than for uniprocessor
  - bus is bandwidth bottleneck
    - » **<u>caching is key: *coherence* problem</u>**
  - low incremental cost



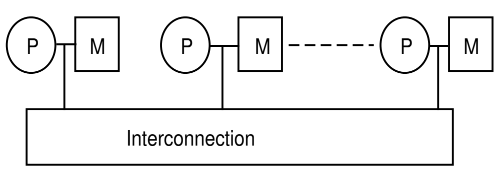
Scalability



- Dance-hall: network still scalable, but lower cost than crossbar
  - **latencies to memory uniform, but uniformly large**
- Distributed memory or non-uniform memory access (NUMA)
  - **observation: data can be laid out so most references are local for many applications**
  - Construct shared address space out of simple message transactions across a general-purpose network
    - read-request, read-response
- Caching shared (particularly nonlocal) data?

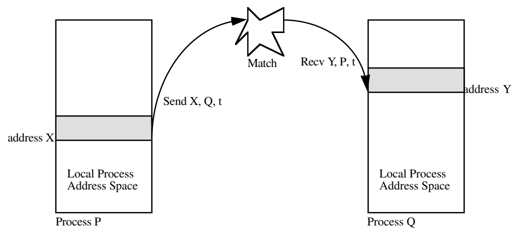
Message Passing Architectures

- Processors can directly access only local memory, and all communication and synchronization is through explicit messages (send, receive). No shared bulletin board.



- Message-passing support typically in I/O system
- Easiest way to build multiprocessor hardware out of uniprocessors
- Key advantage is easy *hardware* scalability

Message-Passing Abstraction



- Send and matching Recv provide a memory-memory copy plus a pairwise synchronization event
- Highly structured programs (SPMD) are common
- Substantial layer of software between user primitives and hardware
  - protection, buffer management
- Sending a message often incurs many overheads:
  - building a header; copying data into network buffers; sending data; receiving data into buffers; copying data from kernel to user space.

Evolution of Message-Passing Machines

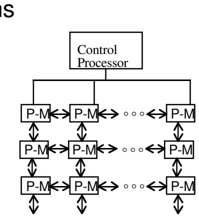
- Early hypercubes provide FIFO on each link
  - HW close to Prog. Model
  - Soon replaced by DMA transfers and non-blocking operation
- **Diminishing role of topology**
  - earlier: store&forward in the comm. software, in the routers
  - now: general purpose (cut-through) routing
- Constraint of only local addresses is artifact of prog. model
- User-level networks (CM-5)
- Complete message processor (Paragon, Meiko CS-2)
- User-to-user DMA

Trend toward Architectural Convergence

- Shared address space machines with distributed memory can be viewed as message passing at lowest level
  – reading a remote node is sending a message to that node and receving the data in another message
- **Important to understand difference between communication abstraction and underlying hardware**
  – shared address space can be emulated on “message-passing machines” and vice versa
  – difference is in efficiency with which abstraction is supported
  – shared address space machnes specialized to handle read/write operations and communicate at cache-line granularity efficiently in hardware
- Basic efficiency needs from network interface are similar
  – low-latency, high-throughput data movement
- So, can support both abstractions on same hardware (Alewife, FLASH)

Data Parallel Architectures

- Programming model
  – **operations are performed on each element of a large (regular) data structure in a single step**
  – arithmetic operations, global data transfer
- 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 O.D.E solvers**
  – **high cost of instruction fetch/sequencing**



Application of Data Parallelism

- Example:

  Each PE contains an employee record with his/her salary

  Code:
```
If salary > 100K then
        salary = salary *1.05
    else
        salary = salary *1.10
```
  Logically, the whole operation is a single step

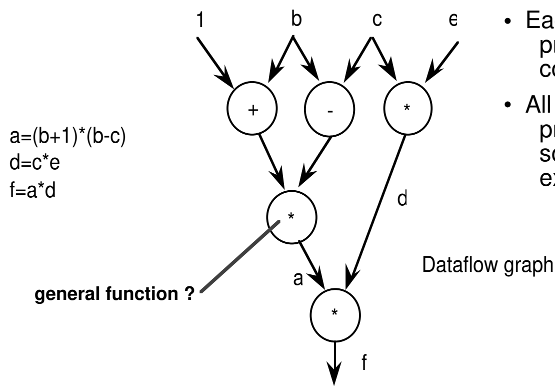
  Notion of **enabled processors.**
- Other examples:
  - Finite differences, linear algebra, ...
  - Document searching, graphics, image processing, ...
- Recent Machines:
  - Thinking Machines CM-1, CM-2 (and CM-5)
  - Maspar MP-1 and MP-2,
  - Wavetracer

Evolution and Convergence

- Rigid control structure (SIMD in Flynn's taxonomy)
  - SISD = uniprocessor, MIMD = multiprocessor
- Cost savings in centralized instruction sequencer
  - 60's when CPU was a cabinet of eqipment
  - mid 80's when 32-bit slices of datapath would just fit on a chip
  - **no longer true with modern microprocessors**
- Simple, regular calculations usually have good locality
  - realize on SM or MP machine with decent compiler
  - may still require fast global synchronization
- **Loss of applicability due to hardwiring data parallelism**
  - MIMD machines can be just as effective to exploit data parallelism (E.g., CM-5 from TMC), and are more general
- Programming model converges with SPMD
  - single program multiple data

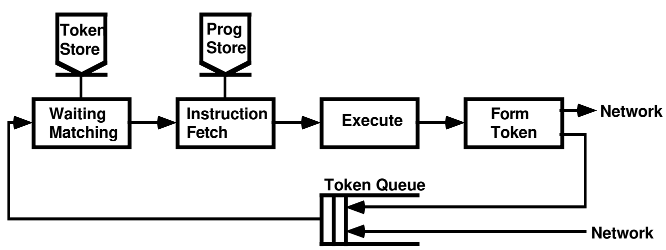
Dataflow Architectures

- Instructions activated by availability of data operands.
- Unlike control flow models, where computation is a series of instructions with implicit or explicit sequencing between them.



- Each node of graph is a process or thread; conceptually a processor
- All dependencies are explicitly present in the dataflow graph, so concurrency is compeletely exposed to hardware.

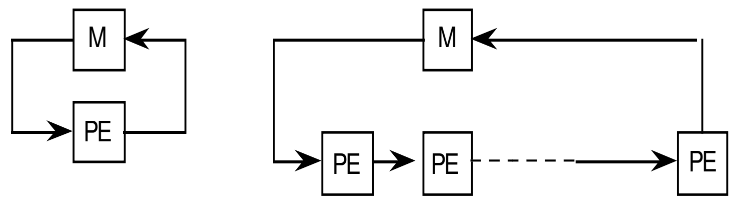
Dataflow (contd.)



- Static dataflow versus dynamic dataflow machines
  - **function (task) invocation**
- Some issues:
  - granularity of operations (locality?)
  - efficient handling of complex data structures like arrays
  - complexity of matching store and memory units
  - problems due to excess parallelism
- Some lasting contributions:
  - **useful concept for software**
  - **fast dispatch on user-level messages**

Systolic Architectures

• *Basic principle*: By replacing a single PE by a regular array of PEs, and by carefully orchestrating the flow of data between the PEs, high throughput can be achieved without increasing memory bandwdith requirements.



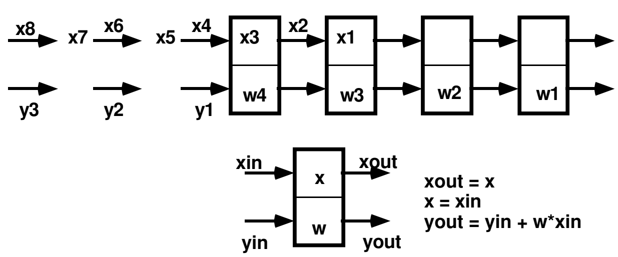
• **Distinguishing features from regular pipelined computers:**

- Array structure can be non linear (e.g, hexagonal)
- Pathways between PEs may be multidirectional
- PEs may have local instruction and data memory, and are more complex than the stage of a pipelined computer.

Systolic Arrays (contd.)

• *Example: Systolic array for 1-D convolution*

y(i) = w1*x(i) + w2*x(i+1) + w3*x(i+2) + w4*x(i+3)

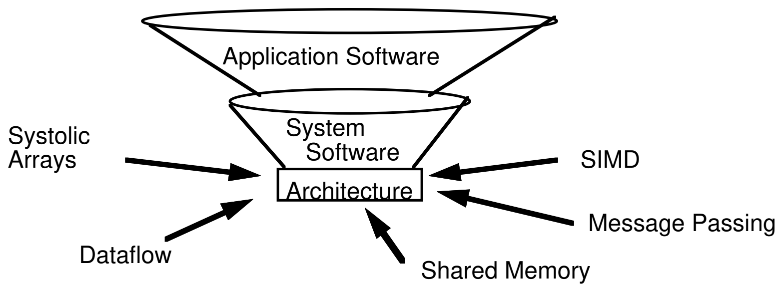


• Issues:

- System integration: Shipping data from host array and back
- Cell architecture and communication architecture (Warp, iWarp)
- Software for automatically mapping computations to systolic arrays
- General purpose?

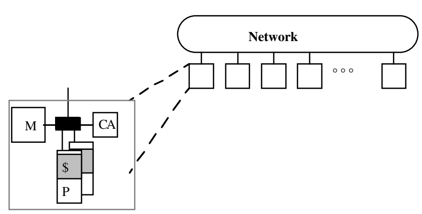
Key contribution to general-purpose: very tight integration of network into node (into processor registers)

Post “Killer Micro” Reality



Convergence

- **A generic modern multiprocessor**



- Node: one or more processors, memory system, plus *communication assist*
  - **network interface and communication controller**
  - **integration with processor, memory system and OS determines fundamental capabilities and efficiency**
- Scalable network

Design Issues Revisited

Fundamental Architectural Issues

- **<u>Naming</u>**: How are communicated data and/or partner referenced?
- **<u>Synchronization</u>**: How can producers and consumers of data coordinate their activities?
- **<u>Latency</u>**: How long does it take to communicate a word of data in a protected fashion?
- **<u>Bandwidth</u>**: How much data can be communicated per second? How many operations per second?

- **<u>Node Granularity</u>:** How should silicon be split between processors and memory?
- **...**

Naming

• Single Global Linear-Address-Space (shared address space)
vs.
• Single Global Segmented-Name-Space (global objects)
vs.
• Multiple Local Address/Name Spaces (message passing)

Naming strategy has implications for:

- Programmer / Software
- Performance
- Design Complexity

Synchronization

• Coordination mainly takes three forms:
  • **multual exclusion** (e.g., spin-locks)

```
lock(A_lock)
A += myA;
unlock(A_lock)
```

  • **event notification**
    • **point-to-point** (e.g., producer-consumer interactions)

```
P1                    P2
a = 5;
flag=1;                   ...
                      while (!flag)
                      read a;
```

    • **global** / **point-to-multipoint** (e.g., end of phase indication)

```
P1                    P2
do phase 1            do phase 1
BARRIER()             BARRIER()
do phase 2            do phase 2
```

Issues in Synchronization

- Synchronization name space (all address space or small portion)
- Granularity (per byte, per word, ... ==> overhead)
- Efficiency affects task granularity that can be sustained
  - Small with hardware support (e.g. HEP, TERA, dataflow)
  - Usually medium grain with software support (queues etc)
- Low latency, low serialization (hot spot handling)
- What primitives?
  - many proposals: test-&-set, compare-&-swap, ldLocked-stCond, F/E bits and traps, queue-based locks, fetch&op with combining, scans, fuzzy barriers...
- **Hardware versus software**

Latency

- Ability to deal with latency is critical with fast processors
  - even the lowest-latency machines have latencies in the hundreds of processor cycles
- Options for dealing with latency:

  1. Lower frequency of long latency events
    » **• algorithmic changes, computation and data distribution, ...**

  2. Reduce latency
    » **caching of shared data, network-interface design, network, ...**

  3. Tolerate latency
    » **message-passing: overlap communication and computation (under programmer control)**
    » **shared address space: overlap access completion and computation using consistency model, prefetching, multithreading or block transfer**

Bandwidth

- **MP programs have private and communication bandwidth requirements**
- **Private bandwith** requirements can be supported using:
  - distributing main memory among PEs
  - application changes, local caches, mem system design, ...
- **Communication bandwidth:**
  - **Node-to-network:**
    - efficient network interfaces and comm controllers
    - applicaton changes to reduce comm and avoid hot spots
  - **Within the network (bisection bandwidth):**
    - scalable interconnects
    - topology matches
    - application changes

Communication Cost

$$\text{Cost} = \text{Frequency} \times \left(\text{Overhead} + \text{Latency} + \frac{\text{Transfer Size}}{\text{Bandwidth}} - \text{Overlap} \right)$$

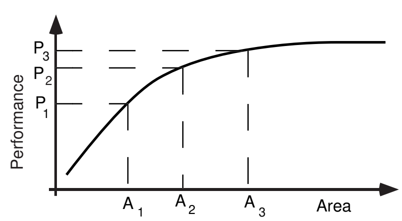
- Frequency - number of communications per unit work
  - **algorithm, placement, replication, bulk**
- Overhead - processor cycles spent initiating or handling
  - **protection check, status, buffer mgmt, copies, events**
- Latency - time to move abit from source to destination
  - **comm. assist, topology, routing, congestion**
- Transfer Time - time through bottleneck
  - **comm. assist, links, congestion**
- Overlap - portion overlapped with useful work
  - **comm. assist, comm. operations, proc. design**

Granularity of PEs

- **How much hardware (e.g., silicon area) should be devoted to each node?**
  - Coarse Grain: E.g., Tens of SGI 4D/340s on an Ethernet / FDDI.
  - Medium Grain: E.g., intel Touchstone (Hundreds of i860 XPs each with 32 Mbytes of memory connected to a wormhole routed mesh.)
  - Fine Grain: E.g., MIT J-Machine (Thousands of MDPs each with 1 Mbyte of memory connected to a wormhole routed mesh.)

- *Some considerations*:
  - **Node performance as a function of node area**
    - processor power • memory/node • communication cost
  - **Application characteristics**
    - concurrency • comm-to-comp ratio • working set size
  - **Practical considerations**
    - available processors; available DRAM / SRAM parts; ...

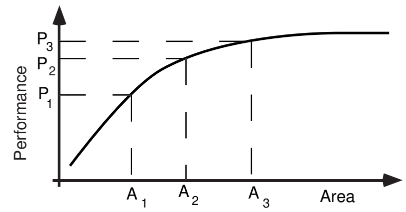
Granularity Tradeoffs



- **In the beginning (upto A1)**, as A grows P grows fairly predictably
  - # of bits operated upon in parallel (bit serial --> 32 --> 64 bits)
  - pipelined instruction execution
  - special hardware units (FPU; MMU; ...)

  ==> No controversy

Granularity Tradeoffs (contd.)



- **Later in the curve, relationship is more complex and application dependent**
  - superscalar / VLIW processors
  - use of caches? what sized caches? (programming ease)
  - technology-power-area-speed tradeoffs
  - amount of main memory per node? application working set size?
  - concurrency in application? (e.g., sequential appln. ==> A3 is best)
  - ...
- **For the time being and in small-scale, coarse-grained nodes advantageous**

==> Longer term, depending on design goals (applns of interest, scale...) and technology, any of coarse, medium, or fine grain may make sense.

Granularity of PEs

- **Very powerful nodes (e.g., Cray-YMP):**
  - \+ broad applicability (e.g., applns with limited parallelism)
  - \- cost-performance not so good
- **Latest microprocessor-based nodes (e.g., DEC alpha):**
  - \+ broad applicability still holds
  - \+ competetive cost-performance even for large scale machines (ride the performance curve for regular workstations)
- **Weaker nodes (e.g., little memory per PE, but more PEs):**
  - \+ where applicable, better cost-performance
  - \- Amdahl's law can come to haunt for applications with limited parallelism

Historical Perspectiive

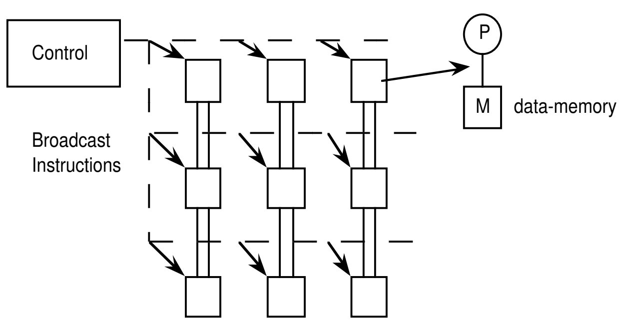
Flynn's Classification

- Based on notions of:
  - **instruction streams**
  - **data streams**

- Characterized by hardware provided to service I and D streams:
  - **SISD**: Single Instruction and Single Data stream (SparcStation)
  - **SIMD**: Single Instruction and Multiple Data streams (CM-2)
  - **MISD**: Multiple Instruction and Single Data streams (CMU Warp)
  - **MIMD**: Multiple Instruction and Multiple Data streams (Sequent)

SIMD Machines

- **Examples:** Illiac-IV, CM-2, Maspar MP-1.

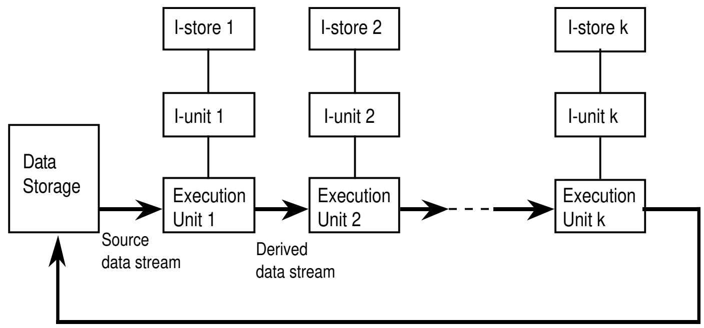


- **Advantages:** Simplicity of control; cheaper; easier to debug programs.

- **Disadvantages:** Restricted model; resource wastage if few PEs are enabled.

MISD Machines

• **Examples:** Systolic arrays such as the CMU Warp machine.

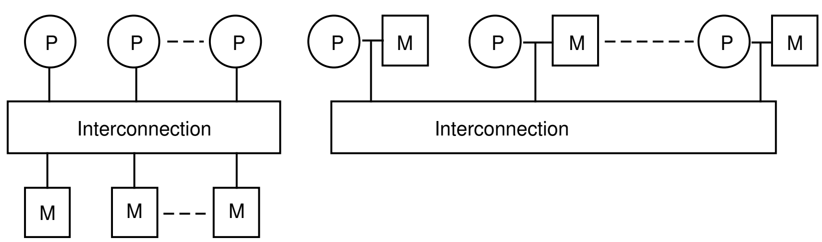


• **Advantages:** Simple to design; great cost-performance when applicable (e.g., for signal processing and image processing applications).

• **Disadvantages:** Limited applicability; can be difficult to program.

MIMD Machines

• **Examples:** Sequent Symmetry, SGI 4D/340, SUN Galaxy, CM-5, NCUBE, ...



Global Memory Scheme

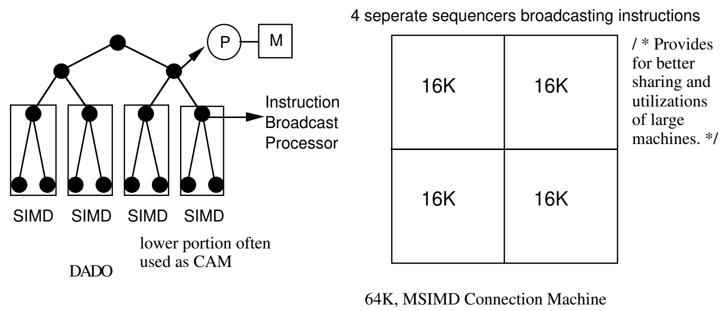
Distributed Memory Scheme

• **Advantages:** Wide applicability, leverage commodity processors.

• **Disadvantages:** Difficult to design well; synch overheads can be large; programming can be difficult.

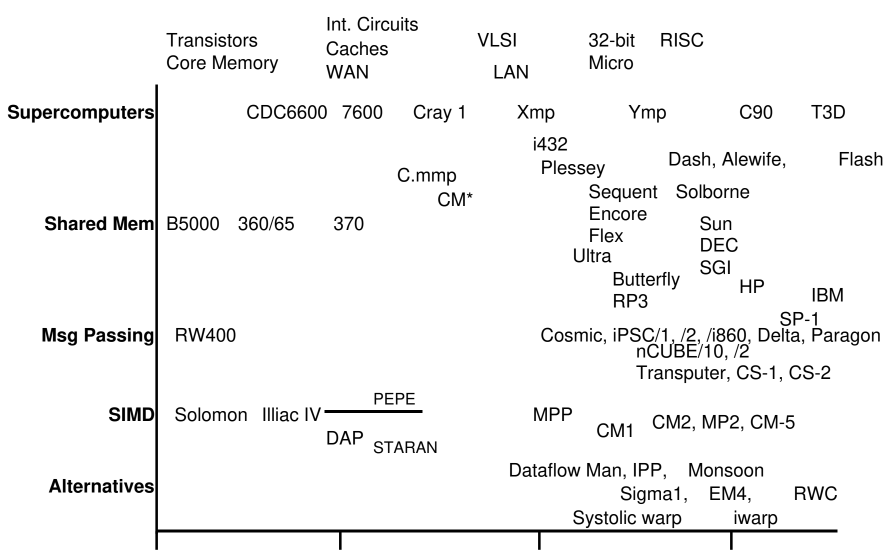
MSIMD Machines

• **Examples:** Columbia's DADO machine; CM-2.



• **Advantages:** Allows more flexibility (better resource sharing) than pure SIMD machines.

Historical Perspective



Summary

- **Many architectures and programming models.**

- **For highly parallel machines many architectural aspects are converging:**
  - powerful microprocessors, distributed main memories, general principles for network design.

- **Open questions remain in the areas of:**
  - Granularity of processing nodes
  - Naming model
  - Latency reducing and tolerating mechanisms
  - Scalable bandwidth in the presence of hot spots
  - Synchronization mechanisms
  - What mechanisms should be supported in hardware and what should be left to software

Technology Summary

- **Microprocessors are the building block of choice**
  - main difficulty is handling very fine-grained tasks and synchronization

- **Moderate numbers of processors are the right scale for this decade (100s to 1000s)**

- **Memory will be physically distributed with processors**

- **Communication with a remote processor or memory will be expensive**