Parallel Architecture

15-740 FALL'18

NATHAN BECKMANN

Miscellany

Assignment #1 due next Wednesday! (9/19)

Presentation topics due the Wednesday after that! (9/26)

- Start looking at the topics online
- Get a group together

Today: Parallel architecture

Different schools of parallel architecture

- I.e., programs are written to expose parallelism *explicitly*
- History of unconventional parallel architectures
- Convergence to today's multiprocessor systems

We will learn…

- Why parallelism?
- Different models for parallel execution + associated architectures

Next time: Fundamental challenges (communication, scalability) introduced by parallelism

Why parallelism?

For any given processing element, in principle:
more processing elements ➔ more performance

High-level challenges:

- Parallelism limited by communication barriers
  e.g., cost between chips ≫ cost within a core
- Leads to design tradeoffs
  e.g., N small cores in 1 chip vs. N big cores in N chips
- N processors often != N× better performance
- Parallel programming is often hard
- ➔ What type of parallelism does app exploit best?
  (In practice, machines exploit parallelism at multiple levels)

One definition of parallel architecture

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

Key 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?

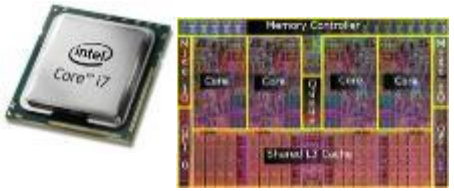
Why study parallel arch & programming?

The Answer from ~15 Years Ago:

- Because it allows you to achieve performance beyond what we get with CPU clock frequency scaling
  - +30% freq/yr vs +40% transistors/yr—**10**× **advantage** over 20 yrs

The Answer Today:

- Because it seems to be the *best available way* to achieve higher performance in the foreseeable future
  - CPU clock rates are no longer increasing! (recall: $P = \frac{1}{2}CV^2F$ and $V \propto F \rightarrow P \propto CF^3$)
  - Instruction-level-parallelism is not increasing either!
  - Improving performance further on sequential code becomes very complicated + diminishing returns
- Without explicit parallelism *or* architectural specialization, performance becomes a zero-sum game.
- Specialization is more disruptive than parallel programming (and is mostly about parallelism anyway)



History

Recurring argument from very early days of computing:

Technology is running out of steam; parallel architectures are more efficient than sequential processors (in perf/mm^2, power, etc)

Except…

- …technology defied expectations (until ~15y ago)

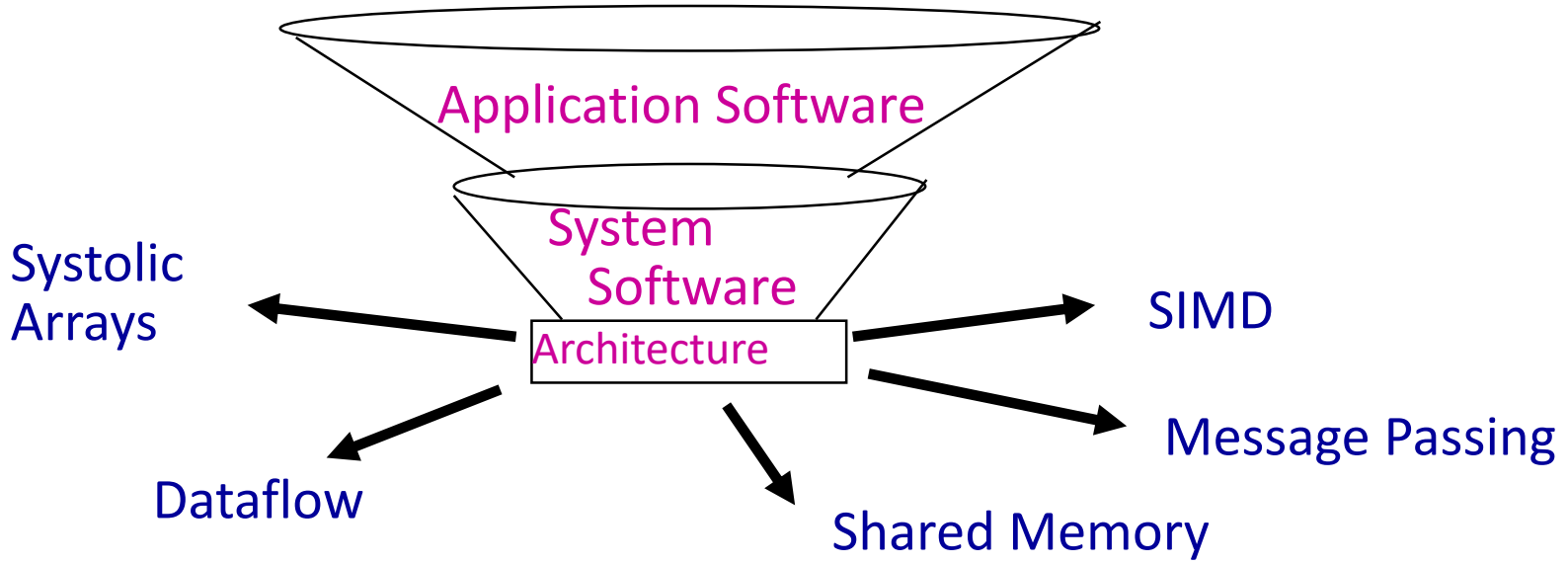
- …parallelism is more efficient in theory, but getting good parallel programs in practice is hard (architecture doesn't exist in a vacuum; see also: scratchpads vs caches)

➔ Sequential/implicitly parallel arch dominant (until ~15y ago)

History

Historically, parallel architectures closely tied to programming models

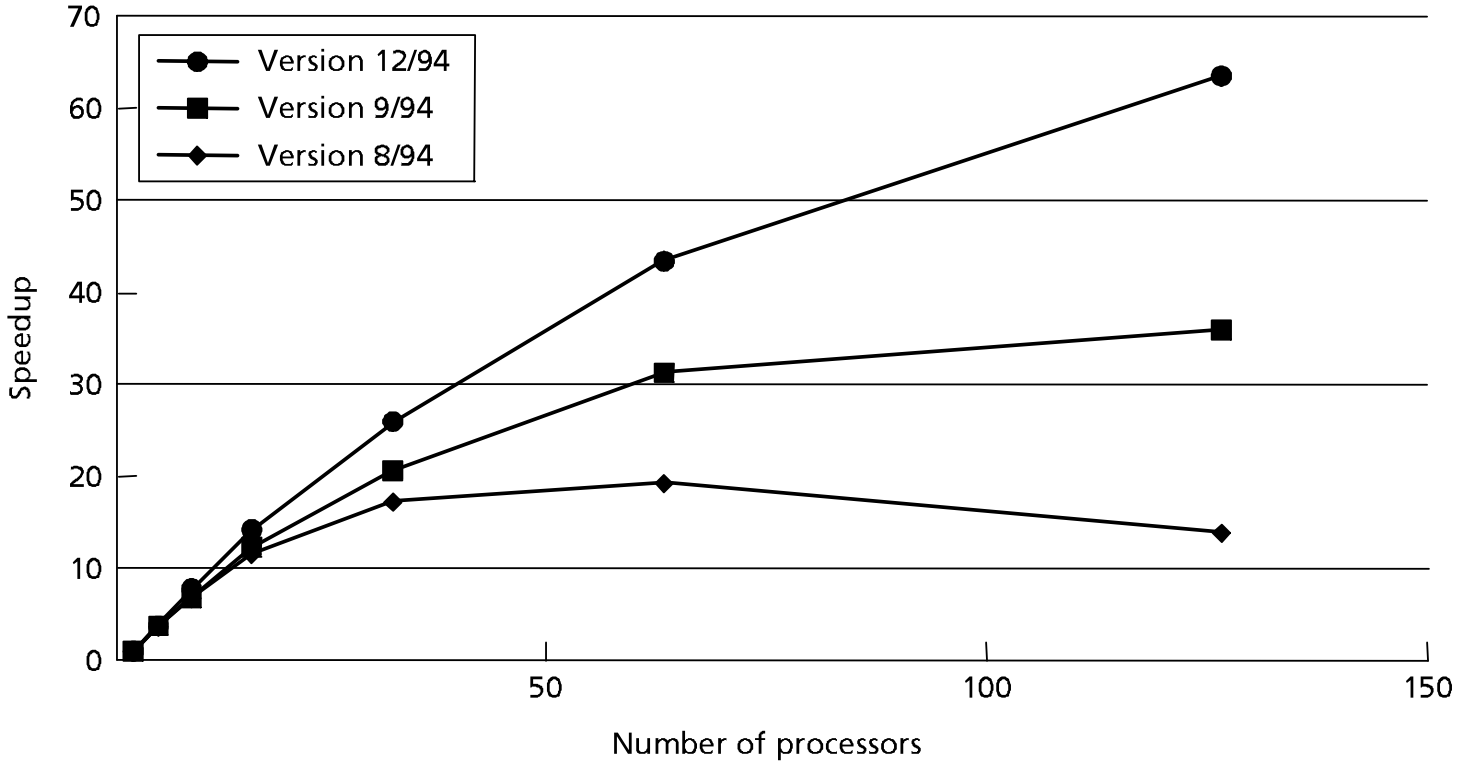
Divergent architectures, with no predictable pattern of growth.



Uncertainty of direction paralyzed parallel software development!
(Parallel programming remains a big problem)

Is parallel architecture enough?

NO. Parallel architectures rely on software for performance!



AMBER code for CRAY-1; ported to Intel Paragon

(slide credit: Culler'99)

Types of parallelism

Bit-level parallelism

- Apply the same operation to many bits at once
- 4004 4b ➔ 8008 8b ➔ 8086 16b ➔ 80386 32b
- E.g., in 8086, adding two 32b numbers takes 2 instructions (add, adc) and multiplies are 4 (mul, mul, add, adc)
- Early machines used transistors to widen datapath
- Aside: 32b ➔ 64b mostly not for performance, instead…
  - Floating point precision
  - Memory addressing (more than 4GB)

- *Not what people usually mean by parallel architecture today!*

Types of parallelism

Instruction-level parallelism (ILP)

- Different instructions within a stream can be executed in parallel
- Pipelining, out-of-order execution, speculative execution, VLIW
- Dataflow

```
A:  LD R2, 0(R1)
    LD R3, 4(R1)
    SUBI R2, R2, #1
    SUBI R3, R3, #1
    BLTZ R2, B
    ST R2, 0(R1)
B:  BLTZ R3, C
    ST R3, 4(R1)
C:  ADDI R1, R1, #8
    SUB R5, R4, R1
    BGTZ R4, A
    RET
```

```
void decrement_all(
    int *array,
    int size) {
  for (int i = 0;
    i < size;
    i++) {
     int x = array[i] - 1;
     if (x > 0) {
       array[i] = x;
     }
  }
}
```

Loop unrolled x2

Types of parallelism

Instruction-level parallelism (ILP)

- Different instructions within a stream can be executed in parallel
- Pipelining, out-of-order execution, speculative execution, VLIW
- Dataflow

```
A:  LD R2, 0(R1)
    LD R3, 4(R1)
    SUBI R2, R2, #1
    SUBI R3, R3, #1
    BLTZ R2, B
    ST R2, 0(R1)
B:  BLTZ R3, C
    ST R3, 4(R1)
C:  ADDI R1, R1, #8
    SUB R5, R4, R1
    BGTZ R4, A
    RET
```

```
void decrement_all(
    int *array,
    int size) {
  for (int i = 0;
    i < size;
    i++) {
     int x = array[i] - 1;
     if (x > 0) {
       array[i] = x;
     }
  }
}
```

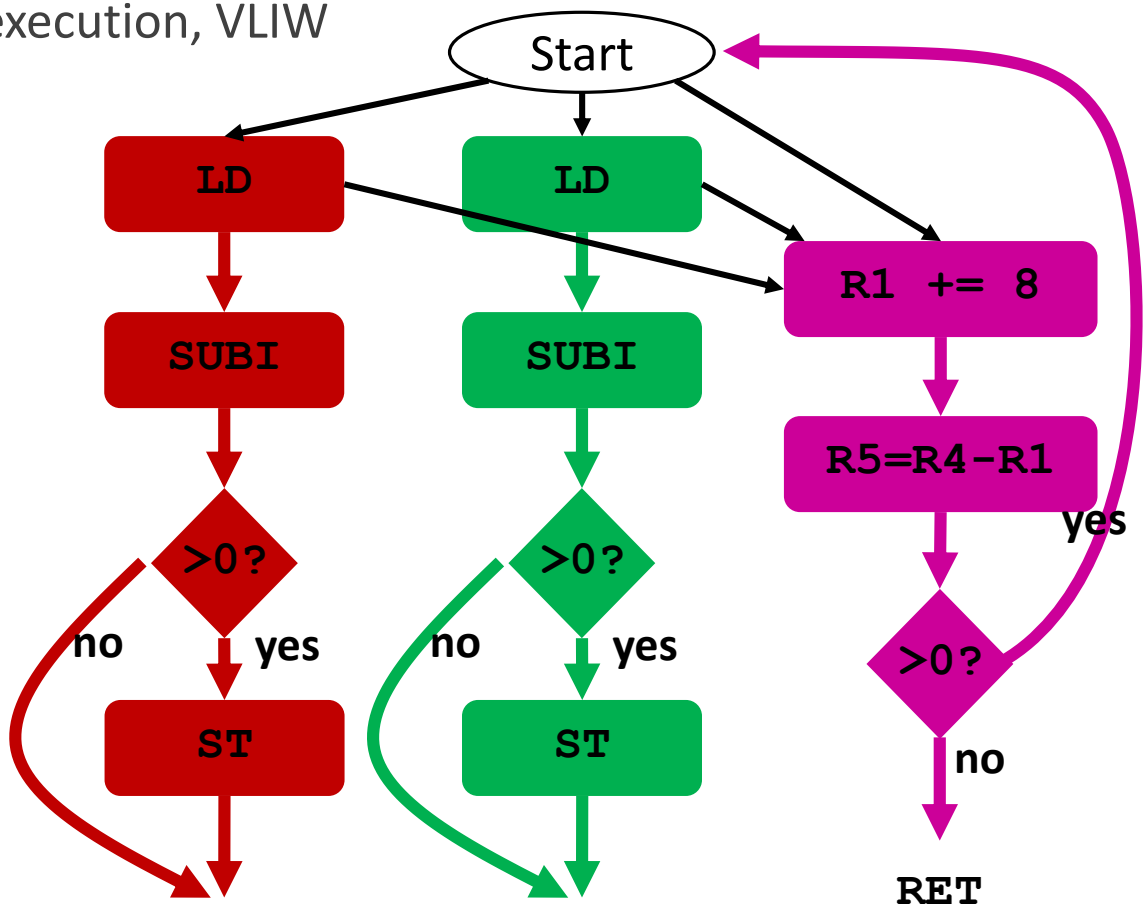
Loop unrolled x2

Types of parallelism

Instruction-level parallelism (ILP)

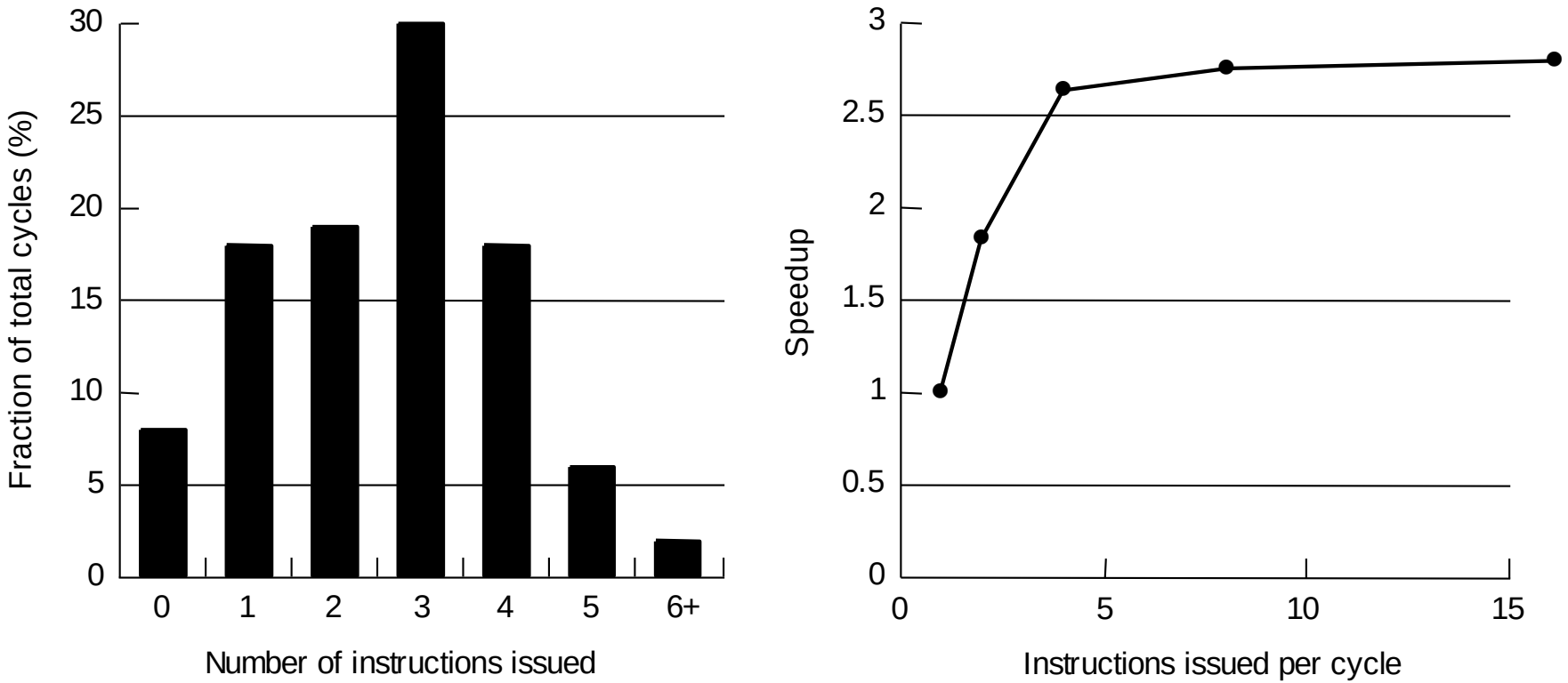
- Different instructions within a stream can be executed in parallel
- Pipelining, out-of-order execution, speculative execution, VLIW
- Dataflow

```
A:  LD R2, 0(R1)
    LD R3, 4(R1)
    SUBI R2, R2, #1
    SUBI R3, R3, #1
    BLTZ R2, B
    ST R2, 0(R1)
B:  BLTZ R3, C
    ST R3, 4(R1)
C:  ADDI R1, R1, #8
    SUB R5, R4, R1
    BGTZ R4, A
    RET
```



Types of parallelism

Instruction-level parallelism peaks @ ~4 ins / cycle



Real programs w realistic cache+pipeline latencies, but unlimited resources

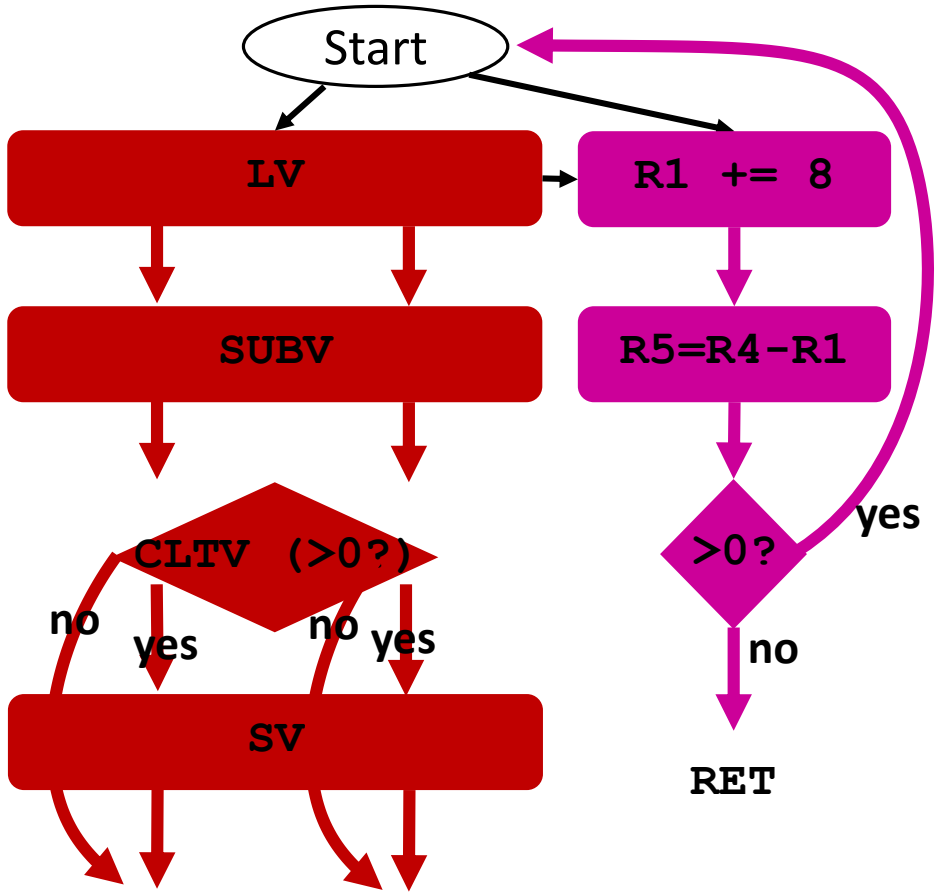
Types of parallelism

Data Parallelism

- Different pieces of data can be operated on in parallel
- Vector processing, array processing
- Systolic arrays, streaming processors

(Not valid assembly)

```
    LUI VLR, #2
A:  LV V1, 0(R1)
    SUBV V1, #1
    SLTV V1, #0
    SV V1, 0(R1)
    ADDI R1, R1, #8
    SUB R5, R4, R1
    BGTZ R5, A
    RET
```



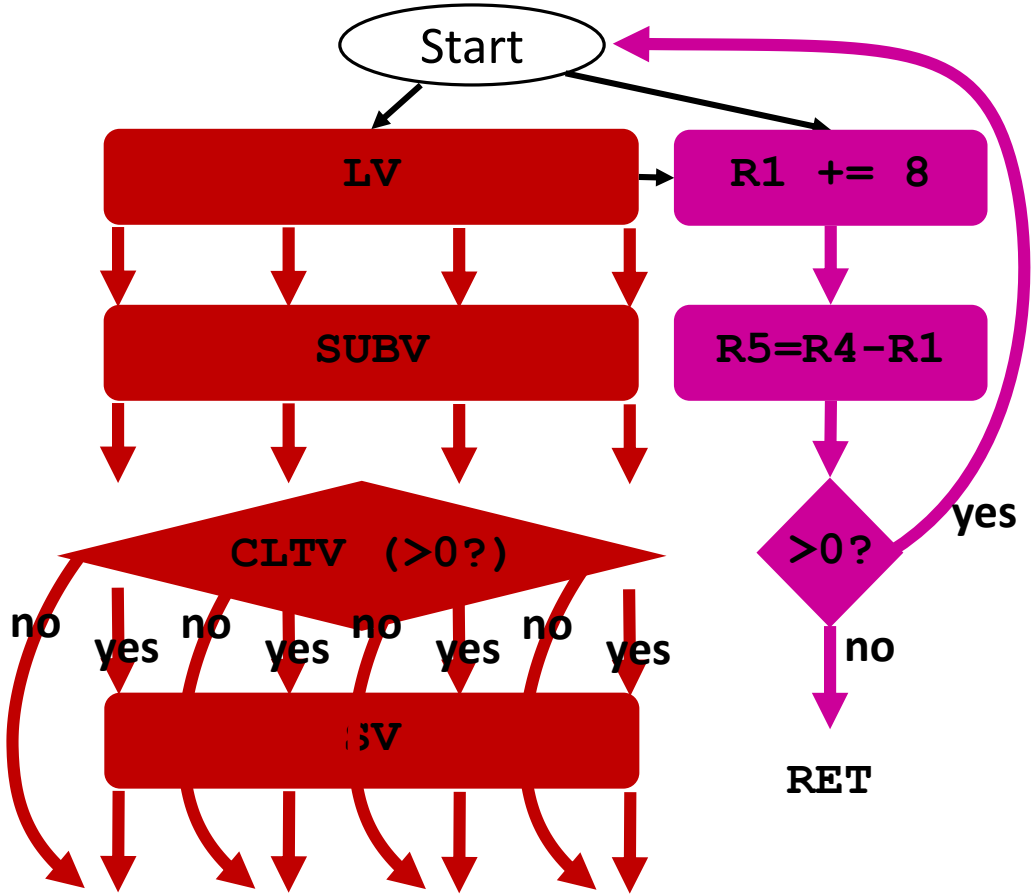
Types of parallelism

Data Parallelism

- Different pieces of data can be operated on in parallel
- Vector processing, array processing
- Systolic arrays, streaming processors

(Not valid assembly)

```
    LUI VLR, #4
A:  LV V1, 0(R1)
    SUBV V1, #1
    CLTV V1, #0
    SV V1, 0(R1)
    ADDI R1, R1, #16
    SUB R5, R4, R1
    BGTZ R5, A
    RET
```



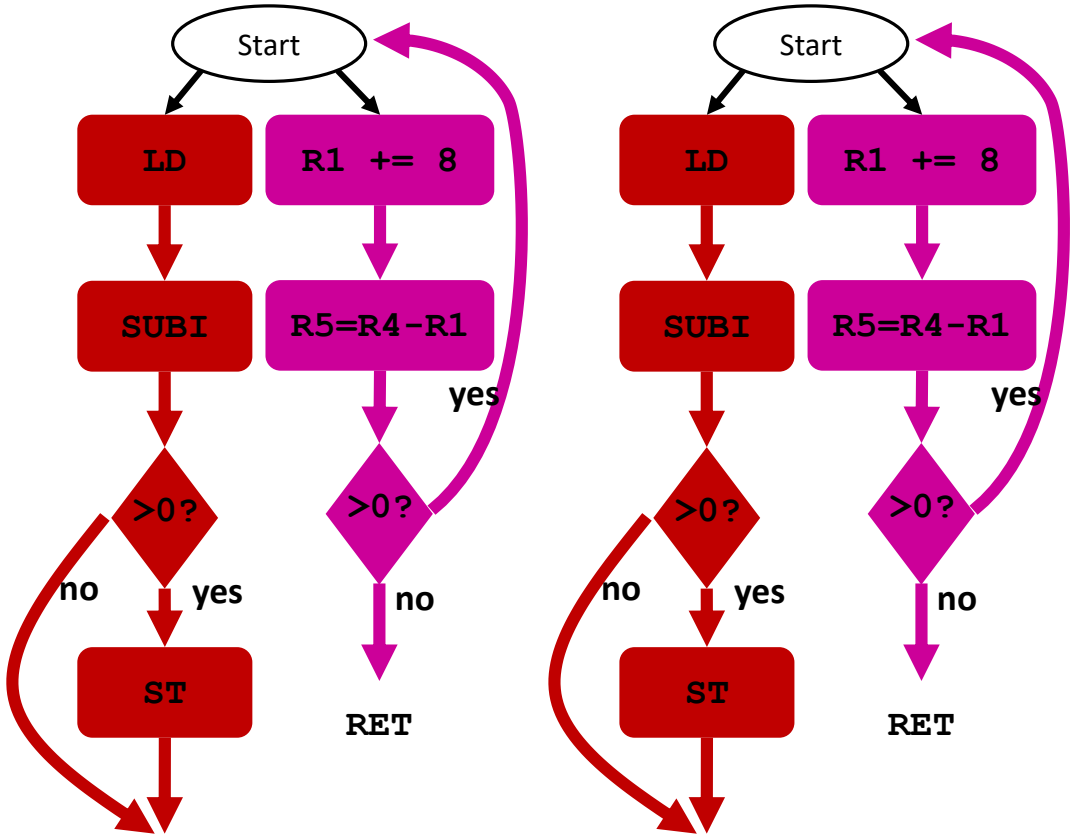
Types of parallelism

Task Level Parallelism

- Different “tasks/threads” can be executed in parallel
- Multithreading
- Multiprocessing (multi-core)

Adjust R1, R5 per thread…

```
A:  LD R2, 0(R1)
    SUBI R2, #1
    BLTZ R2, #0
    ST R2, 0(R1)
    ADDI R1, R1, #4
    SUB R5, R4, R1
    BGTZ R4, A
    RET
```



Types of parallelism

Instruction Level Parallelism
- Different instructions within a stream can be executed in parallel
- Pipelining, out-of-order execution, speculative execution, VLIW
- Dataflow

Data Parallelism
- Different pieces of data can be operated on in parallel
- SIMD: Vector processing, array processing
- Systolic arrays, streaming processors

Task Level Parallelism
- Different “tasks/threads” can be executed in parallel
- Multithreading
- Multiprocessing (multi-core)

Flynn's Taxonomy of Computers

Mike Flynn, "Very High-Speed Computing Systems," 1966

SISD: Single instruction operates on single data element

SIMD: Single instr operates on multiple data elements

- Array processor
- Vector processor

MISD: Multiple instrs operate on single data element

- Closest form?: systolic array processor, streaming processor

MIMD: Multiple instructions operate on multiple data elements (multiple instruction streams)

- Multiprocessor
- Multithreaded processor

Parallel architecture

Extension of “computer architecture” to support communication and cooperation

- OLD: Instruction Set Architecture
- NEW: *Communication Architecture*

Communication architecture defines

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

Compilers, libraries and OS are crucial bridges

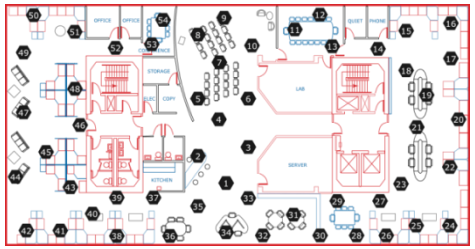
Convergence crosses parallel architectures to include what historically were distributed systems

Concurrent Systems

Embedded-Physical Distributed

Claytronics

Sensor Networks

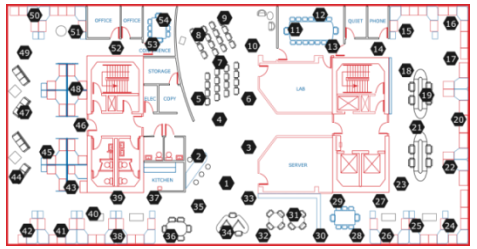


Concurrent Systems

Embedded-Physical Distributed

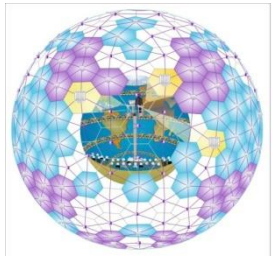
Claytronics

Sensor Networks



Geographically Distributed

Internet



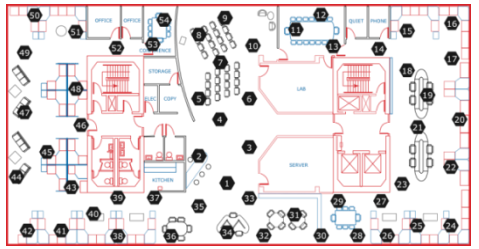
Power Grid

Concurrent Systems

Embedded-Physical Distributed

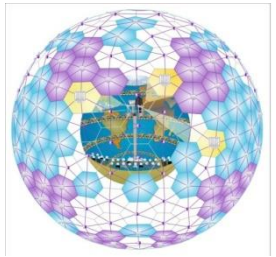
Claytronics

Sensor Networks



Geographically Distributed

Internet

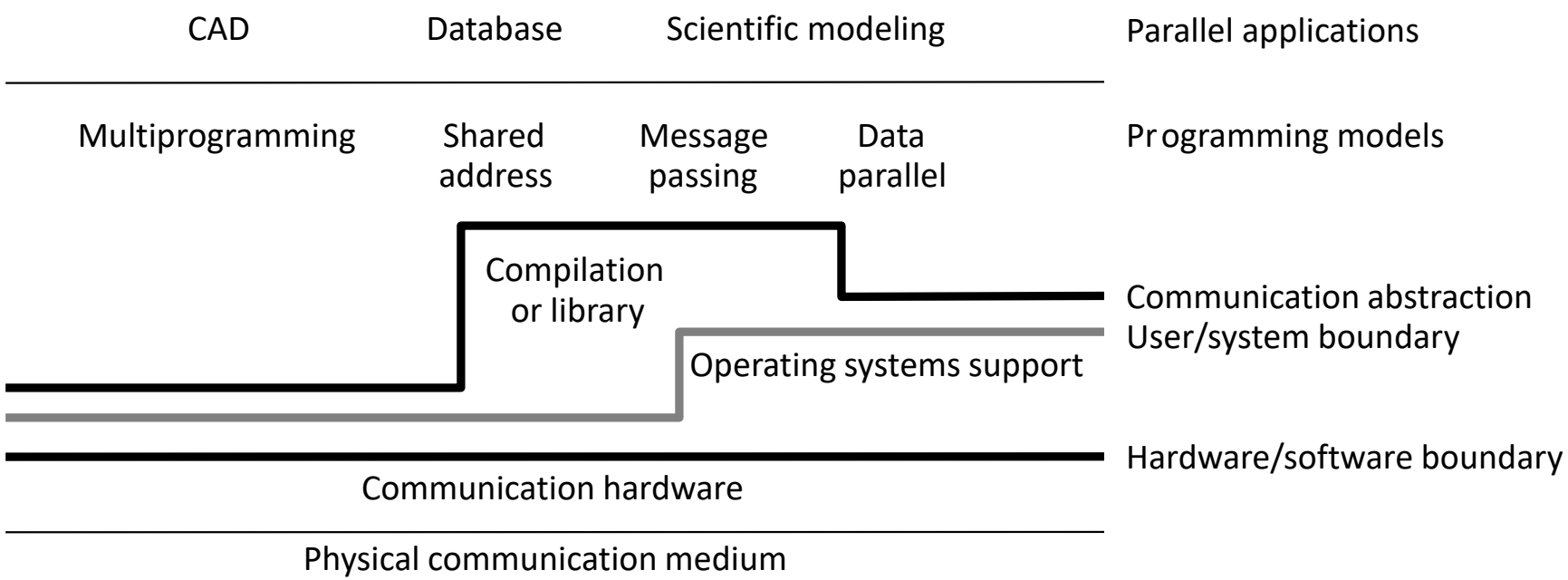


Power Grid

Cloud Computing

EC2 Azure

Modern Layered Framework



Programming Model

What programmer uses in coding applications

Specifies operations, naming, and ordering – focus on communication and synchronization

Examples:

- Multiprogramming: no communication or synch. at program level
- *Shared address space*: like bulletin board, need separate synchronization (eg, atomic operations)
- *Message passing*: like letters or phone calls, explicit point-to-point messages act as both communication and synchronization
- *Data parallel*: more regimented, global actions on data

Communication Abstraction

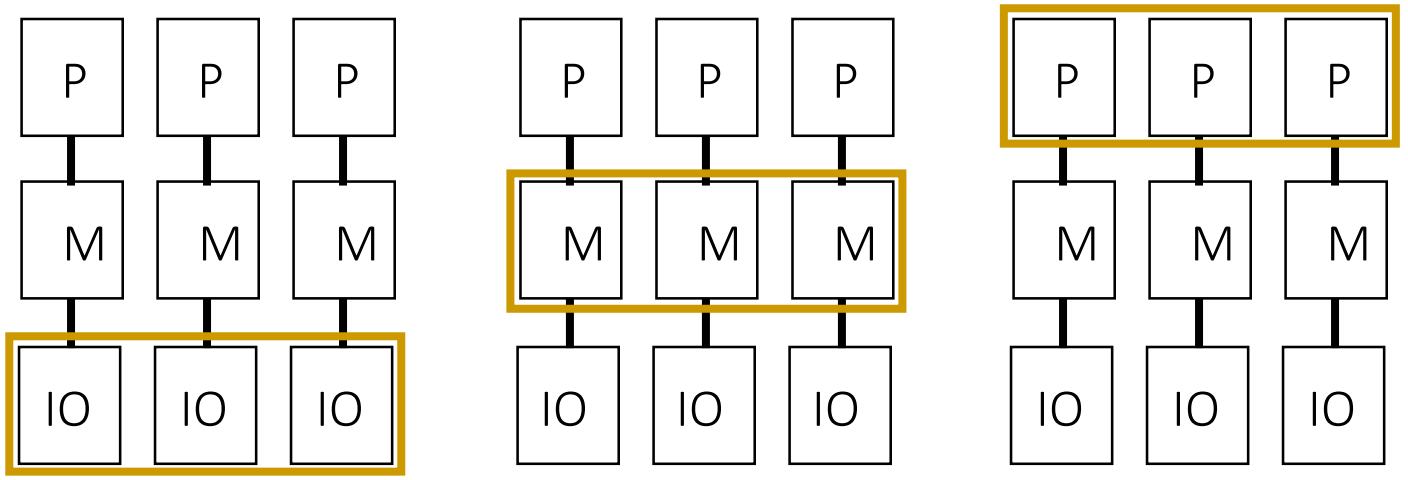
User-level communication primitives provided

- Realizes the programming model
- Mapping exists between language primitives of programming model and these primitives

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

Lot of debate about what to support in sw and gap between layers

Where Communication Happens



Join At:	I/O (Network)	Memory	Processor
Program With:	Message Passing	Shared Memory	Dataflow/Systolic

Evolution of Architectural Models

Historically, machines tailored to programming models

- Programming model, communication abstraction, and machine organization lumped together as the "architecture"

Most Common Models:

- Shared Address Space, Message Passing, Data Parallel

Other Models:

- Dataflow, Systolic Arrays

Let's examine each programming model, its motivation, intended applications, and contributions to convergence

Shared Address Space (SAS) Architectures

Any processor can directly reference any memory location

- Communication occurs implicitly as result of loads and stores

Convenient:

- Location transparency (don't need to worry about physical placement of data)
- 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

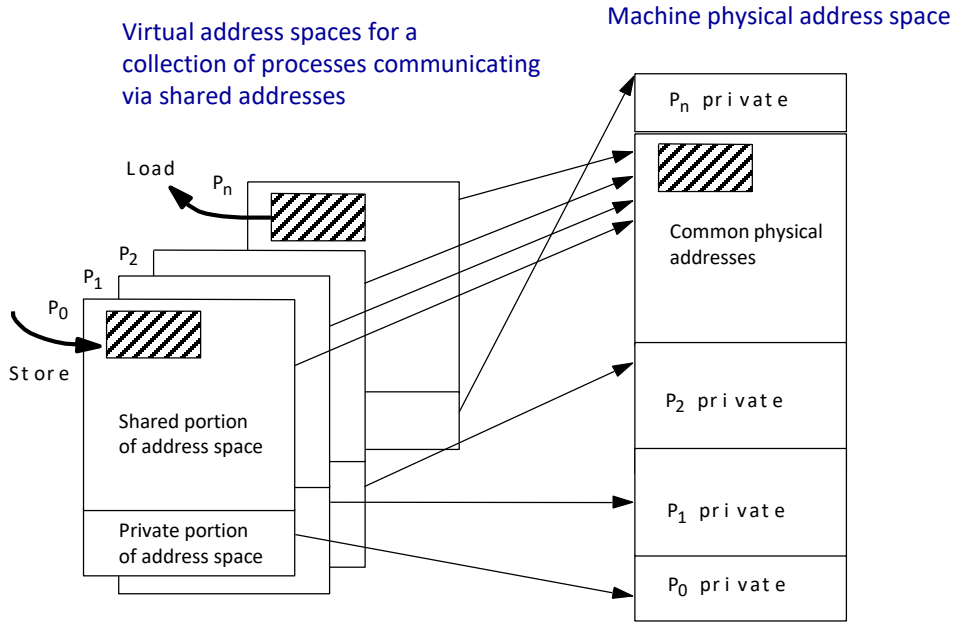
- History dates at least to precursors of mainframes in early 60s
- Wide range of scale: few to hundreds of processors

Popularly known as *shared-memory* machines / model

- Ambiguous: memory may be physically distributed among processors

SAS Programming Model

Process: virtual address space plus one or more threads of control
Portions of address spaces of processes are shared

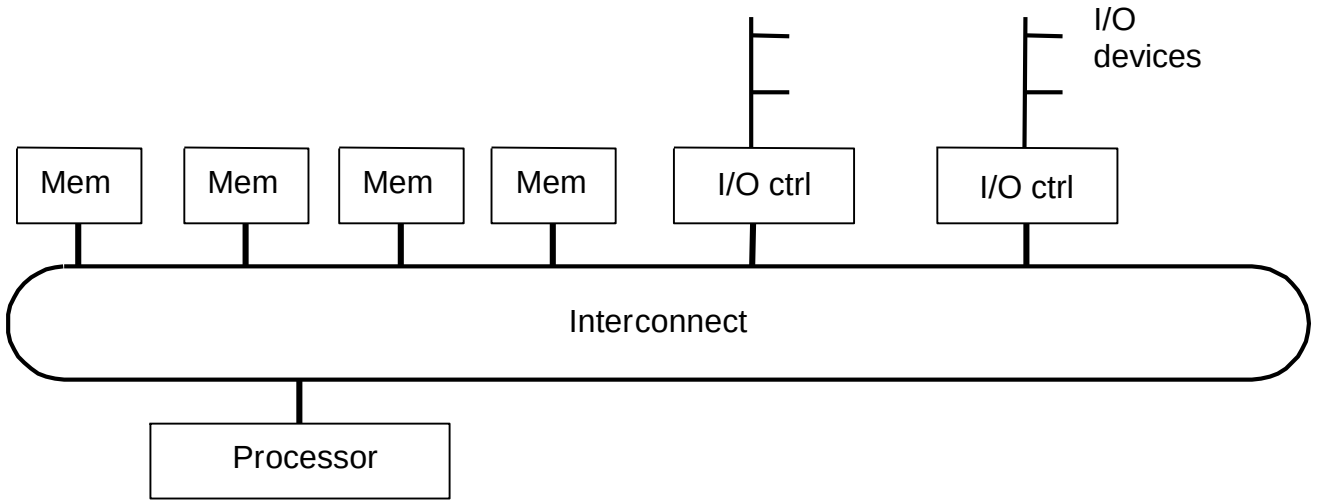


- Writes to shared address visible to other threads, processes
- OS uses shared memory to coordinate processes

SAS Communication Hardware

Also a natural extension of a uniprocessor

Already have processor, one or more memory modules and I/O controllers connected by hardware interconnect of some sort

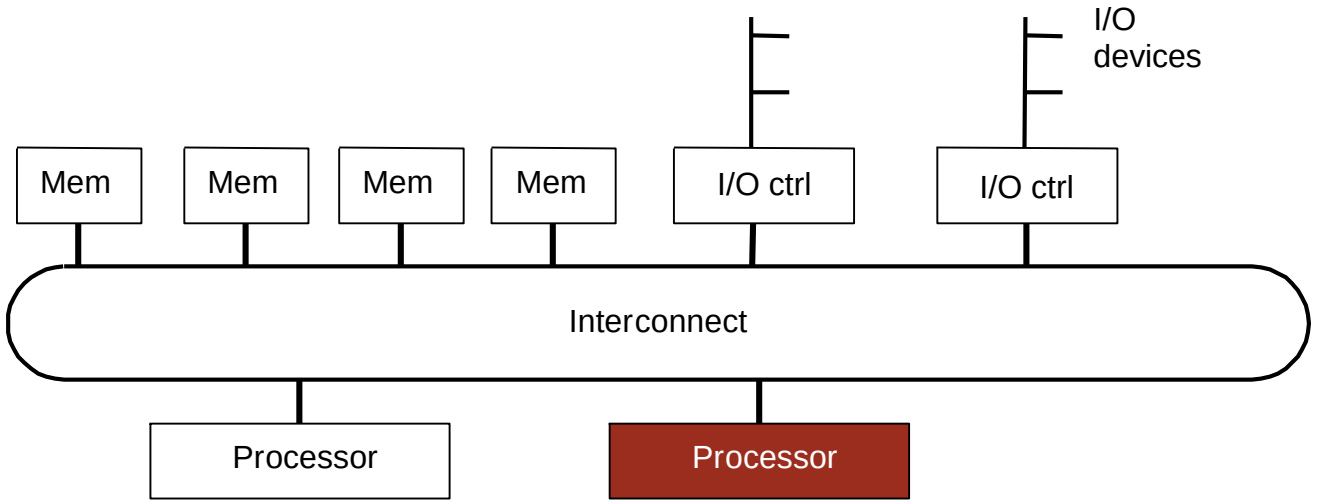


Memory capacity increased by adding modules, I/O by controllers

SAS Communication Hardware

Also a natural extension of a uniprocessor

Already have processor, one or more memory modules and I/O controllers connected by hardware interconnect of some sort



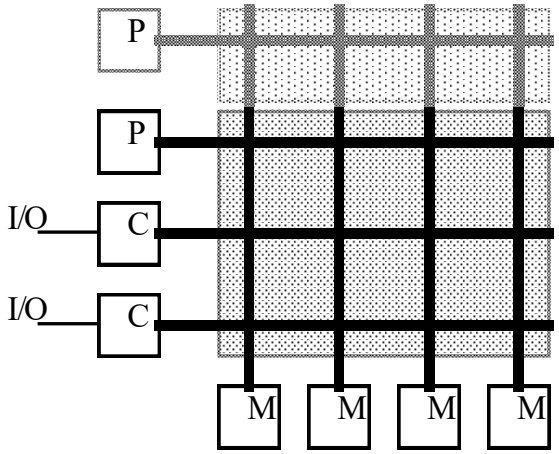
Memory capacity increased by adding modules, I/O by controllers

➔ **Add processors for processing!**

SAS History

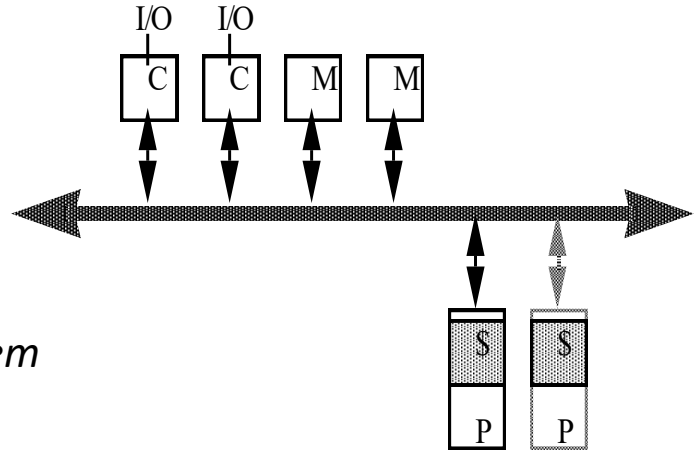
"Mainframe" approach:

- Motivated by multiprogramming
- Extends crossbar used for memory and I/O
- At first, processor cost limited scaling, then crossbar itself
- + **Bandwidth scales with *P***
- – **High incremental cost** ➔ use multistage instead



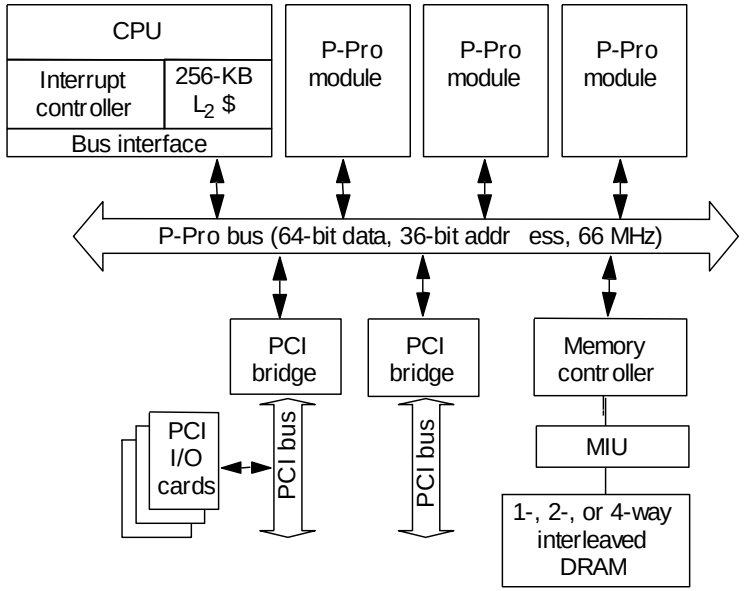
"Minicomputer" approach:

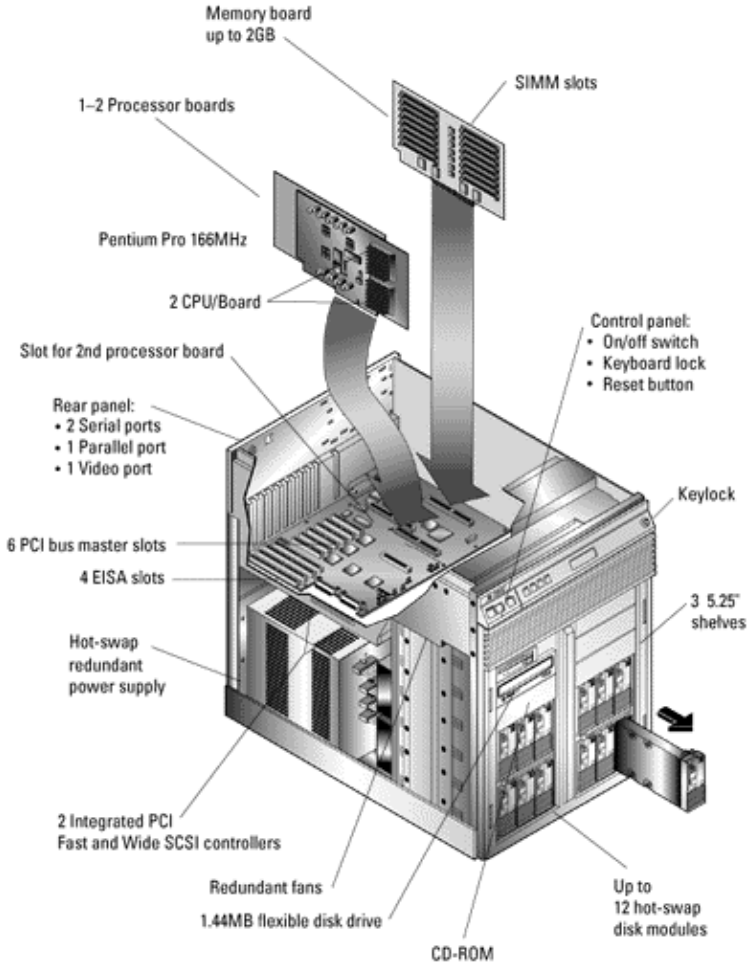
- Almost all microprocessor systems have bus
- Motivated by multiprogramming & task parallelism
- Called symmetric multiprocessor (SMP)
- Latency larger than for uniprocessor
- + **Low incremental cost**
- – **Bus is bandwidth bottleneck** ➔ caching ➔ *coherence problem*



Example: Intel Pentium Pro Quad (’98)

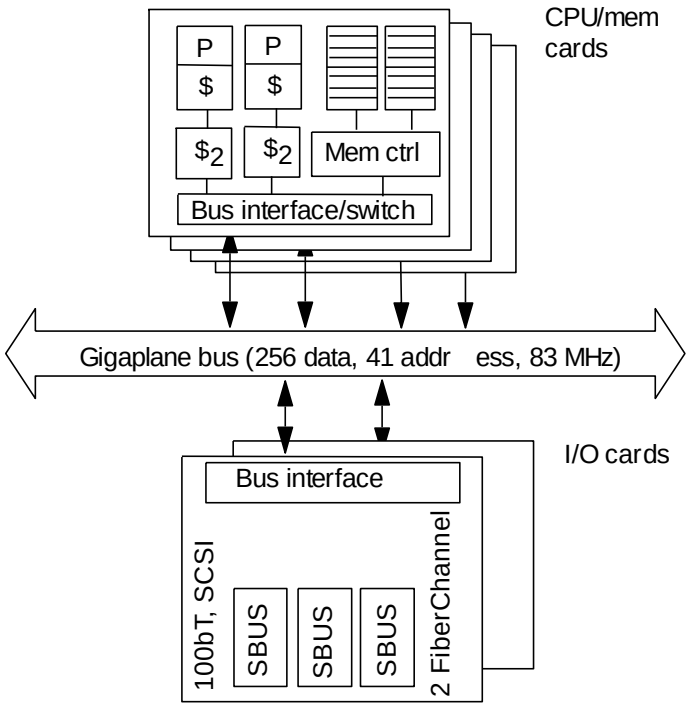
- All coherence and multiprocessing glue in processor module
- Highly integrated, targeted at high volume
- **Low latency and bandwidth**
- Current Intel chips use point-to-point network (QPI) instead



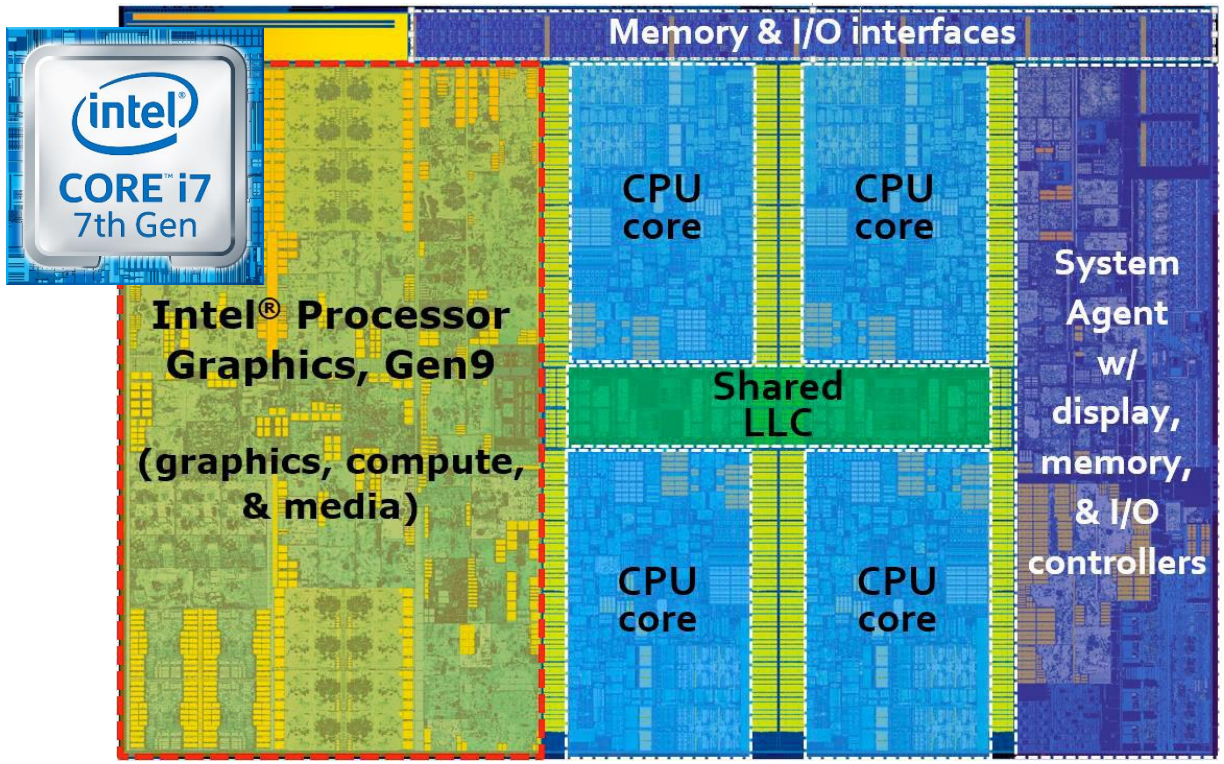


Example: SUN Enterprise (‘96)

- 16 cards of either type: processors + memory, or I/O
- Up to 30 processors
- All memory accessed over bus, so symmetric
- Higher bandwidth, higher latency bus



Recent (‘17) x86 Example

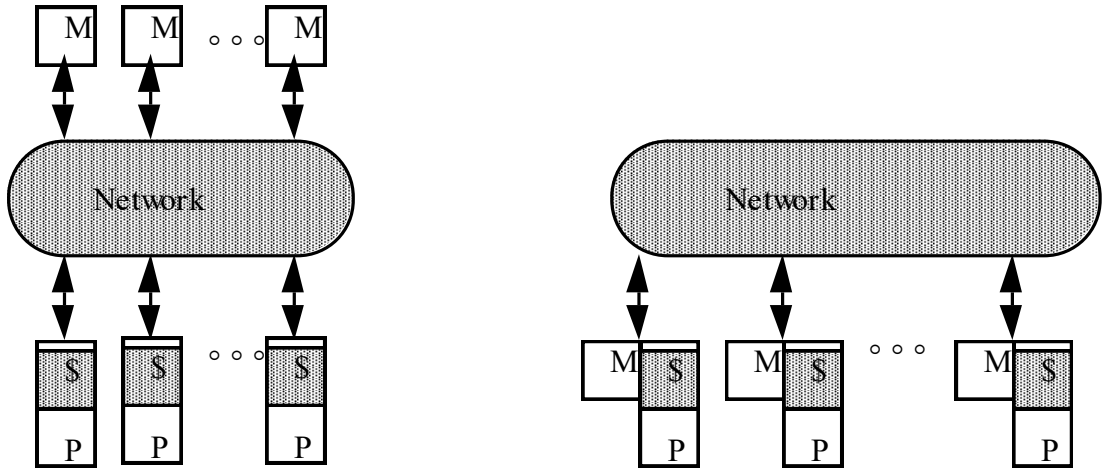


Intel’s Core i7 7th generation

- Highly integrated, commodity systems
- On-chip: low-latency, high-bandwidth communication via shared cache
- Current scale = ~4 processors (up to 12 on some models, more on server parts)

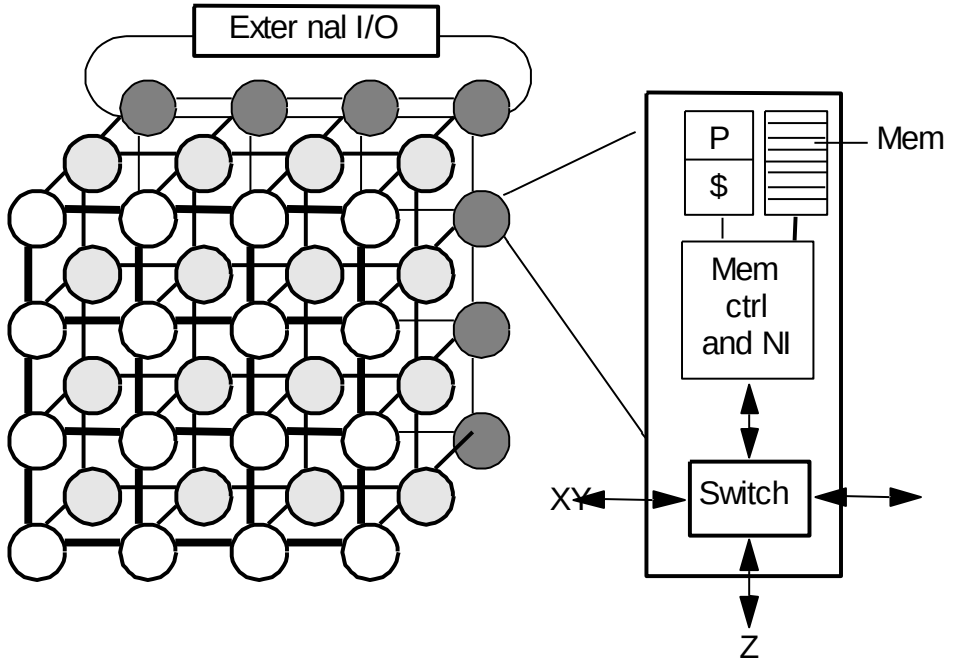
Scaling Up

- Problem is interconnect: cost (crossbar) or bandwidth (bus)
- "Dance-hall" topologies: 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?



"Dance hall" Distributed memory

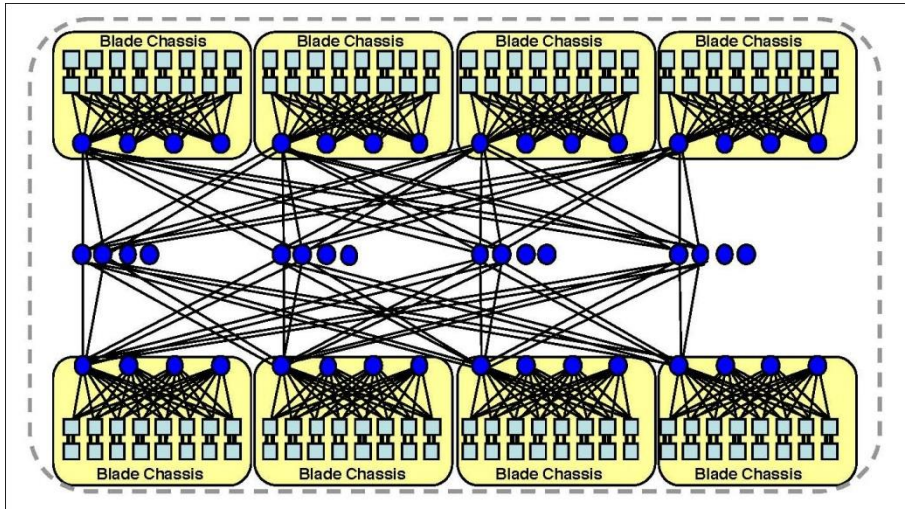
Example: Cray T3E (’96)



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

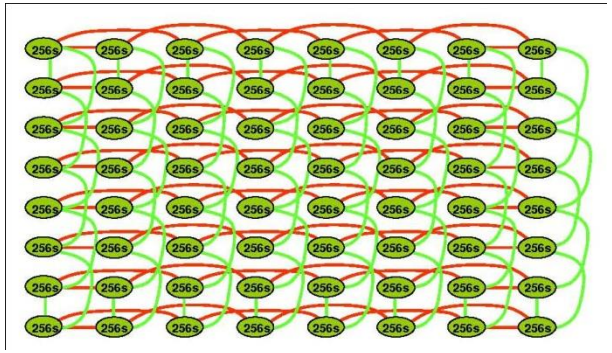
Example: SGI Altix UV 1000 (‘09)

Blacklight at the PSC (4096 cores)



256 socket (2048 core) fat-tree
(this size is doubled in Blacklight via a torus)

- Scales up to 131,072 Xeon cores
- 15GB/sec links
- Hardware cache coherence for blocks of 16TB with 2,048 cores



8x8 torus

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)
- communicate via explicit messages (send/receive)

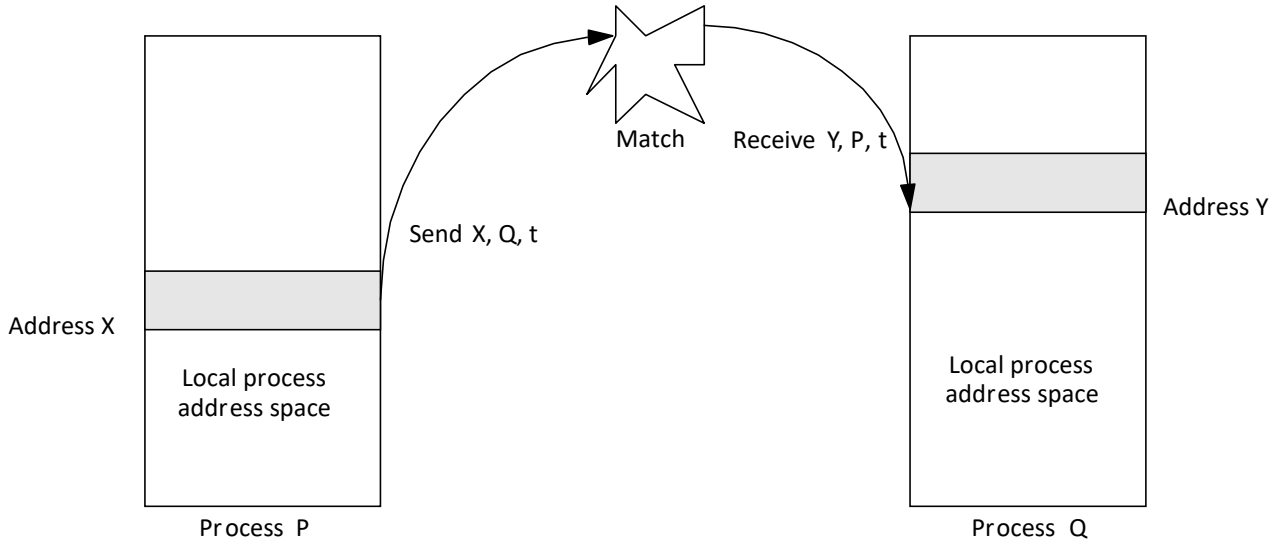
High-level block diagram similar to distributed-mem SAS

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

Programming model further from basic hardware ops

- 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
- Semantics: Memory to memory copy, but need to name processes
  - Optional tag on send and matching rule on receive
- In simplest form, the send/recv match achieves pairwise synch event
  - Other variants too (asynch message passing)
- Many overheads: copying, buffer management, protection

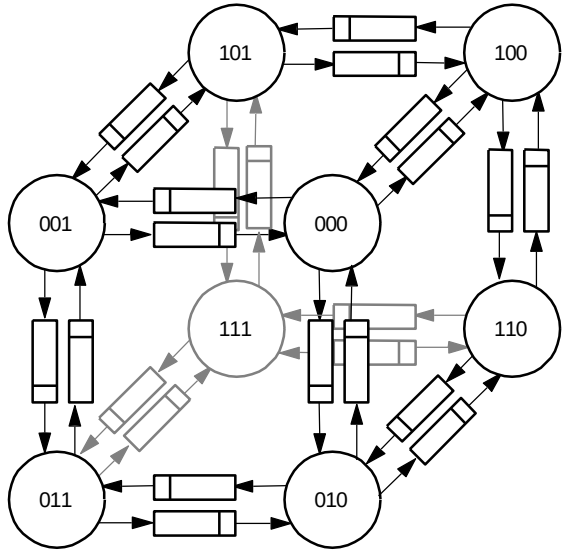
Evolution of Message Passing

Early machines: FIFO on each link

- Hardware close to programming 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



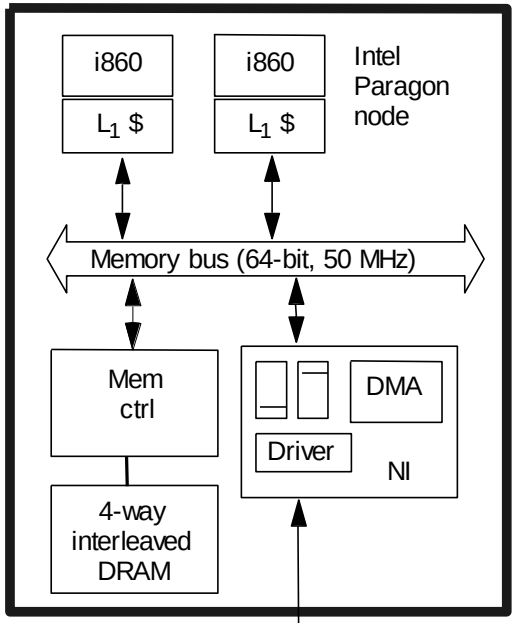
Example: Intel Paragon (‘93)

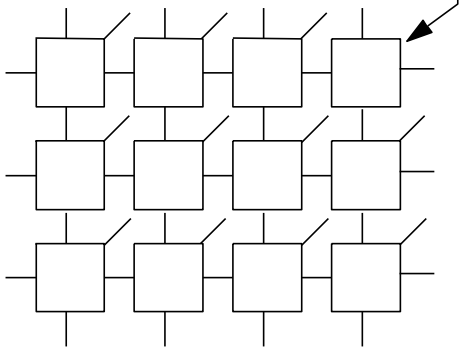
Intel RISC processor (!!)

Up to 2048 nodes

B/W limited by I/O bus

Sandia’s Intel Paragon XP/S-based Supercomputer

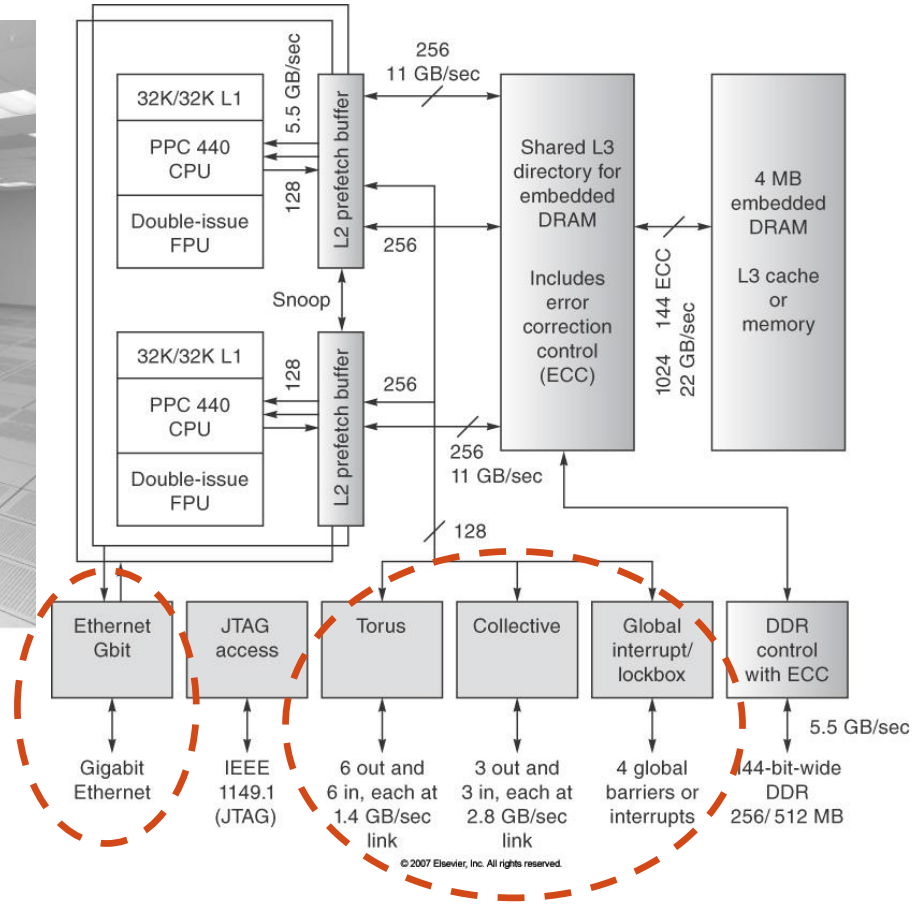




8 bits, 175 MHz, bidirectional

2D grid network with processing node attached to every switch

Example: IBM Blue Gene/L ('04)



Nodes: 2 low-power PowerPC 400s

Everything but DRAM on-chip

Up to 64K nodes

Most power supercomputer for 3.5 years (until 2008)

First machine to achieve 100 TFLOPs on real application

Example: IBM Blue Gene/Q (‘11)

Successor to Blue Gene/L

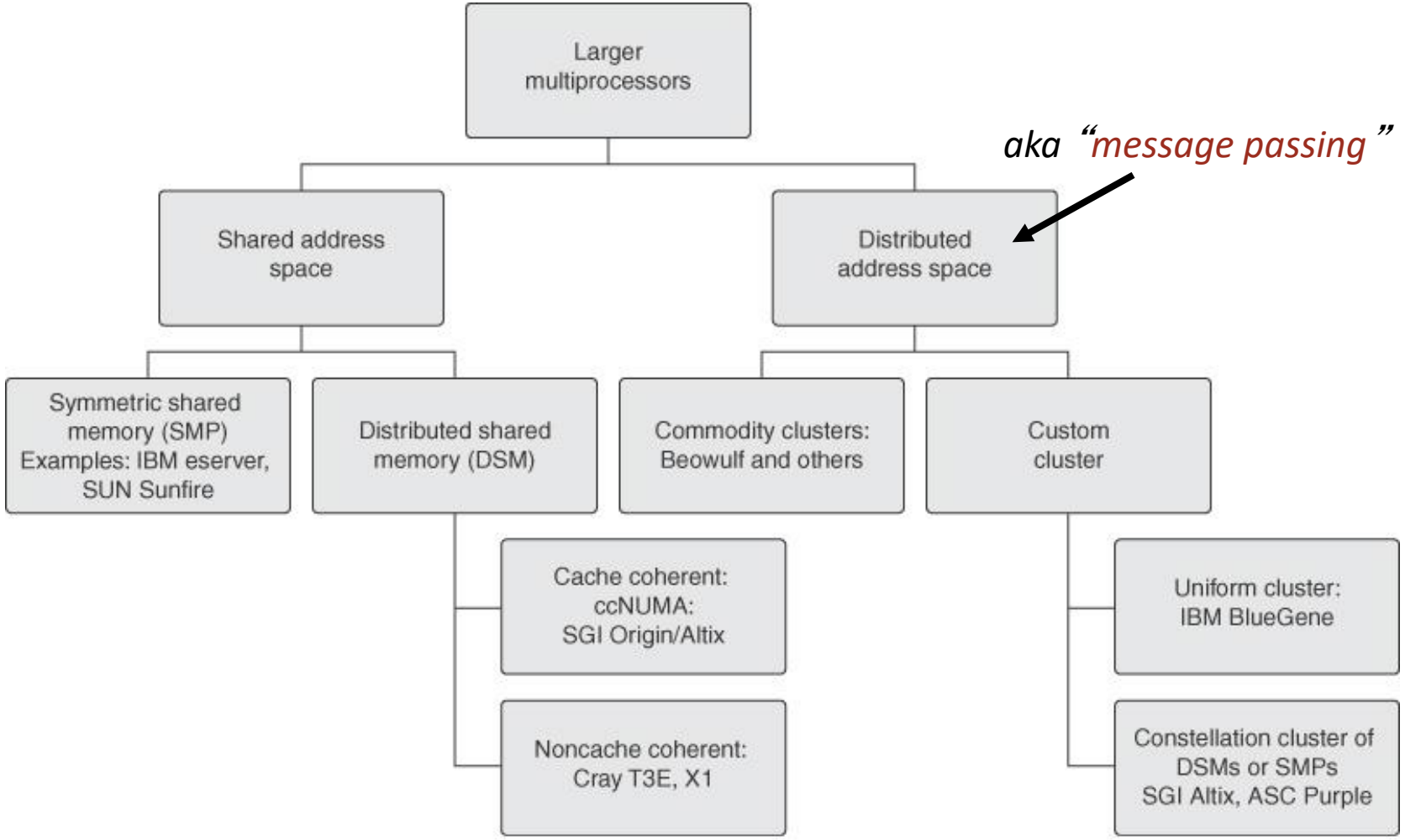
Node: 18 cores,
4-way issue @ 1.6GHz,
SIMD (vector) instructions,
coherence within node

16 user cores (1 for OS, 1 spare)

Top of “green Top500” (2.1GFLOPS/W)

First to achieve 10PFLOPS on real application
(100x BQ/L)

Taxonomy of Large-Scale SAS and MP Systems



Towards 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 converging too

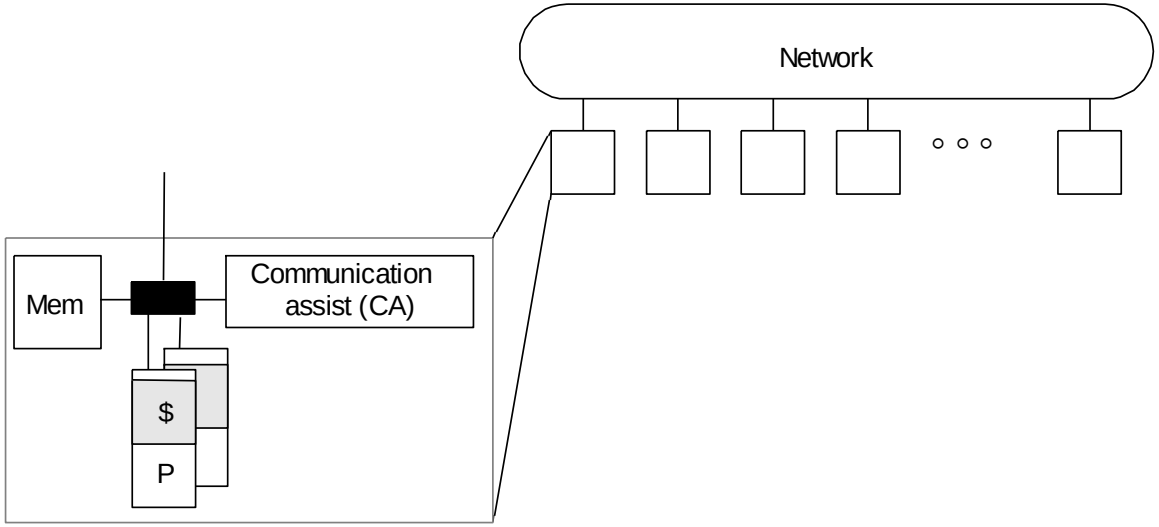
- Tighter NI integration even for MP (low-latency, high-bandwidth)
- At lower level, even hardware SAS passes hardware messages

Programming models distinct, but organizations converging

- Nodes connected by general network and communication assists
- Implementations also converging, at least in high-end machines

Convergence: General 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...**

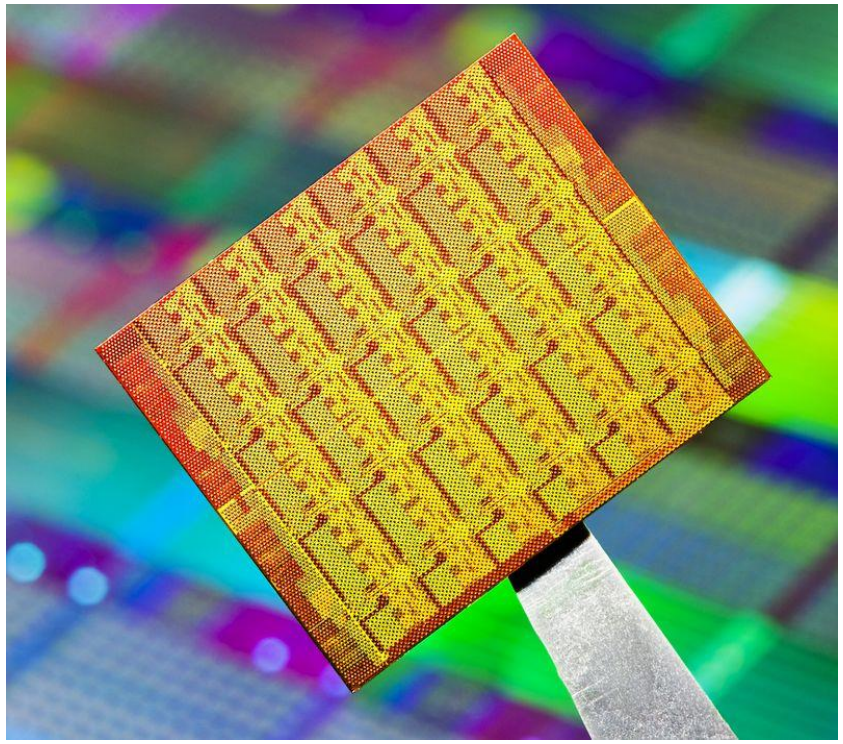
Intel Single-chip Cloud Computer (‘09)

48 cores

2D mesh network

- 24 tiles in 4x6 grid
- 2 cores / tile
- 16KB msg buffer / tile

4 DDR3 controllers



No hardware coherence

Message passing hardware

Coherence available through software library

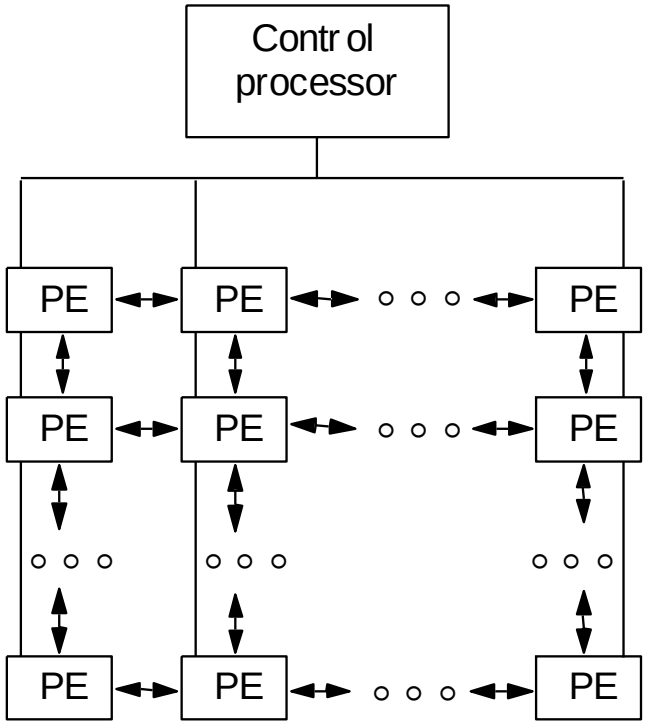
Data Parallel Systems

Programming model:

- Operations performed in parallel on each element of data structure
- Logically single thread of control, performs sequential or parallel steps
- Conceptually, a processor associated with each data element

Architectural model:

- Array of many simple, cheap processors with little memory each
  - Processors don't sequence through instructions
- Attached to a control processor that issues instructions
- Specialized and general communication, cheap global synchronization



Original motivation:

- **Matches simple differential equation solvers**
- **Centralize high cost of instruction fetch & sequencing**

Example Application of Data Parallelism

- Each PE contains an employee record with his/her salary

```
If salary > 100K then
   salary = salary * 1.05
else
   salary = salary * 1.10
```

- Logically, the whole operation is a single step
- Some processors enabled for arithmetic operation, others disabled

Other examples:

- Finite differences, linear algebra, ...
- Document searching, graphics, image processing, ...

Some examples:

- Thinking Machines CM-1, CM-2 (and CM-5)
- Maspar MP-1 and MP-2,

Evolution and Convergence

Rigid control structure (SIMD in Flynn taxonomy)

Popular when cost savings of centralized sequencer high (‘70s – ‘80s)

- 60s when CPU was a cabinet; replaced by vectors in mid-70s
- Revived in mid-80s when 32-bit datapath slices just fit on chip

Decline in popularity (‘90s – ’00s)

- Caching, pipelining, and online scheduling (somewhat) weakened this argument
- Simple, regular applications have good locality, can do well anyway
- Loss of generality due to hardwiring data parallelism
- ➔ MIMD machines also effective for data parallelism and more general
- ➔ SPMD (single program, multiple data) model for data-parallel execution on multiprocessors
  - Need hardware support for fast global synchronization

Resurgence (‘10s – now)

- Power dominant concern
- SIMD amortizes fetch & decode energy

Lasting Contributions of Data Parallel

"Multimedia extensions" of ISAs (e.g., SSE)

- Limited SIMD for 4-8 lanes
- Called "vector instructions" but **not really** traditional "vector architecture"

GPGPU computing

- Programming model looks like MIMD, but processor actually executes multi-threaded SIMD
- GPU jargon: vector lane == "core"
  ➔ 1000s of cores
- Reality: 16-64 multithreaded SIMD (vector) cores

Example: Nvidia Pascal 100 (‘16)

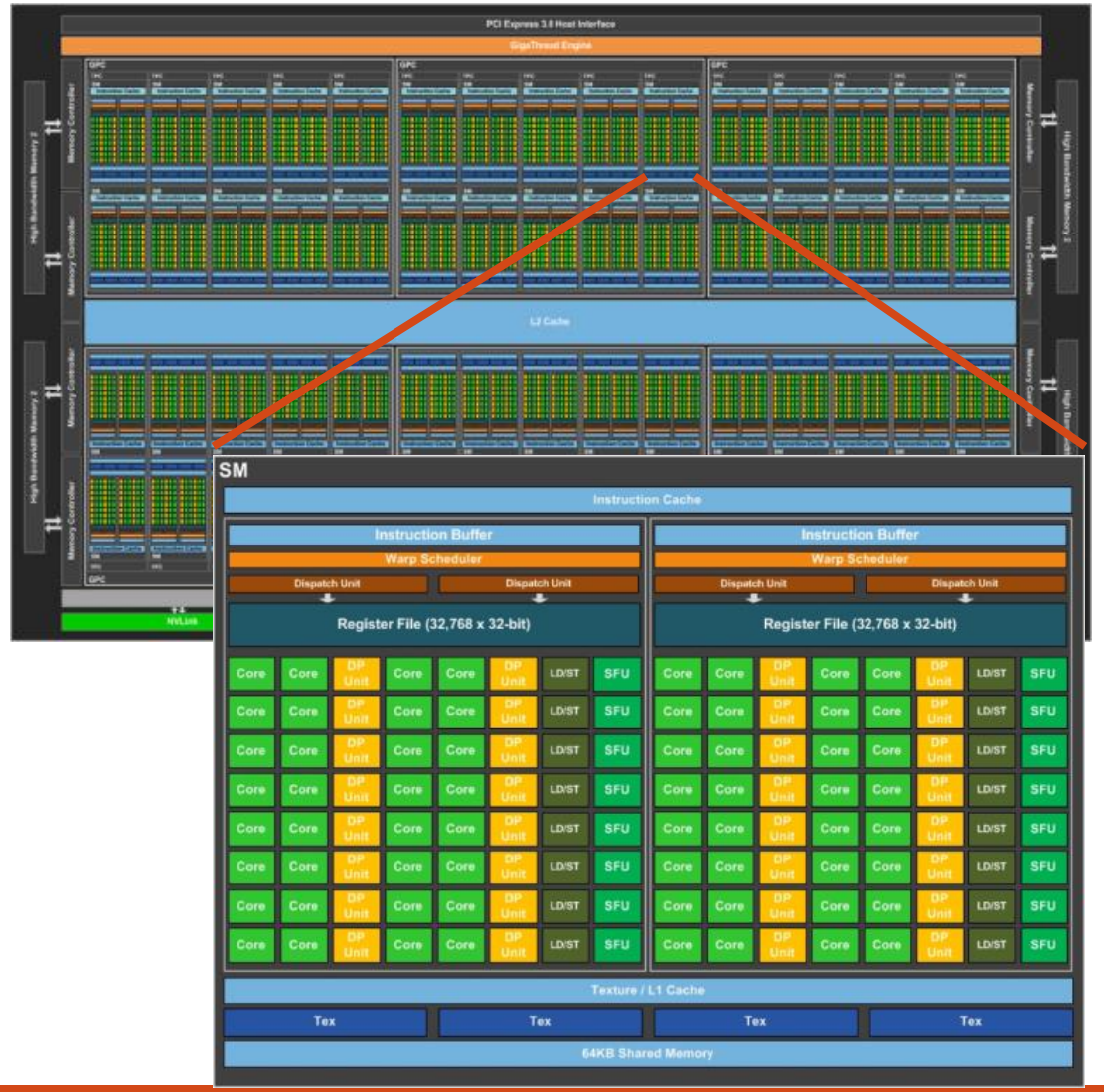
60x streaming multiprocessors (SMs)

64 “CUDA cores” each

➔ 3840 total “cores”

732 GB/s mem bw using 3D stacking technology

256KB registers / SM

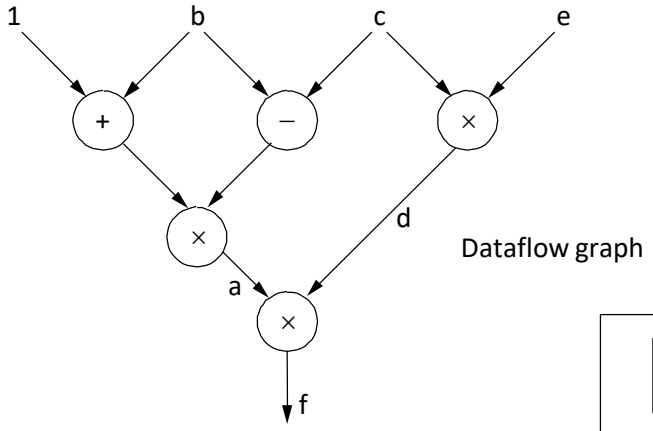


Dataflow Architectures

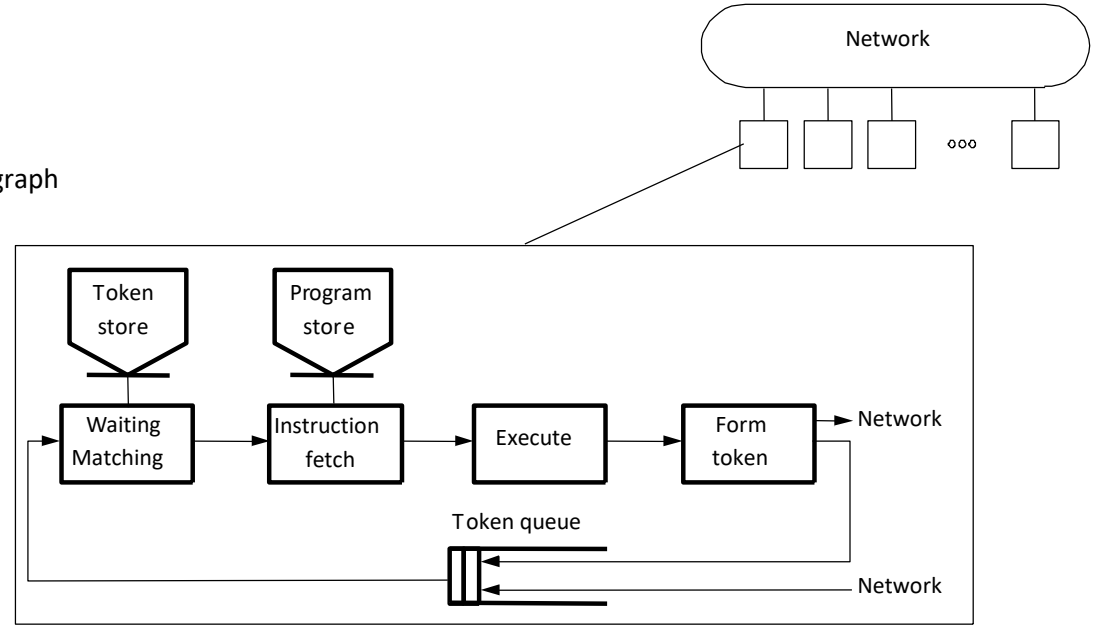
Represent computation as a graph of essential dependences

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

$$a = (b + 1) \times (b - c)$$
$$d = c \times e$$
$$f = a \times d$$



Dataflow graph



Evolution and Convergence

Key characteristics:

- Ability to name operations, synchronization, dynamic scheduling

Problems:

- **Operations have locality**, should be grouped together
- Exposes too much parallelism (!?!)
- Handling complex data structures like arrays
- Complexity of matching store and memory units (tons of power burned in token store)

Converged to use conventional processors and memory

- Support for large, dynamic set of threads to map to processors
- Typically shared address space as well
- But separation of programming model from hardware (like data parallel)

Lasting Contributions of Dataflow

Out-of-order execution (more on this later)

- Most von Neumann processors today contain a dataflow engine inside
- OOO considers dataflow within a bounded region of a program
- Limiting parallelism mitigates dataflow's problems
- …But also sacrifices the extreme parallelism available in dataflow

Many other research proposals to exploit dataflow

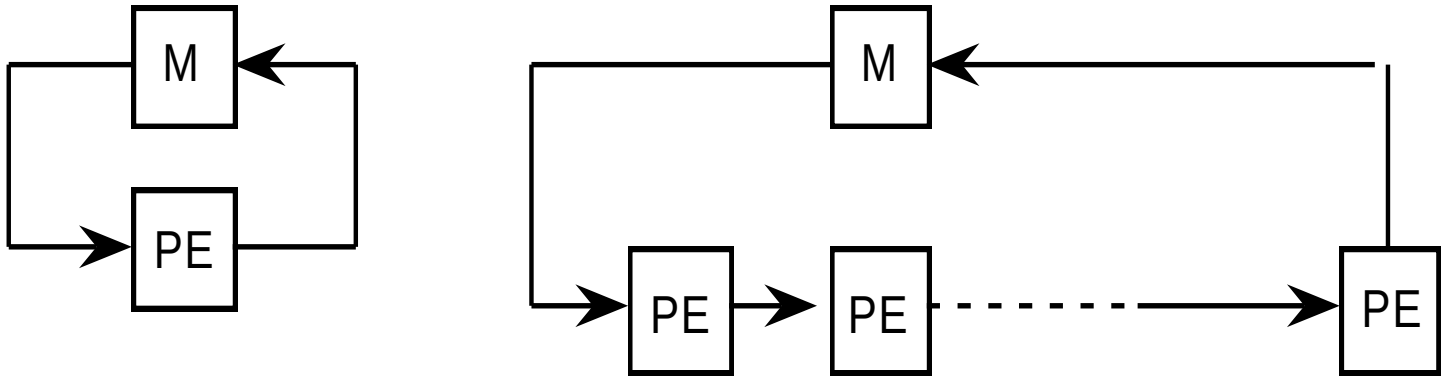
- Dataflow at multiple granularities
- Dataflow amongst many von Neumann tasks

Beyond architecture, many lasting ideas:

- Integration of communication with thread (handler) generation
- Tightly integrated communication and fine-grained synchronization
- Remained useful concept for software (compilers etc.)

Systolic/Spatial Architectures

- Replace single processor with array of regular processing elements
- Orchestrate data flow for high throughput with less memory access



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

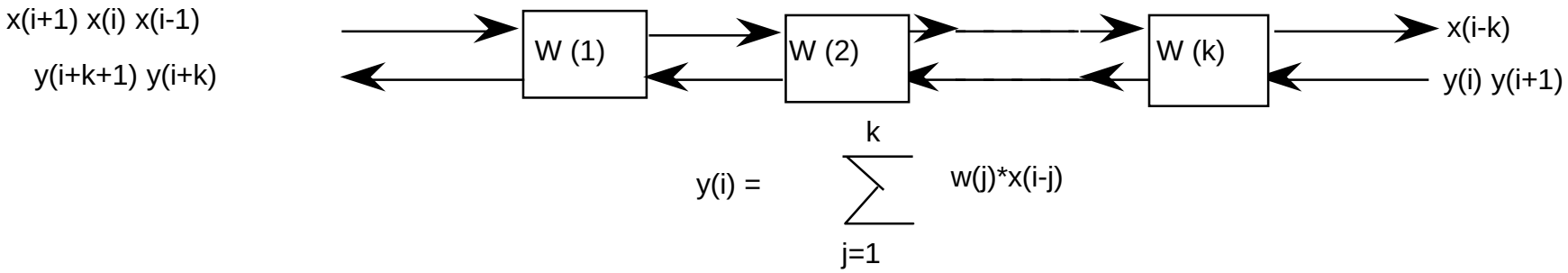
Different from SIMD: each PE may do something different

Initial motivation: VLSI enables inexpensive special-purpose chips

Represent algorithms directly by chips connected in regular pattern

Systolic Arrays (Cont)

Example: Systolic array for 1-D convolution



- **Practical realizations (e.g. iWARP from CMU) use quite general processors**
  - Enable variety of algorithms on same hardware
- **But dedicated interconnect channels**
  - Data transfer directly from register to register across channel
- **Specialized, and same problems as SIMD**
  - General purpose systems work well for same algorithms (locality etc.)
- **Recently, revived interest in neural network accelerators, processing-in-memory**

MIT RAW Processor (‘02)

Tiled mesh multicore

Very simple cores

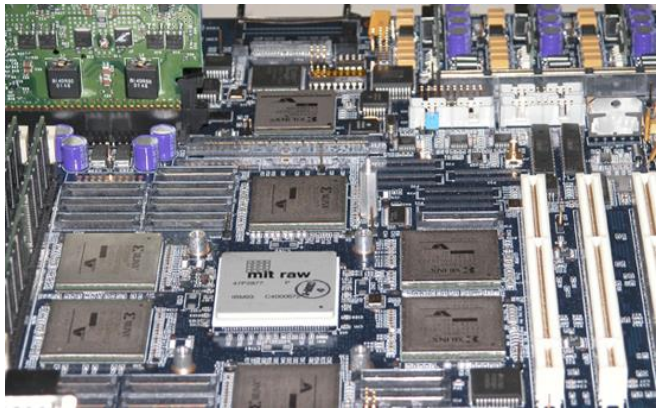
No hardware coherence

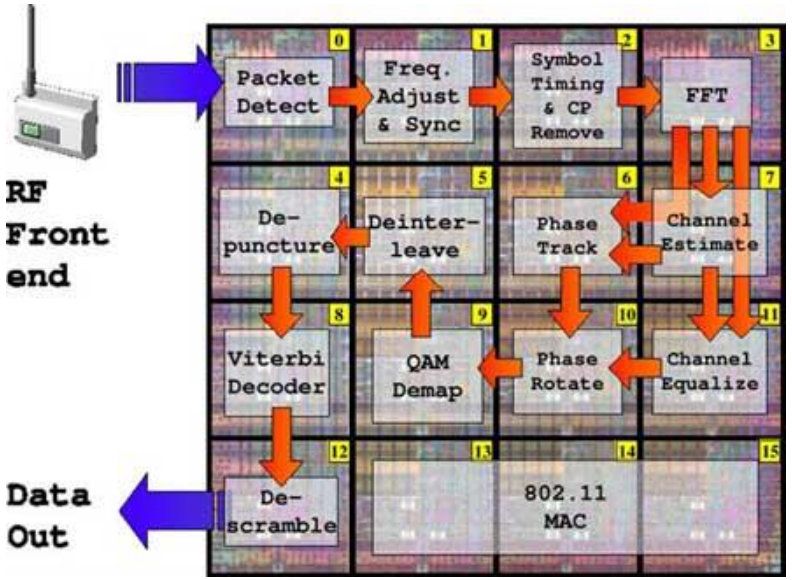
Register-to-register messaging

Programmable routers

Programs split across cores

Looks like a systolic array!





Fundamental Issues in Parallel Architecture

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?

Understand at programming model first, since that sets requirements

Other design & implementation issues:

Node Granularity: How to split between processors and memory?

Communication Cost: Latency, bandwidth, overhead, occupancy...

Replication: How are data replicated to reduce communication?

Sequential Programming Model

Naming:

Can name any variable in virtual address space

- Hardware/compilers translate to physical addresses

Operations:

Loads and Stores

Ordering:

Sequential program order

- E.g., read-after-write (RAW) or “true” dependences
- Also: write-after-read (WAR) or “anti-”dependences, write-after-write (WAW) or “output” dependences, and “control” dependences

Sequential Performance

Compilers and hardware violate other orders without getting caught

- Compiler: reordering and register allocation
- Hardware: out of order, speculation

Modern, OOO cores eliminate all but true dependences

Node granularity: Large (maximize sequential perf)

Communication costs: Accessing memory (or L2 cache)

Replication: Transparent replication in caches (in hw)

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: Sequential Consistency (SC)

- 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

Today, machines implement weaker consistency models than SC to relax ordering restrictions and increase parallelism (discussed later)

Add'l Ordering via Synchronization

Mutual exclusion (locks)

- Intuition: Room that only one person can enter at a time
- acquire()/release() – only one thread at a time can acquire
- Ensure certain operations on certain data can be performed by only one thread at a time
- No ordering guarantees among threads
- Example: transferring money between bank accounts

Many other ways to synchronize

- Events ("wait until X is set by someone else")
- Barriers ("wait until X threads reach here")
- Read-write locks (allow many readers, but only one writer)
- Semaphores (allow up to X concurrent threads)
- Many more…

Message Passing Programming Model

Naming:

Local memory

Remote processes (+ tag)

- No global addressing!

Operations:

Load/store locally

Send/receive remote

Ordering:

Program order locally

Messages order sender-receiver remote

Mutual exclusion for free (in pure message passing)

Comparison of Parallel Arch Schools

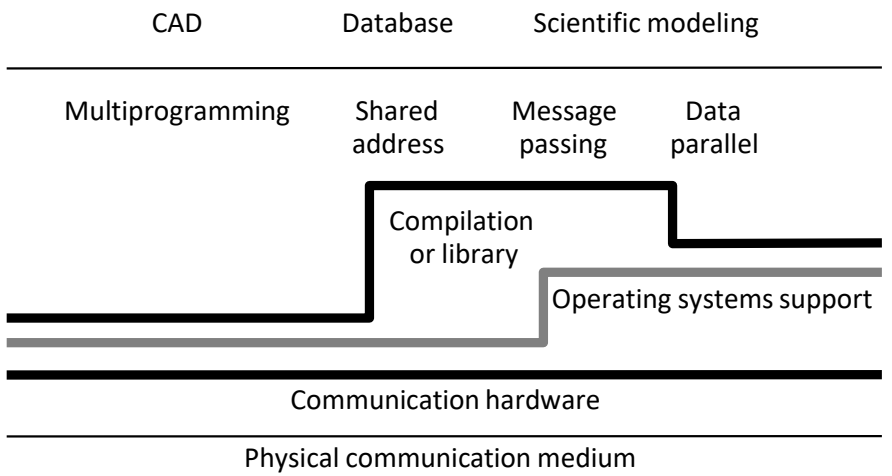
	Naming	Operations	Ordering	Processing Granularity
Sequential	Anything	Load/store	Program	Large (ILP)
Shared memory	Anything	Load/store	SC + synch	Large-to-medium
Message passing	Remote processes	Send/receive	Messages	Large-to-medium
Dataflow	Operations	Send token	Tokens	Small
Data parallel	Anything	Simple compute	Bulk-parallel	Tiny
Systolic/ spatial	Local mem + input	Complex compute	Local messages	Small

Design Issues Apply at All Layers

Programming model's position provides constraints/goals for system

In fact, each interface between layers supports or takes a position on:

- Naming model
- Set of operations on names
- Ordering model
- Replication
- Communication performance



Any set of positions can be mapped to any other by software

Let's see issues across layers:

- How lower layers can support contracts of programming models
- Performance issues

Same Model, Diff Implementation

Example: Shared address space in programming model

Supported in hardware

- Hardware handles transfers and bookkeeping (eg, consistency)

Supported in software

- Can provide SAS through OS using virtual memory
  - page mappings only for data that are local
  - remote data accesses incur page faults; brought in via page fault handlers
  - same programming model, different hardware reqs and cost model
- Or through compilers or runtime
  - program tags shared objects
  - compiler instruments accesses to shared objects
  - can be more efficient than OS (eg, app knows how big each object is, sharing pattern, etc)

Same Model, Diff Implementation

Example: Implementing Message Passing

Support at hardware interface

- Limited flexibility for different apps (matching tags, allocating buffers)

Support at system/user interface in software (most common)

- Hardware interface provides basic data transport (well suited)
- Choices at user/system interface:
  - OS only…but syscalls are expensive
  - OS sets up protection, userspace software handles buffers directly
- Or, if hardware supports SAS, just use load/stores (allocate buffer in destination local memory)

Need to examine the issues and tradeoffs at every layer

- Frequencies and types of operations, costs

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 this is too simplistic for modern systems

Communication Cost Model

Communication 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 *O(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

Total Comm. Cost = frequency * (Comm. time - overlap)

Replication

Very important for reducing data transfer/communication

Again, depends on naming model

Uniprocessor: caches do it automatically

- Reduce communication with memory

Message Passing naming model at an interface

- A receive replicates, giving a new name; subsequently use new name
- Replication is explicit in software above that interface

SAS naming model at an interface

- A load brings in data transparently, so can replicate transparently
- Hardware caches do this, e.g. in shared physical address space
- OS can do it at page level in shared virtual address space, or objects
- No explicit renaming, many copies for same name: *coherence problem*
  - in uniprocessors, “coherence” of copies is natural in memory hierarchy

Are We Asking Right Questions?

- Programming model:
  - SAS/MP/DP?
  - Is this what should be exposed to the programmer?
- Design issues:
  - Naming/operations/ordering/replication/communication
  - Should any of this be exposed to programmer?

Holy grail is a system that

- **Is easy to program**
- **Yields good performance (and efficiency)**
- **Scales well (adding more resources improves performance)**

Can these be achieved with evolutionary improvements?
Or do we need to rethink computing for parallelism?

Parallel Arch Recap

Exotic designs have contributed much, but given way to convergence

- Push of technology, cost and application performance
- Basic processor-memory architecture is the same
- Key architectural issue is in communication architecture

Fundamental design issues:

- Functional: naming, operations, ordering
- Performance: organization, replication, performance characteristics

Design decisions driven by workload-driven evaluation

- Integral part of the engineering focus

MAKE THE COMMON CASE FAST

Performance Metrics

Parallel Speedup

$$\frac{\text{Time to execute the program with 1 processor}}{\text{Time to execute the program with } N \text{ processors}}$$

Parallel Speedup Example

Computation: $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$

Assume each operation 1 cycle, no communication cost, each op can be executed in a different processor

How fast is this with a single processor?

- Assume no pipelining or concurrent execution of instructions

How fast is this with 3 processors?

Takeaway

To calculate parallel speedup fairly you need to use the **best known algorithm** for each system with N processors

"Scalability! But at what COST?" McSherry et al, HotOS'15

- Large, distributed research systems are outperformed by an off-the-shelf laptop

Utilization, Redundancy, Efficiency

Traditional metrics

- Assume all P processors are tied up for parallel computation

Utilization: How much processing capability is used

- U = (# Operations in parallel version) / (processors x Time)

Redundancy: how much extra work is done

- R = (# of operations in parallel version) / (# operations in best uni-processor algorithm version)

Efficiency

- E = (Time with 1 processor) / (processors x Time with P procs)
- E = U/R

Amdahl's Law: The Limits of Parallelism

Amdahl's law

You plan to visit a friend in Normandy France and must decide whether it is worth it to take the Concorde SST ($3,100) or a 747 ($1,021) from NY to Paris, assuming it will take 4 hours Pgh to NY and 4 hours Paris to Normandy.

	Time NY→Paris
Boeing 747	8.5 hrs
Concorde SST	3.75 hrs

Taking the SST (which is 2.2 times faster) speeds up the overall trip by only a factor of 1.4!

Amdahl's law (cont)

Old program (unenhanced)

Old time: $T = T_1 + T_2$

New program (enhanced)

New time: $T' = T_1' + T_2'$

T_1 = time that can NOT be enhanced.

T_2 = time that can be enhanced.

T_2' = time after the enhancement.

Speedup: $S_{overall} = T / T'$

Amdahl's law (cont)

Two key parameters:

$F_{enhanced} = T_2 / T$ (fraction of original time that can be improved)
$S_{enhanced} = T_2 / T_2'$ (speedup of enhanced part)

Amdahl's Law:

$$S_{overall} = T / T' = \frac{1}{(1 - F_{enhanced}) + \frac{F_{enhanced}}{S_{enhanced}}}$$

Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

Key idea: Amdahl's law quantifies the general notion of diminishing returns. It applies to any activity, not just computer programs.

Amdahl's law (cont)

Trip example: Suppose that for the New York to Paris leg, we now consider the possibility of taking a rocket ship (15 minutes) or a handy rip in the fabric of space-time (0 minutes):

	Time NY→Paris	Total Trip Time	Speedup vs. 747
Boeing 747	8.5 hrs	16.5 hrs	-
Concorde SST	3.75 hrs	11.75 hrs	1.4×
Atlas V	0.25 hrs	8.25 hrs	2×
Rip in space-time	0.0 hrs	8 hrs	2.1×

Amdahl's law (cont)

Corollary: $1 \leq S_{overall} \leq \frac{1}{1-F_{enhanced}}$

$F_{enhanced}$	Max $S_{overall}$	$F_{enhanced}$	Max $S_{overall}$
0.0	1	0.9375	16
0.5	2	0.96875	32
0.75	4	0.984375	64
0.875	8	0.9921875	128

Moral: It is hard to speed up programs! (Parallelism has limits)

Moral++ : It is easy to make premature optimizations.

Caveats of Parallelism (I): Amdahl's Law

Amdahl's Law

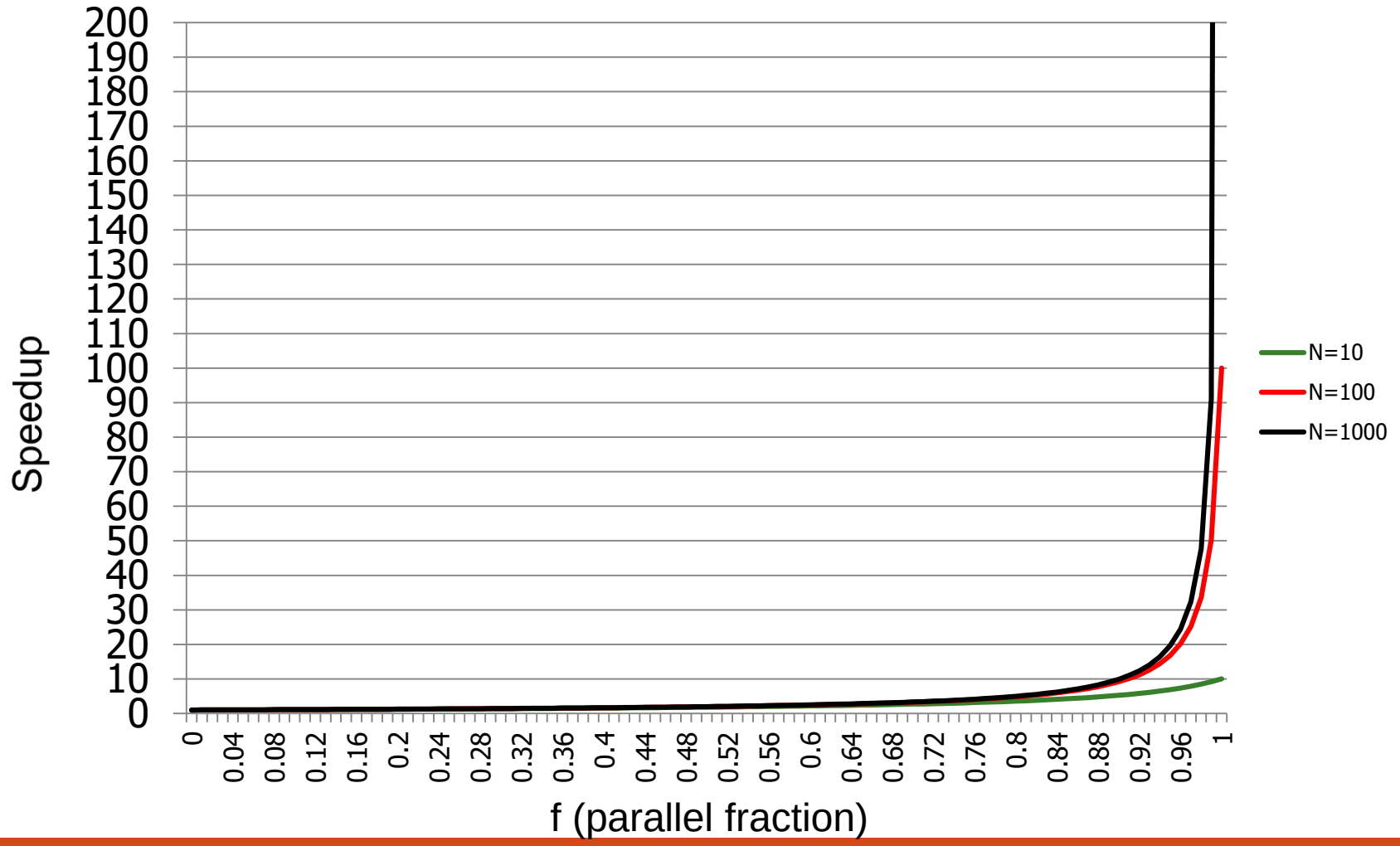
- f: Parallelizable fraction of a program
- P: Number of processors

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{P}}$$

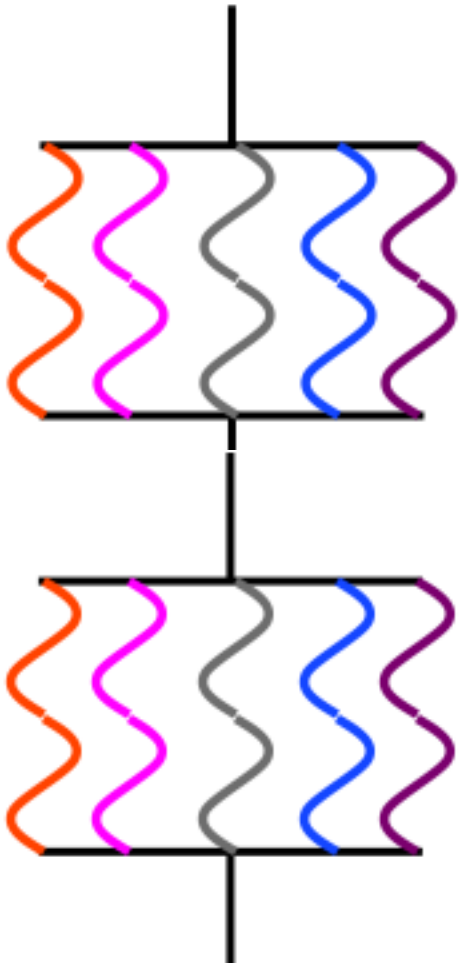
- Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

Maximum speedup limited by serial portion—aka the **Serial Bottleneck**

Sequential Bottleneck



Why the Sequential Bottleneck?



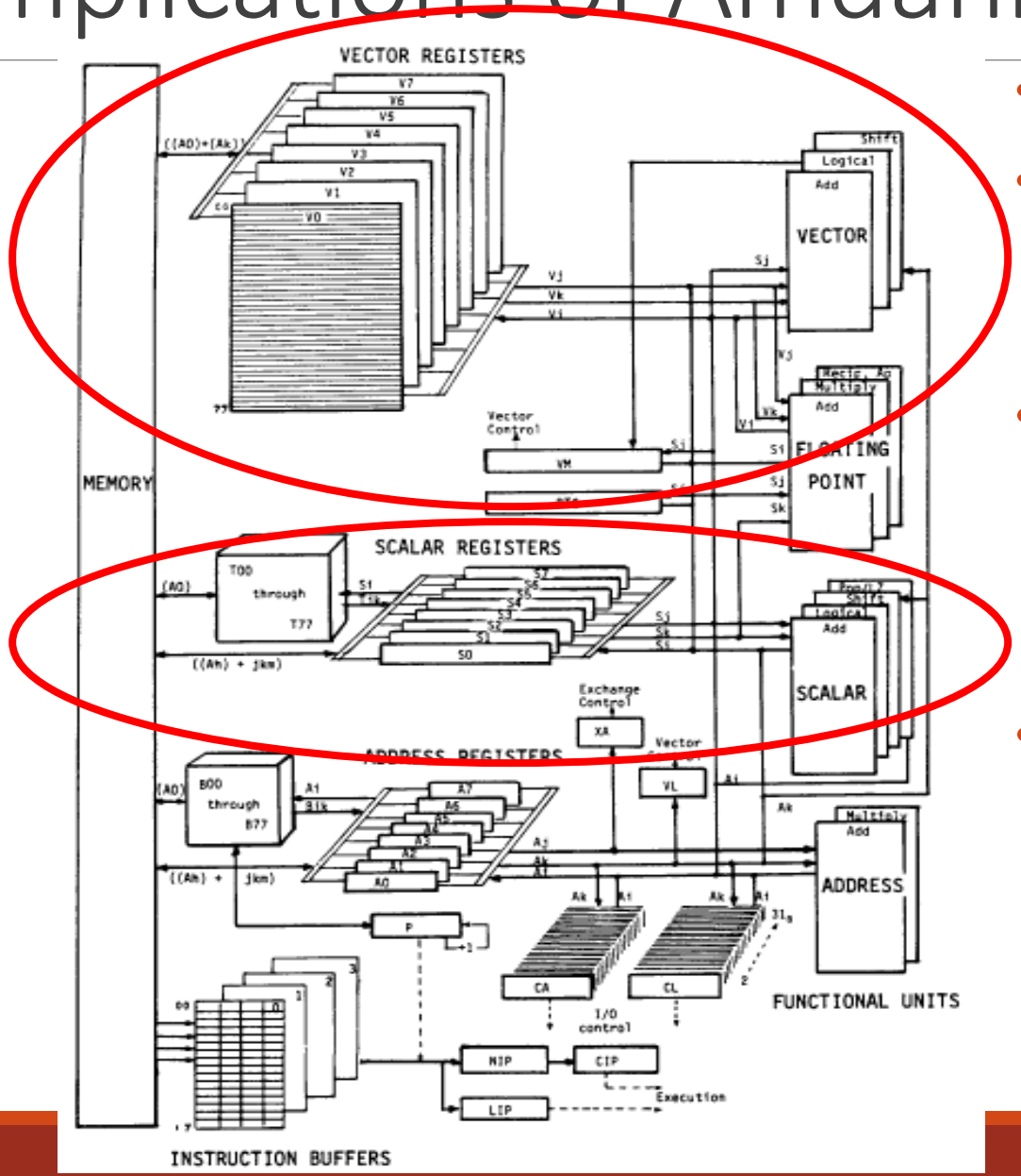
Parallel machines have the sequential bottleneck

Main cause: Non-parallelizable operations on data
(e.g. non-parallelizable loops)

```
for ( i = 0 ; i < N; i++)
      A[i] = (A[i] + A[i-1]) / 2
```

Single thread prepares data and spawns parallel tasks
(usually sequential)

Implications of Amdahl's Law on Design



- CRAY-1
- Russell, "The CRAY-1 computer system," CACM 1978.
- Well known as a fast vector machine
  - 8 64-element vector registers
- The fastest SCALAR machine of its time!
  - Reason: Sequential bottleneck!

Caveats of Parallelism (II)

Amdahl's Law

- f: Parallelizable fraction of a program
- P: Number of processors

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{P}}$$

- Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

Maximum speedup limited by serial portion: Serial bottleneck

Parallel portion is usually not perfectly efficient

- Synchronization overhead (e.g., updates to shared data)
- Load imbalance overhead (imperfect parallelization)
- Resource sharing overhead (contention among N processors)

Bottlenecks in Parallel Portion

Synchronization: Operations manipulating shared data cannot be parallelized

- Locks, mutual exclusion, barrier synchronization
  - Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS 2009.
- Communication: Tasks may need values from each other
  - Causes thread serialization when shared data is contended

Load Imbalance: Parallel tasks may have different lengths

- Due to imperfect parallelization or microarchitectural effects
  - Reduces speedup in parallel portion

Resource Contention: Parallel tasks can share hardware resources, delaying each other

- Replicating all resources (e.g., memory) expensive
  - Additional latency not present when each task runs alone

Difficulty in Parallel Programming

Little difficulty if parallelism is natural

- "Embarrassingly parallel" applications
- Multimedia, physical simulation, graphics
- Large web servers, databases?

Big difficulty is in

- Harder-to-parallelize algorithms
- Getting parallel programs to work correctly
- Optimizing performance in the presence of bottlenecks

Much of **parallel computer architecture** is about

- Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
- Making programmer's job easier in writing correct and high-performance parallel programs