

# Introduction

What is Parallel Architecture?

Why Parallel Architecture?

Evolution and Convergence of Parallel Architectures

Fundamental Design Issues

# What is Parallel Architecture?

A parallel computer is a collection of processing elements that cooperate to solve large problems fast

Some broad issues:

- Resource Allocation:
  - how large a collection?
  - how powerful are the elements?
  - how much memory?
- Data access, Communication and Synchronization
  - how do the elements cooperate and communicate?
  - how are data transmitted between processors?
  - what are the abstractions and primitives for cooperation?
- Performance and Scalability
  - how does it all translate into performance?
  - how does it scale?

# Why Study Parallel Architecture?

Role of a computer architect:

To design and engineer the various levels of a computer system to maximize *performance* and *programmability* within limits of *technology* and *cost*.

Parallelism:

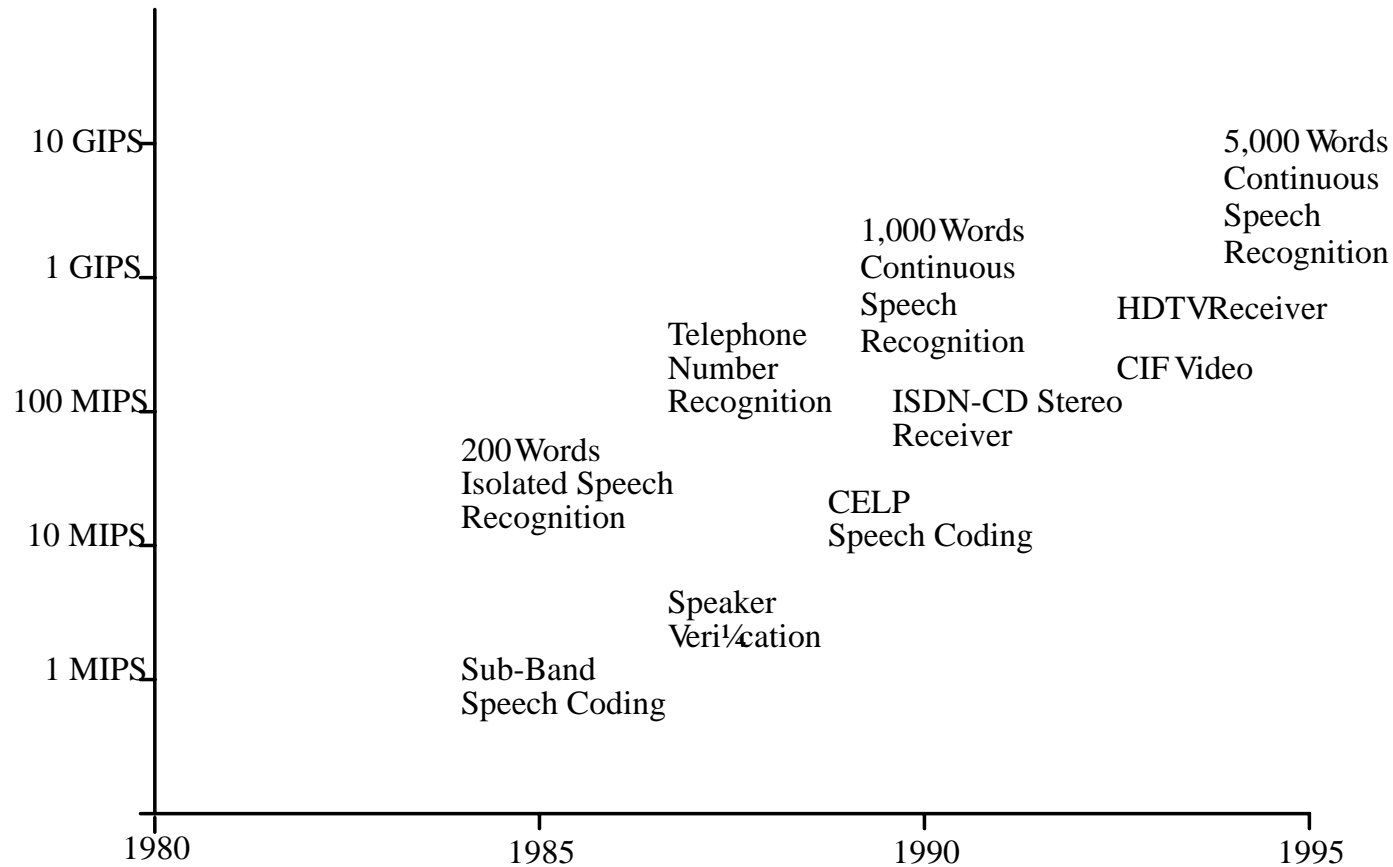
- Provides alternative to faster clock for performance
- Applies at all levels of system design
- Is a fascinating perspective from which to view architecture
- Is increasingly central in information processing

# Engineering Computing Demand

Large parallel machines a mainstay in many industries

- Petroleum (reservoir analysis)
- Automotive (crash simulation, drag analysis, combustion efficiency),
- Aeronautics (airflow analysis, engine efficiency, structural mechanics, electromagnetism),
- Computer-aided design
- Pharmaceuticals (molecular modeling)
- Visualization
  - in all of the above
  - entertainment (films like Toy Story)
  - architecture (walk-throughs and rendering)
- Financial modeling (yield and derivative analysis)
- etc.

# Applications: Speech and Image Processing



- Also CAD, Databases, . . .
- *100 processors gets you 10 years, 1000 gets you 20 !*

# Summary of Application Trends

Transition to parallel computing has occurred for scientific and engineering computing

In rapid progress in commercial computing

- Database and transactions as well as financial
- Usually smaller-scale, but large-scale systems also used

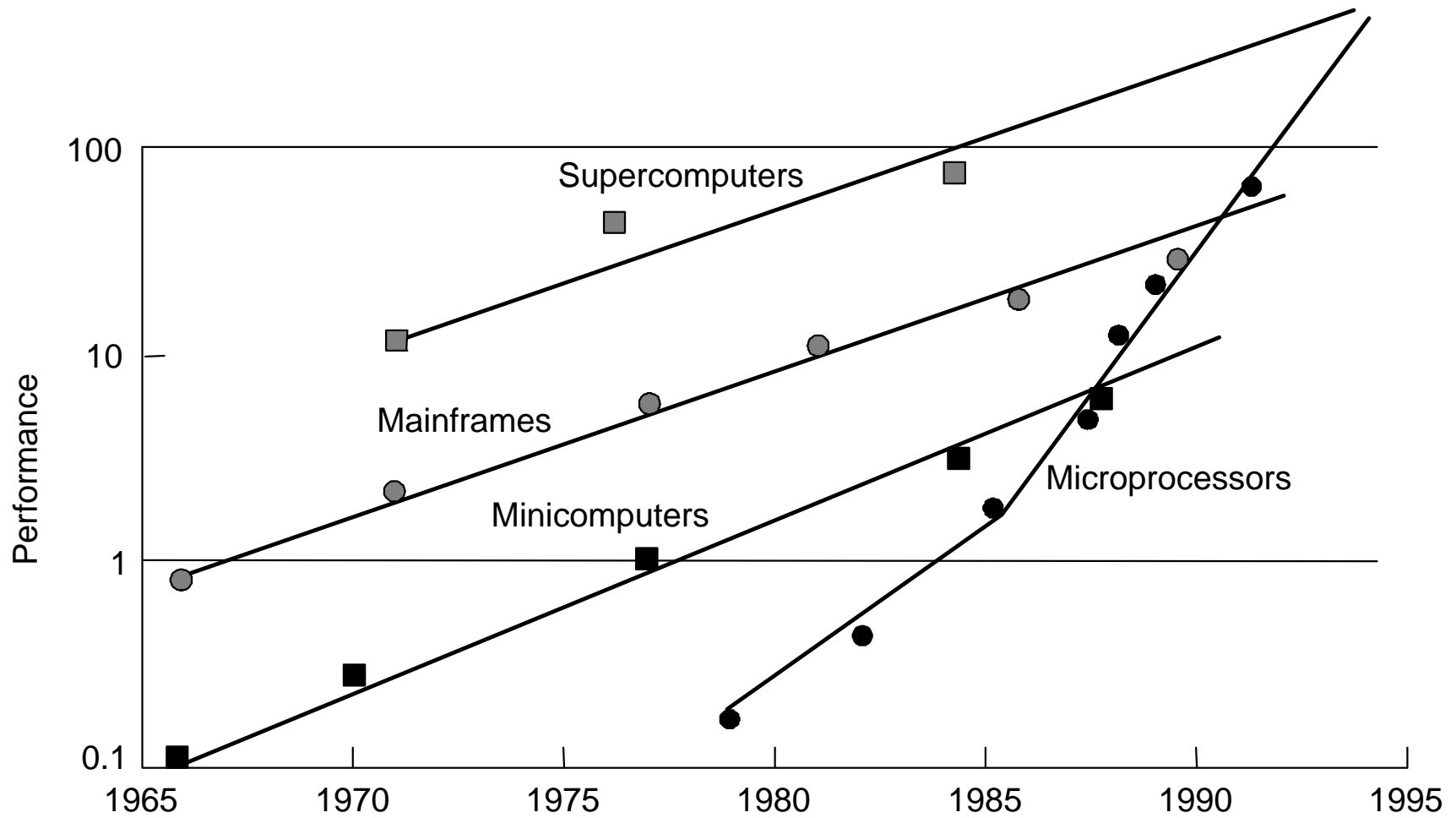
Desktop also uses multithreaded programs, which are a lot like parallel programs

Demand for improving throughput on sequential workloads

- Greatest use of small-scale multiprocessors

Solid application demand exists and will increase

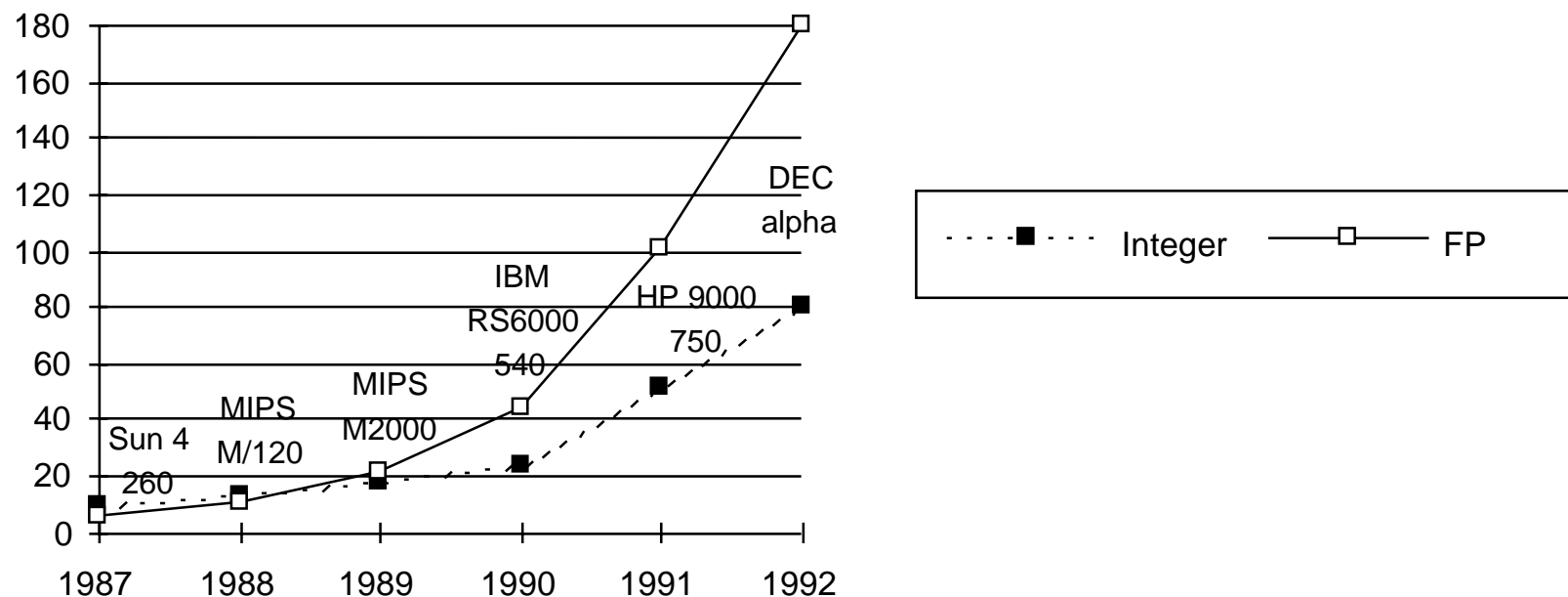
# Technology Trends



The natural building block for multiprocessors is now also about the fastest!

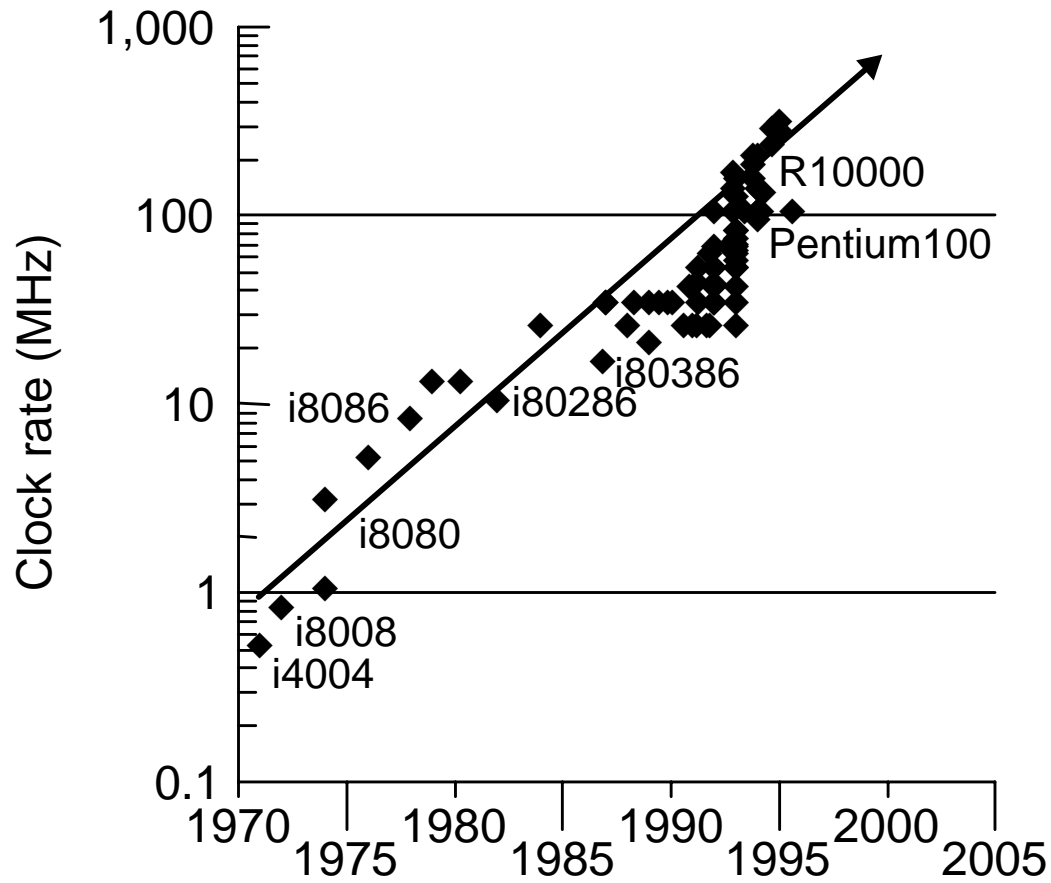
# General Technology Trends

- *Microprocessor performance* increases 50% - 100% per year
- *Transistor count* doubles every 3 years
- *DRAM size* quadruples every 3 years
- Huge investment per generation is carried by huge commodity market



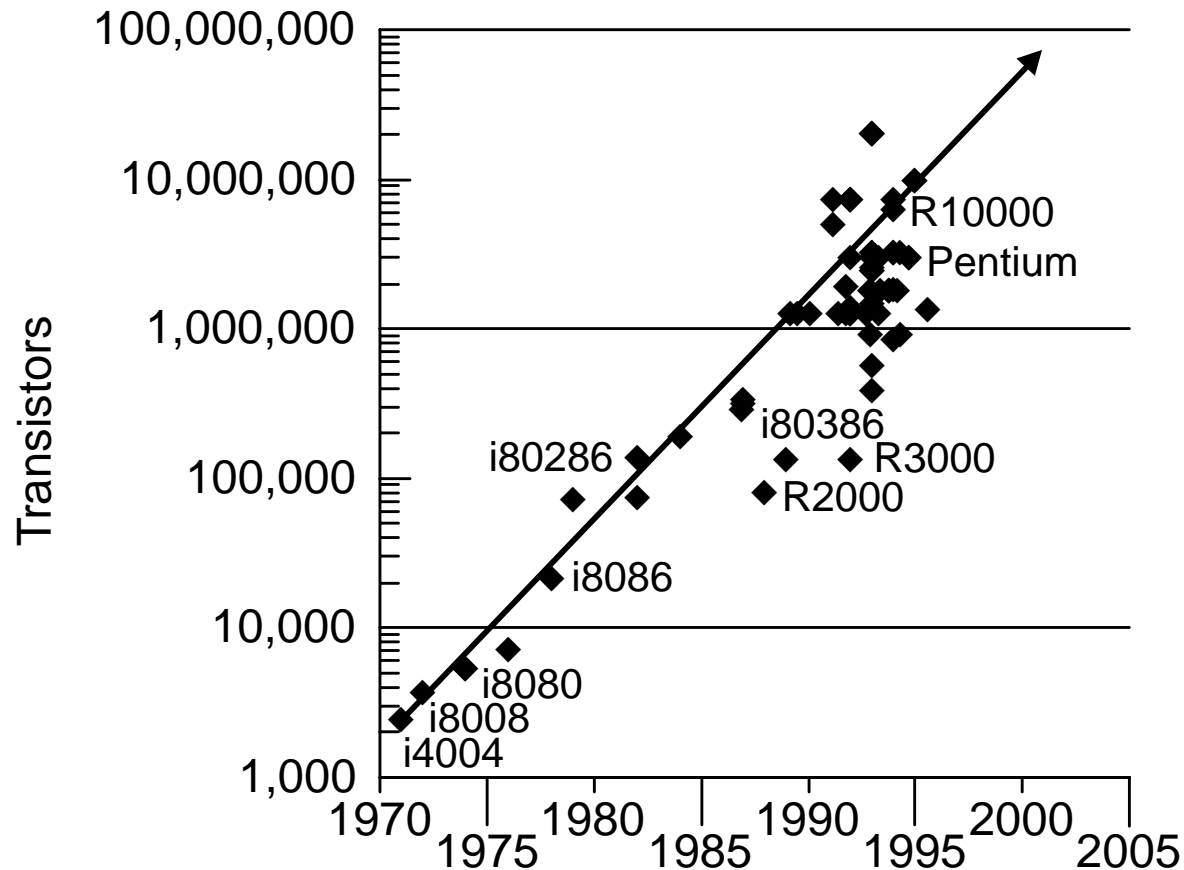
- Not that single-processor performance is plateauing, but that parallelism is a natural way to improve it.

# Clock Frequency Growth Rate



- 30% per year

# Transistor Count Growth Rate



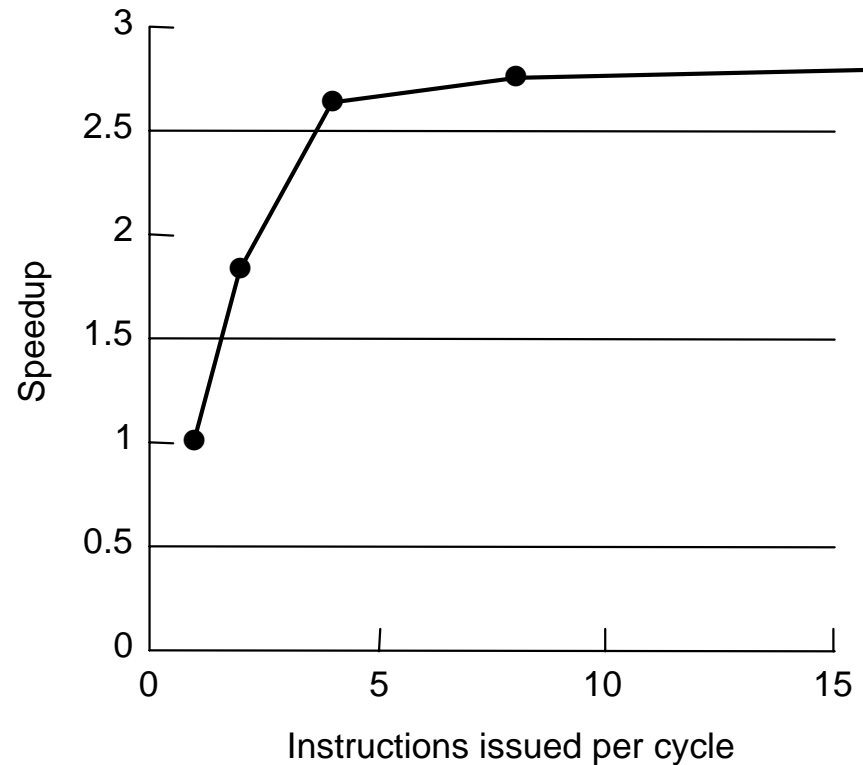
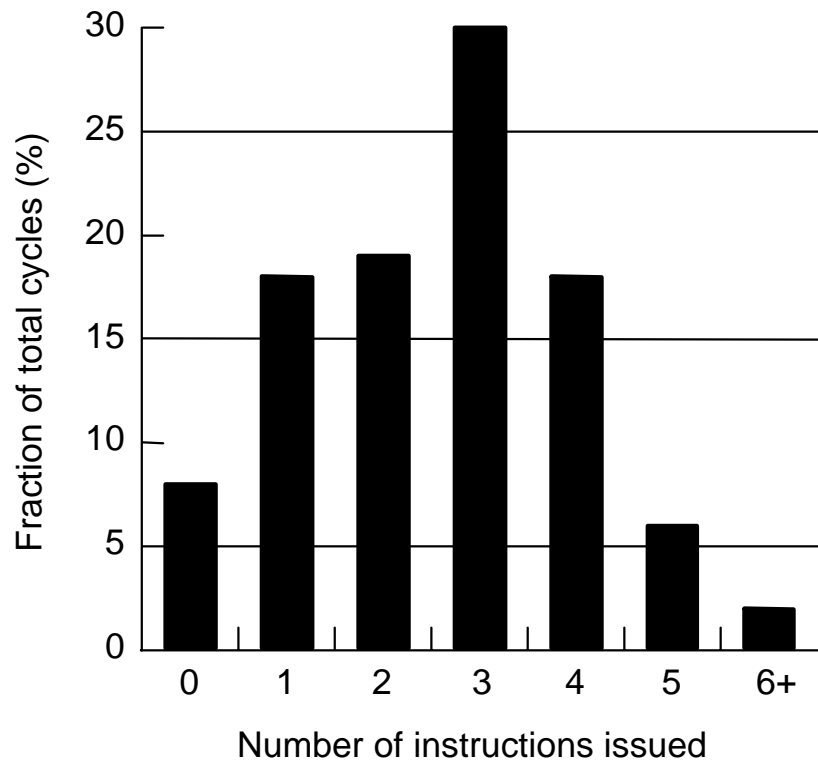
- 100 million transistors on chip by early 2000's A.D.
- Transistor count grows much faster than clock rate
  - 40% per year, order of magnitude more contribution in 2 decades

# Architectural Trends

Greatest trend in VLSI generation is increase in parallelism

- Up to 1985: bit level parallelism: 4-bit -> 8 bit -> 16-bit
  - slows after 32 bit
  - adoption of 64-bit now under way, 128-bit far (not performance issue)
  - great inflection point when 32-bit micro and cache fit on a chip
- Mid 80s to mid 90s: instruction level parallelism
  - pipelining and simple instruction sets, + compiler advances (RISC)
  - on-chip caches and functional units => superscalar execution
  - greater sophistication: out of order execution, speculation, prediction
    - to deal with control transfer and latency problems
- Next step: thread level parallelism

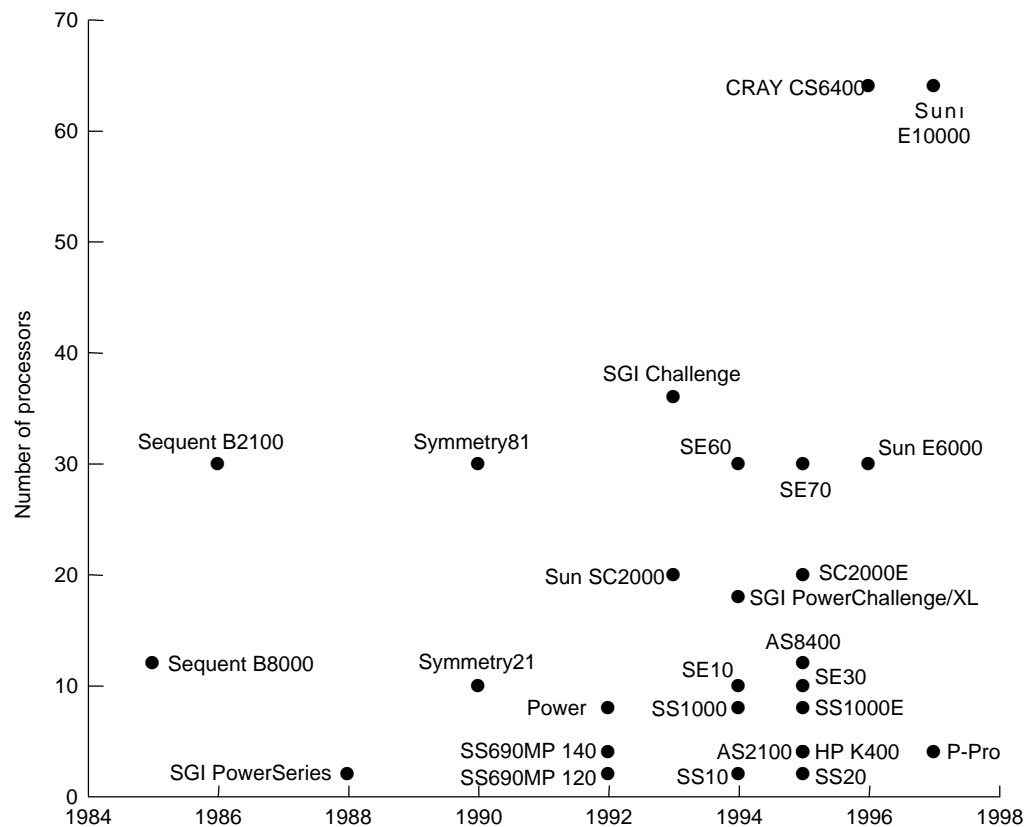
# ILP Ideal Potential



- Infinite resources and fetch bandwidth, perfect branch prediction and renaming
  - real caches and non-zero miss latencies

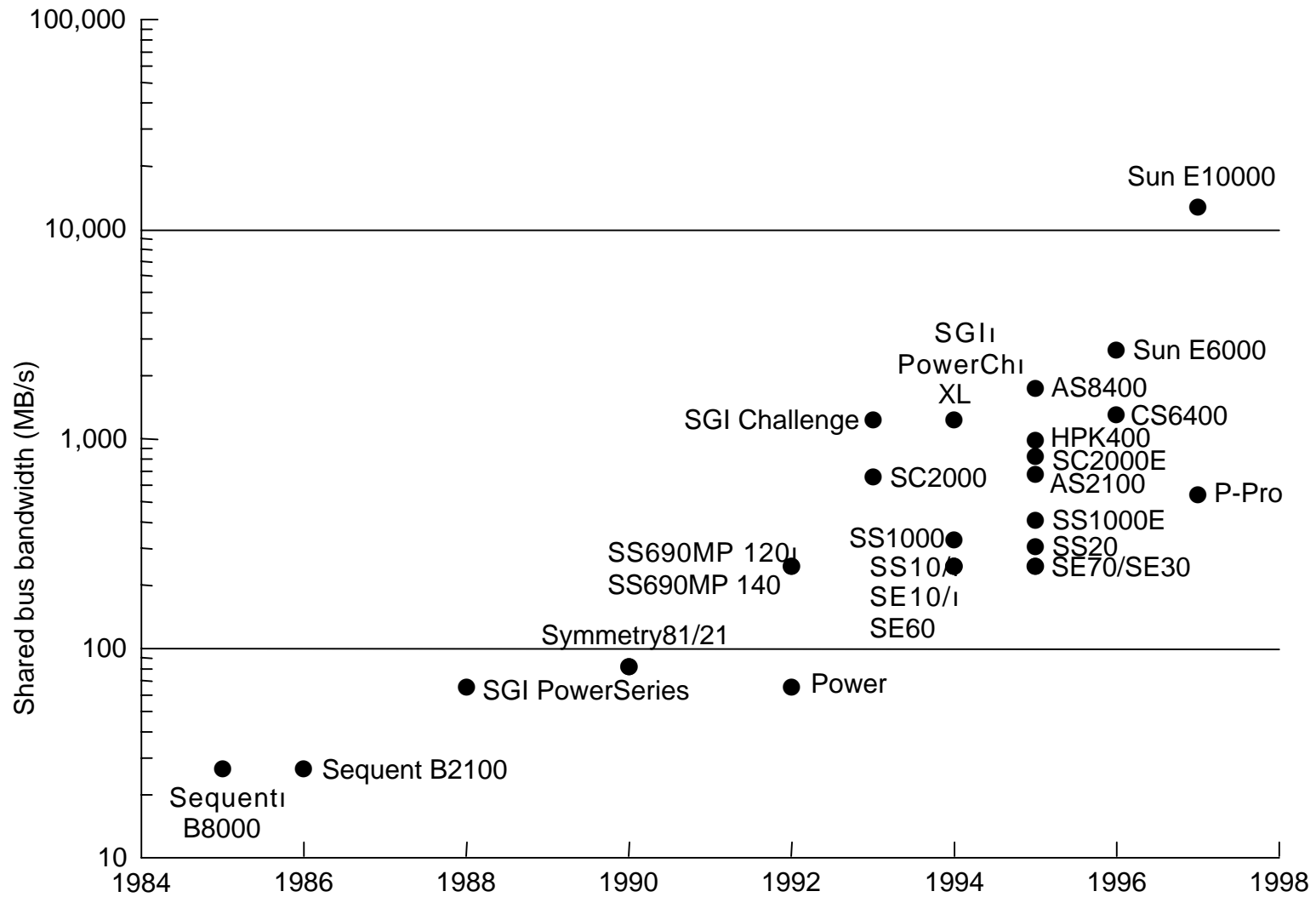
# Architectural Trends: Bus-based MPs

- Micro on a chip makes it natural to connect many to shared memory
  - dominates server and enterprise market, moving down to desktop
- Faster processors began to saturate bus, then bus technology advanced
  - today, range of sizes for bus-based systems, desktop to large servers



No. of processors in fully configured commercial shared-memory systems

# Bus Bandwidth



# Economics

Commodity microprocessors not only fast but CHEAP

- Development cost is tens of millions of dollars (5-100 typical)
- BUT, many more are sold compared to supercomputers
- Crucial to take advantage of the investment, and use the commodity building block
- Exotic parallel architectures no more than special-purpose

Multiprocessors being pushed by software vendors (e.g. database) as well as hardware vendors

Standardization by Intel makes small, bus-based SMPs commodity

Desktop: few smaller processors versus one larger one?

- Multiprocessor on a chip

# Summary: Why Parallel Architecture?

Increasingly attractive

- Economics, technology, architecture, application demand

Increasingly central and mainstream

Parallelism exploited at many levels

- Instruction-level parallelism
- Multiprocessor servers
- Large-scale multiprocessors (“MPPs”)

Focus of this class: multiprocessor level of parallelism

Same story from memory system perspective

- Increase bandwidth, reduce average latency with many local memories

Wide range of parallel architectures make sense

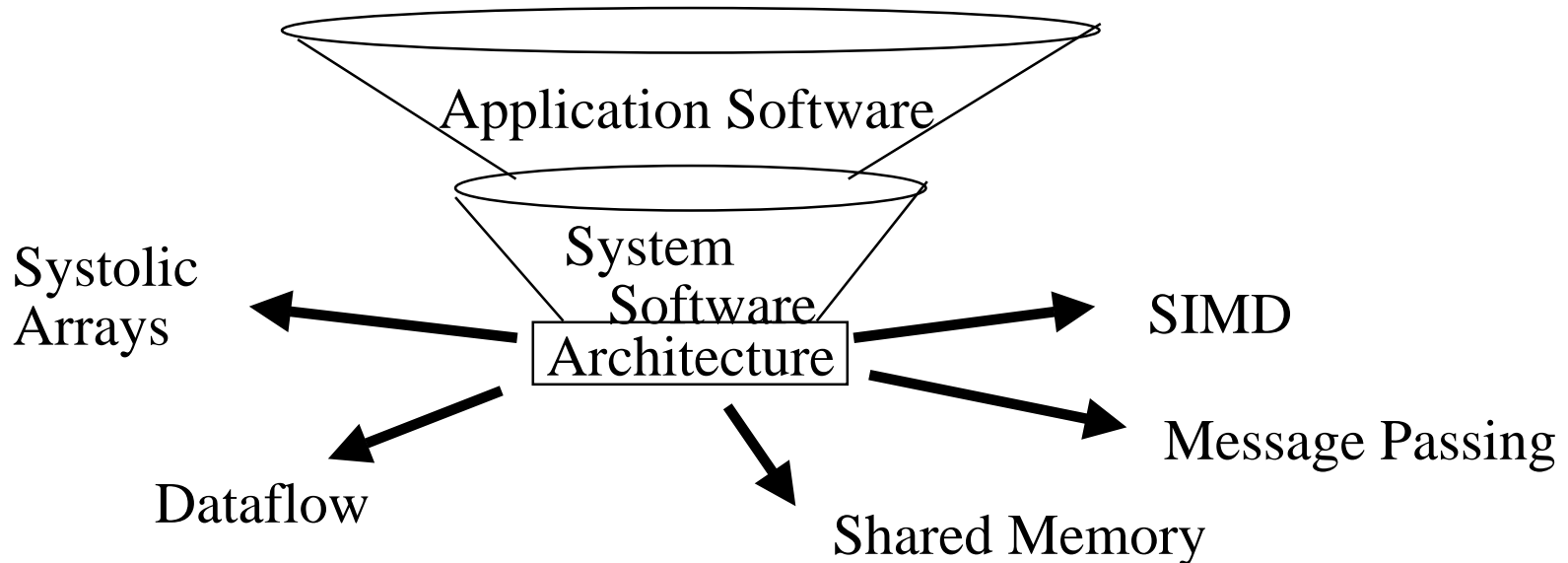
- Different cost, performance and scalability

# Convergence of Parallel Architectures

# History

Historically, parallel architectures tied to programming models

- 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: *Communication Architecture*

Defines

- Critical abstractions, boundaries, and primitives (interfaces)
- Organizational structures that implement interfaces (hw or sw)

Compilers, libraries and OS are important bridges today

# Programming Model

What programmer uses in coding applications

Specifies communication and synchronization

Examples:

- Multiprogramming: no communication or synch. at program level
- *Shared address space*: like 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

# Communication Architecture

= *User/System Interface + Implementation*

## 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

# Shared Address Space Architectures

Any processor can directly reference any memory location

- Communication occurs implicitly as result of loads and stores

Convenient:

- Location transparency
- Similar programming model to time-sharing on uniprocessors
  - Except processes run on different processors
  - Good throughput on multiprogrammed workloads

Naturally provided on wide range of platforms

- History dates at least to precursors of mainframes in early 60s
- Wide range of scale: few to hundreds of processors

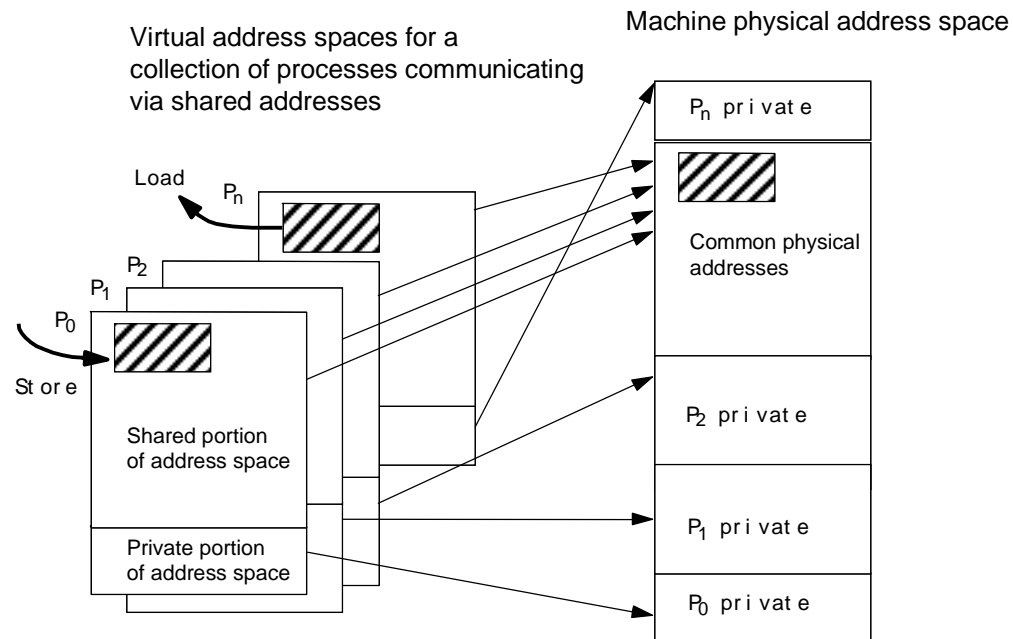
Popularly known as *shared memory* machines or model

- Ambiguous: memory may be physically distributed among processors

# Shared Address Space Model

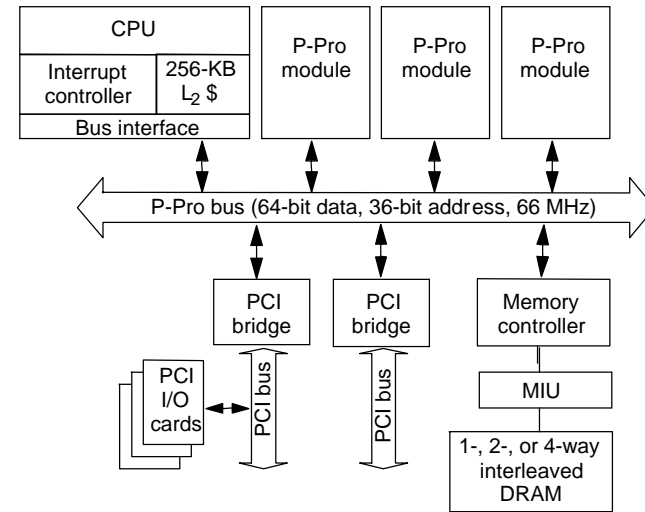
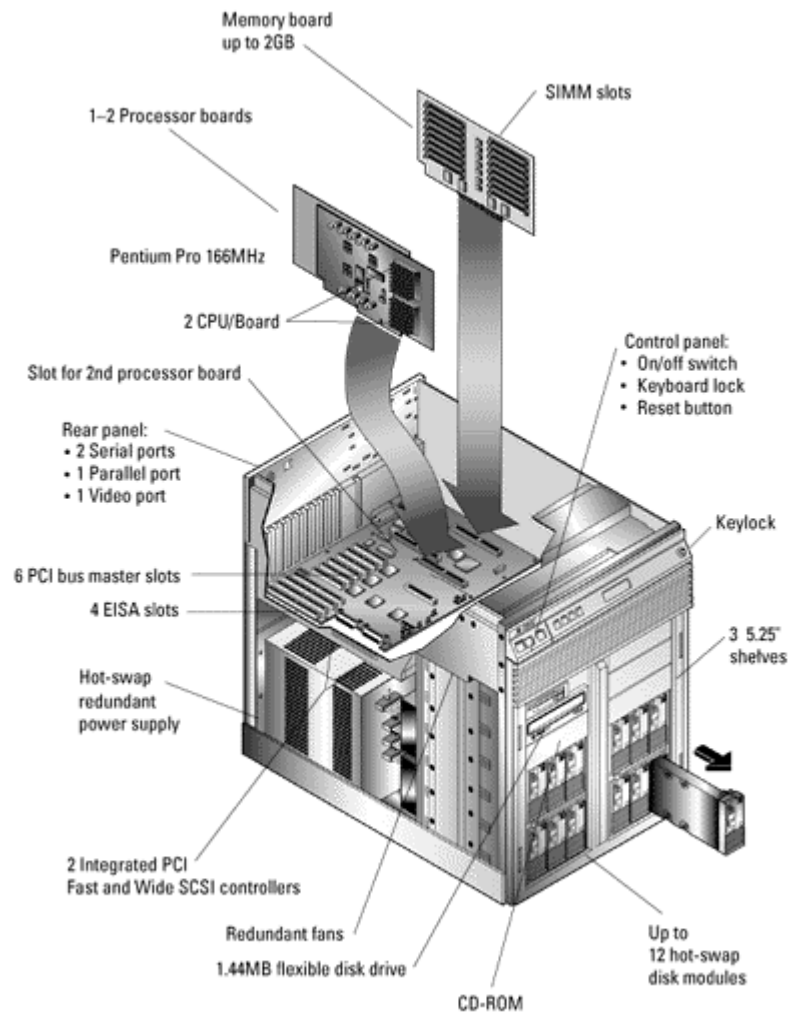
Process: virtual address space plus one or more threads of control

Portions of address spaces of processes are shared



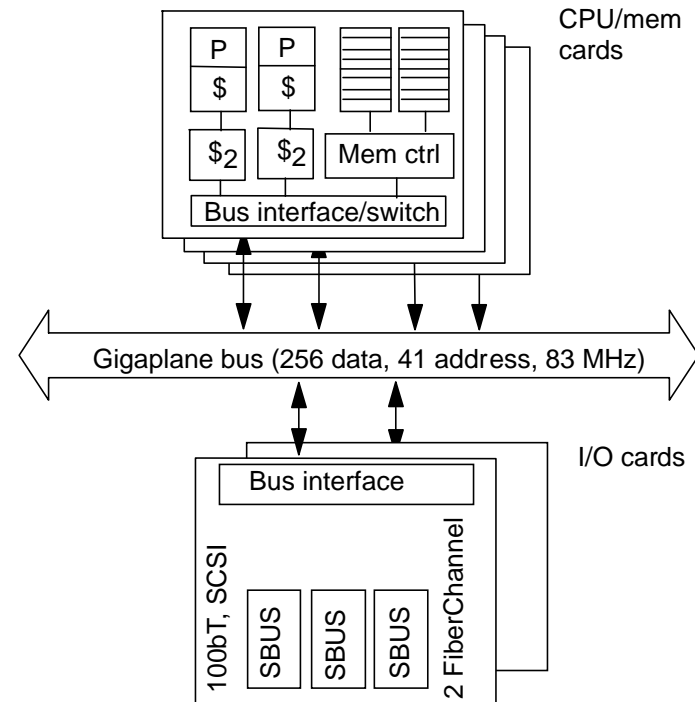
- Writes to shared address visible to other threads (in other processes too)
- Natural extension of uniprocessors model: conventional memory operations for comm.; special atomic operations for synchronization
- OS uses shared memory to coordinate processes

# Example: Intel Pentium Pro Quad



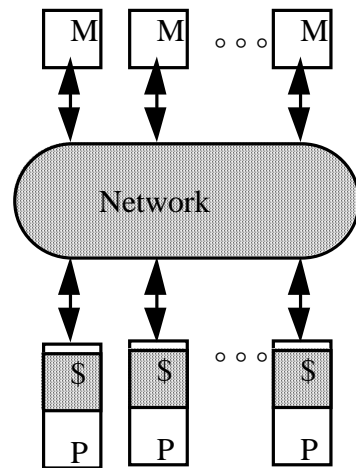
- All coherence and multiprocessing glue in processor module
- Highly integrated, targeted at high volume
- Low latency and bandwidth

# Example: SUN Enterprise

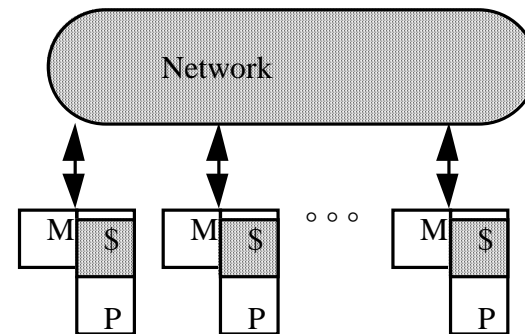


- 16 cards of either type: processors + memory, or I/O
- All memory accessed over bus, so symmetric
- Higher bandwidth, higher latency bus

# Scaling Up



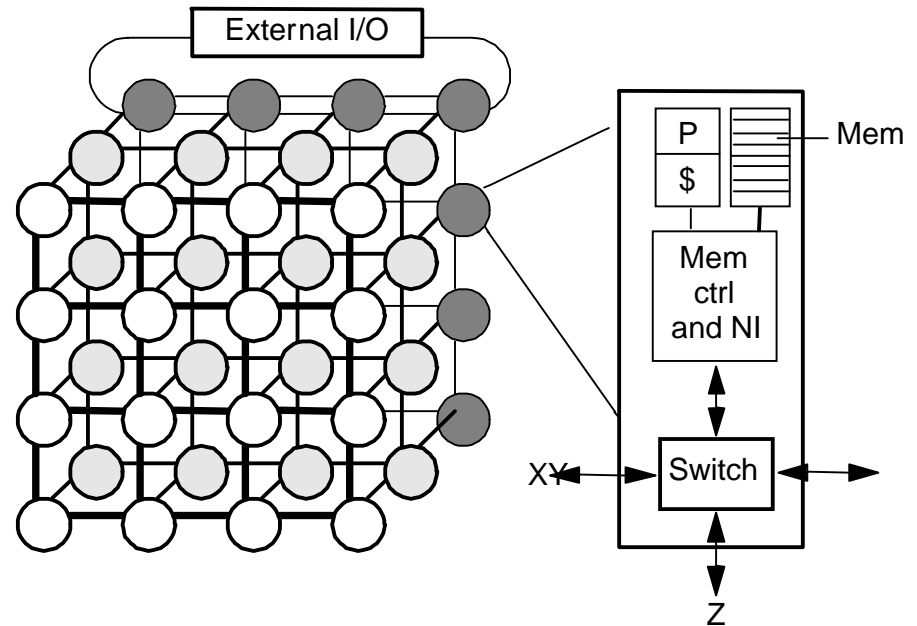
“Dance hall”



Distributed memory

- Problem is interconnect: cost (crossbar) or bandwidth (bus)
- Dance-hall: bandwidth still scalable, but lower cost than crossbar
  - latencies to memory uniform, but uniformly large
- Distributed memory or non-uniform memory access (NUMA)
  - Construct shared address space out of simple message transactions across a general-purpose network (e.g. read-request, read-response)
- Caching shared (particularly nonlocal) data?

# Example: Cray T3E



- Scale up to 1024 processors, 480MB/s links
- Memory controller generates comm. request for nonlocal references
- No hardware mechanism for coherence (SGI Origin etc. provide this)

# Message Passing Architectures

Complete computer as building block, including I/O

- Communication via explicit I/O operations

Programming model: directly access only private address space (local memory), comm. via explicit messages (send/receive)

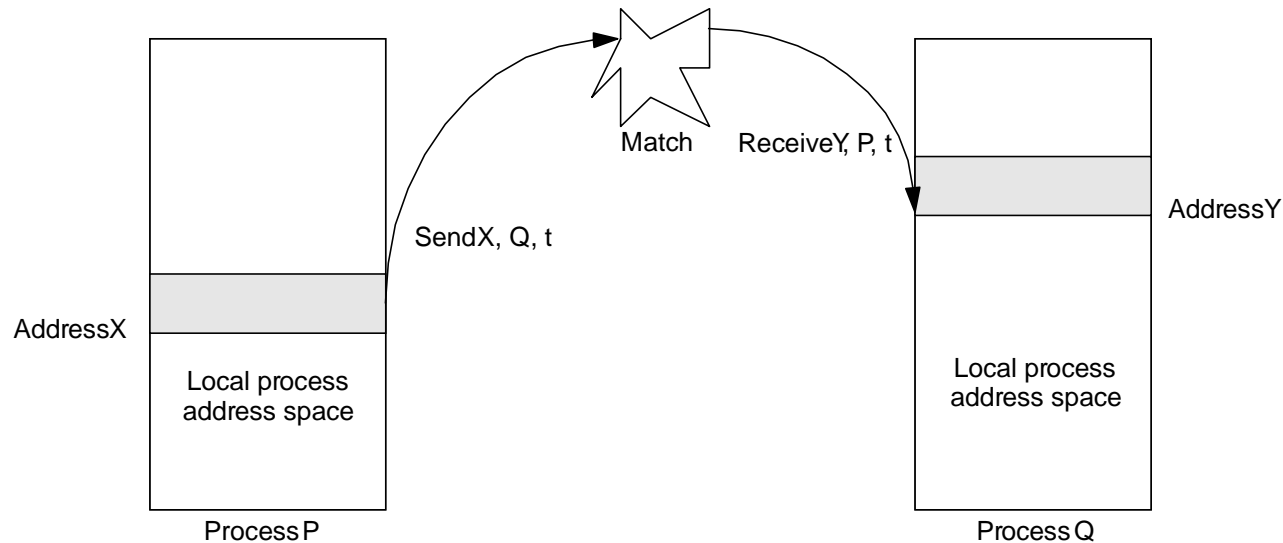
High-level block diagram similar to distributed-memory SAS

- But comm. integrated at IO level, needn't be into memory system
- Like networks of workstations (clusters), but tighter integration
- Easier to build than scalable SAS

Programming model more removed from basic hardware operations

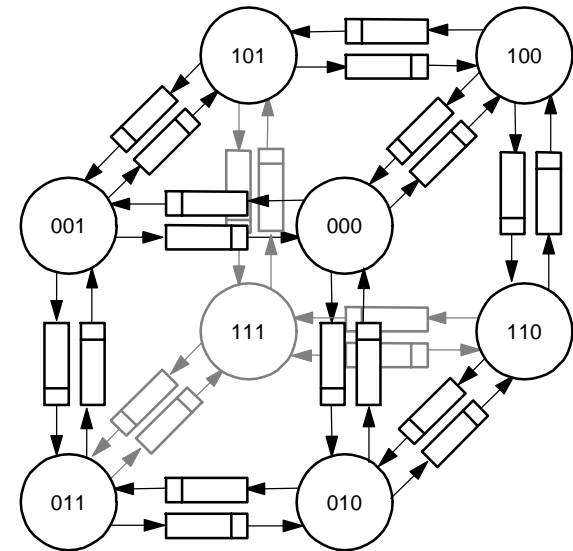
- Library or OS intervention

# Message-Passing Abstraction



- Send specifies buffer to be transmitted and receiving process
- Recv specifies sending process and application storage to receive into
- Memory to memory copy, but need to name processes
- Optional tag on send and matching rule on receive
- User process names local data and entities in process/tag space too
- In simplest form, the send/recv match achieves pairwise synch event
  - Other variants too
- Many overheads: copying, buffer management, protection

# Evolution of Message-Passing Machines



## Early machines: FIFO on each link

- Hw close to prog. Model; synchronous ops
- Replaced by DMA, enabling non-blocking ops
  - Buffered by system at destination until recv

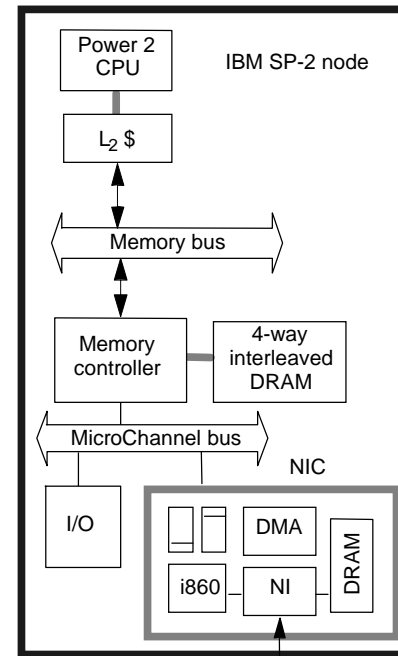
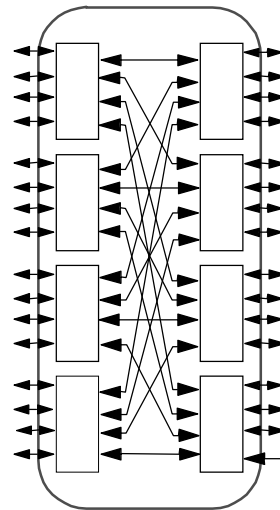
## Diminishing role of topology

- Store&forward routing: topology important
- Introduction of pipelined routing made it less so
- Cost is in node-network interface
- Simplifies programming

# Example: IBM SP-2

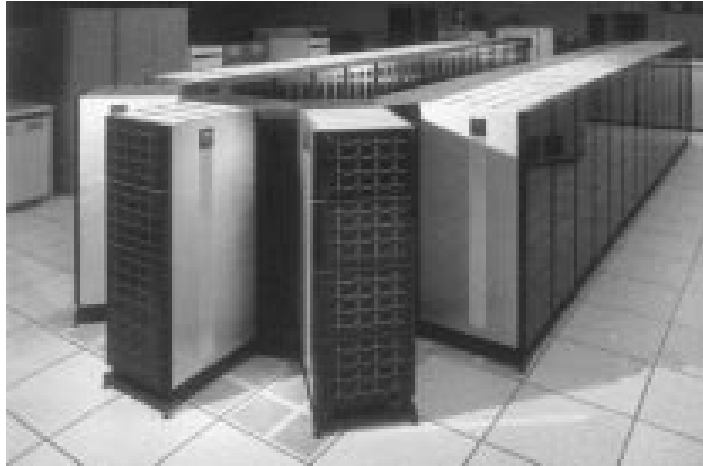


General interconnection network formed from 8-port switches

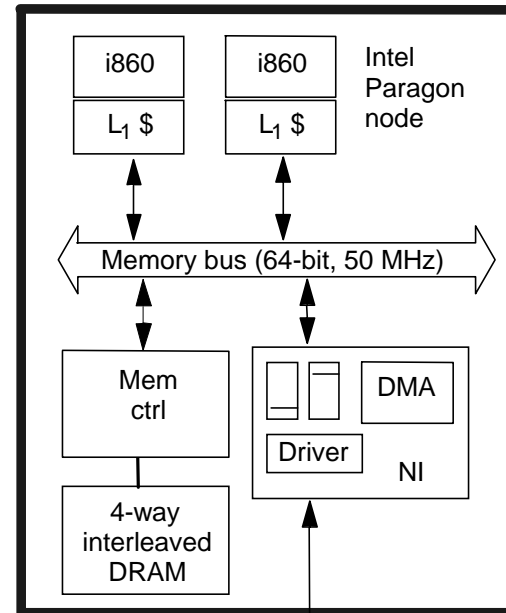


- Made out of essentially complete RS6000 workstations
- Network interface integrated in I/O bus (bw limited by I/O bus)

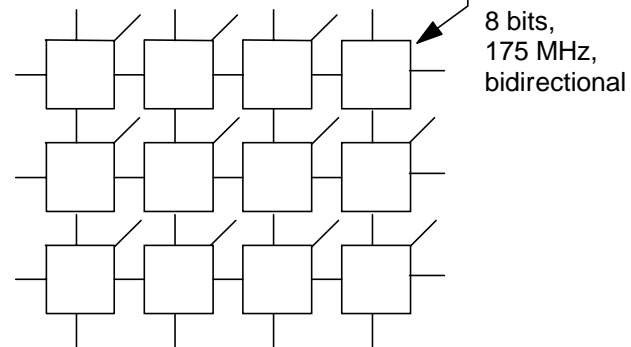
# Example Intel Paragon



Sandia' s Intel Paragon XP/S-based Supercomputer



2D grid network  
with processing node  
attached to every switch



# 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 using hashing
- Page-based (or finer-grained) shared virtual memory

Hardware organization converging too

- Tighter NI integration even for MP (low-latency, high-bandwidth)
- At lower level, even hardware SAS passes hardware messages

Even clusters of workstations/SMPs are parallel systems

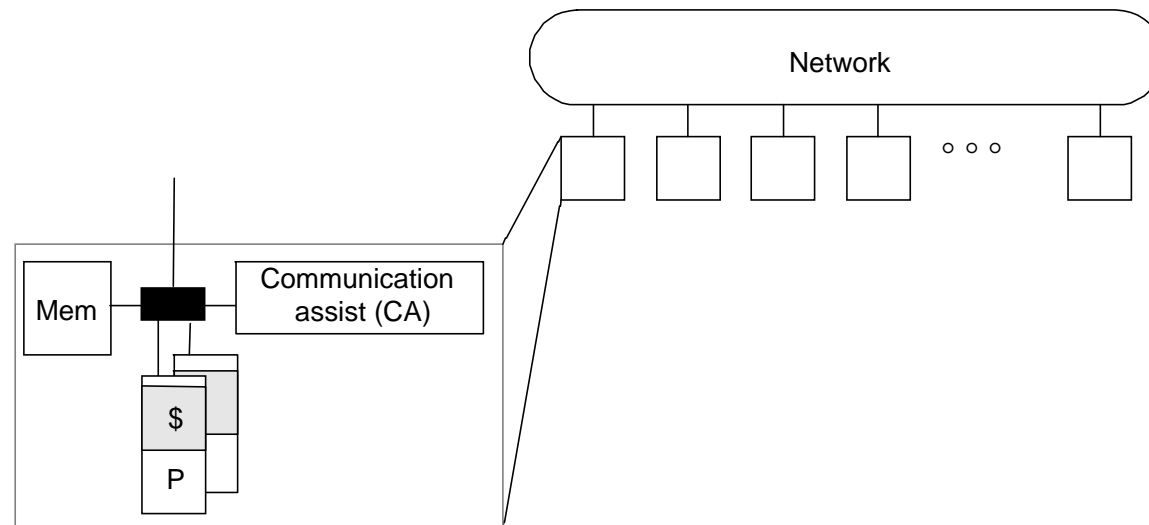
- Emergence of fast system area networks (SAN)

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

A generic modern multiprocessor



Node: processor(s), memory system, plus *communication assist*

- Network interface and communication controller
- Scalable network
- Convergence allows lots of innovation, now within framework
  - Integration of assist with node, what operations, how efficiently...

# **Fundamental Design Issues**

# 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

Understand at programming model first, since that sets requirements

Other issues

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

# Sequential Programming Model

## Contract

- Naming: Can name any variable in virtual address space
  - Hardware (and perhaps compilers) does translation to physical addresses
- Operations: Loads and Stores
- Ordering: Sequential program order

## Performance

- Rely on dependences on single location (mostly): *dependence order*
- Compilers and hardware violate other orders without getting caught
- Compiler: reordering and register allocation
- Hardware: out of order, pipeline bypassing, write buffers
- 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 orders through synchronization
- Again, compilers/hardware can violate orders without getting caught
  - Different, more subtle ordering models also possible (discussed later)

# 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

Can construct global address space:

- Process number + address within process address space
- But no direct operations on these names

# 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, however small

We'll focus on communication or data transfer across nodes

# 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

$$= o_v + o_c + l + n/B + T_c$$

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