Schools of Parallel Architecture & Amdahl's Law

15-740 FALL'19

NATHAN BECKMANN

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
- Fundamental challenges (communication, scalability) introduced by parallelism

Why parallelism?

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

High-level challenges:

- **Communication**
- N processors often != N× better performance
- Parallel programming is often hard
- Granularity: many “small and slow” cores vs. few “big and fast” cores
- What type of parallelism does app exploit best?
  (In practice, machines exploit parallelism at multiple levels)

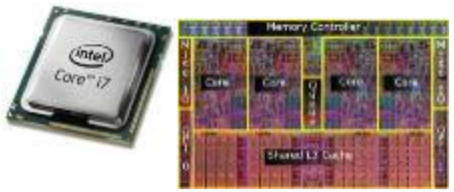
Why study parallel arch & programming?

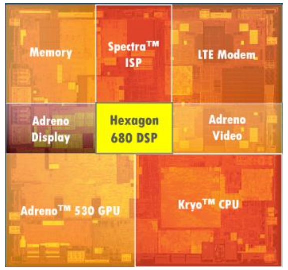
The Answer from ~15 Years Ago:

- 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
  - In practice, was **not enough** of a benefit for most apps ➔ explicit parallelism a niche area

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 $V \propto F \rightarrow P \propto CF^3$)
  - Implicit parallelism is not increasing either!
  - ➔ Improving performance on sequential code is 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: Why parallelism?

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…

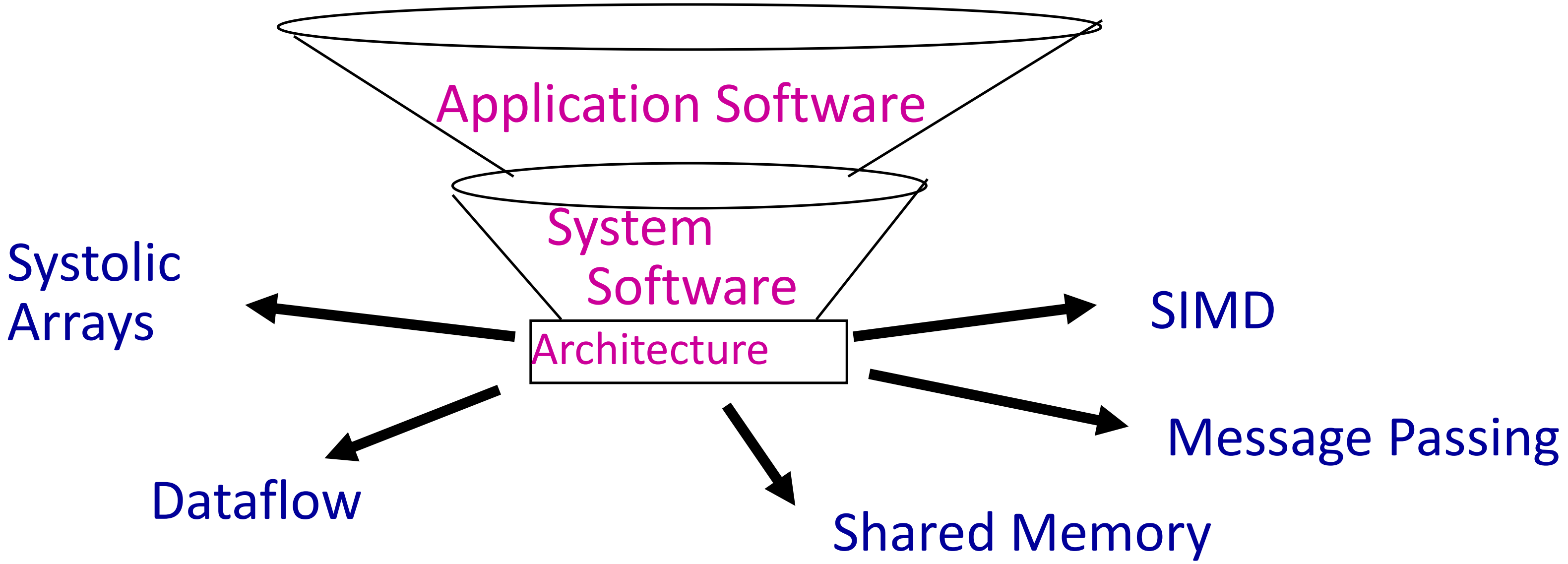
- …technology defied expectations (until ~15y ago)
- …parallelism is more efficient in theory, but getting good parallel programs in practice is hard (architecture doesn't exist in a vacuum; see also: scratchpads vs caches)

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

History: Different schools of parallelism

Historically, parallel architectures closely tied to programming models

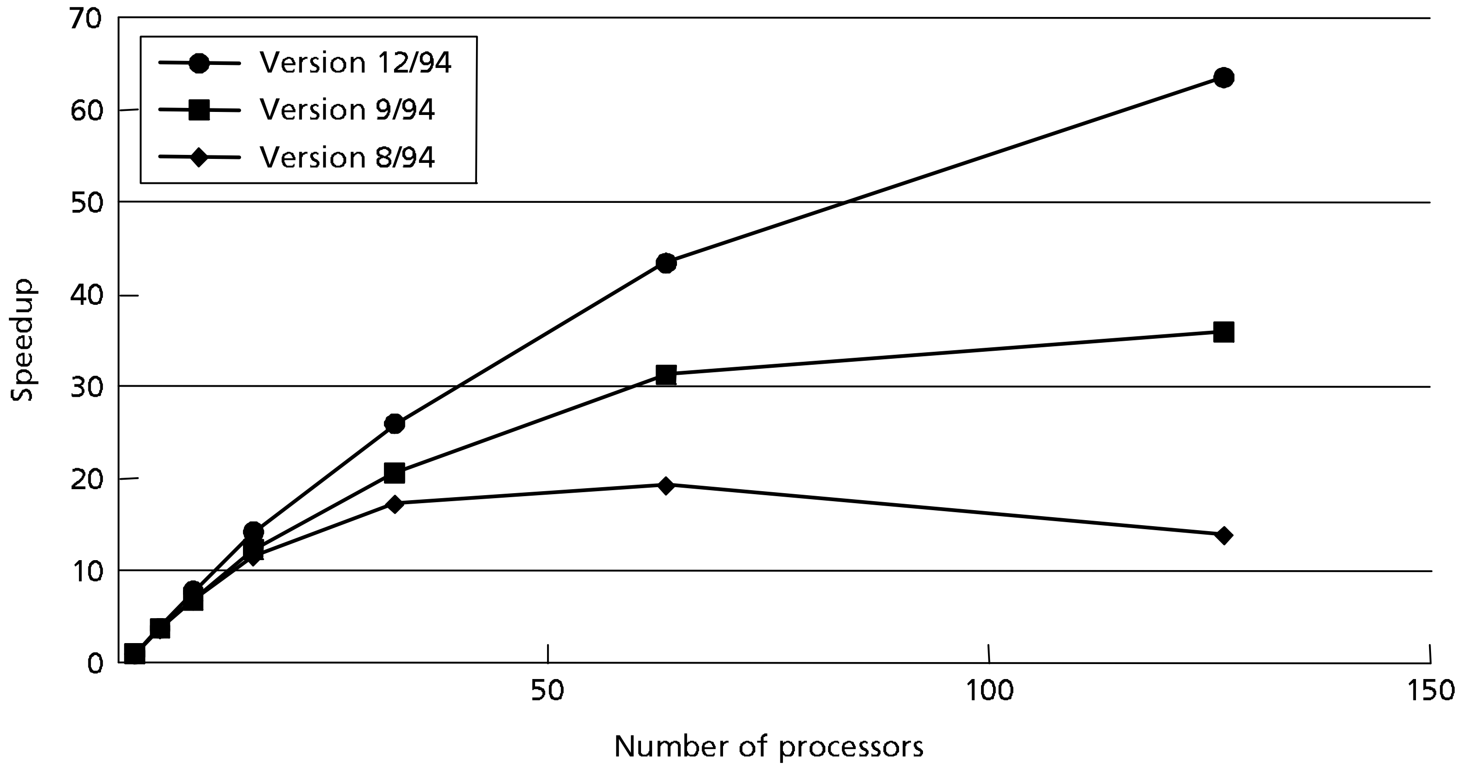
Divergent architectures, with no predictable pattern of growth.



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** (vector); ported to **Intel Paragon** (message-passing)

(slide credit: Culler'99)

Schools of parallelism via an example

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!*

Instruction-level parallelism (ILP)

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

```
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

Instruction-level parallelism (ILP)

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

```
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
```

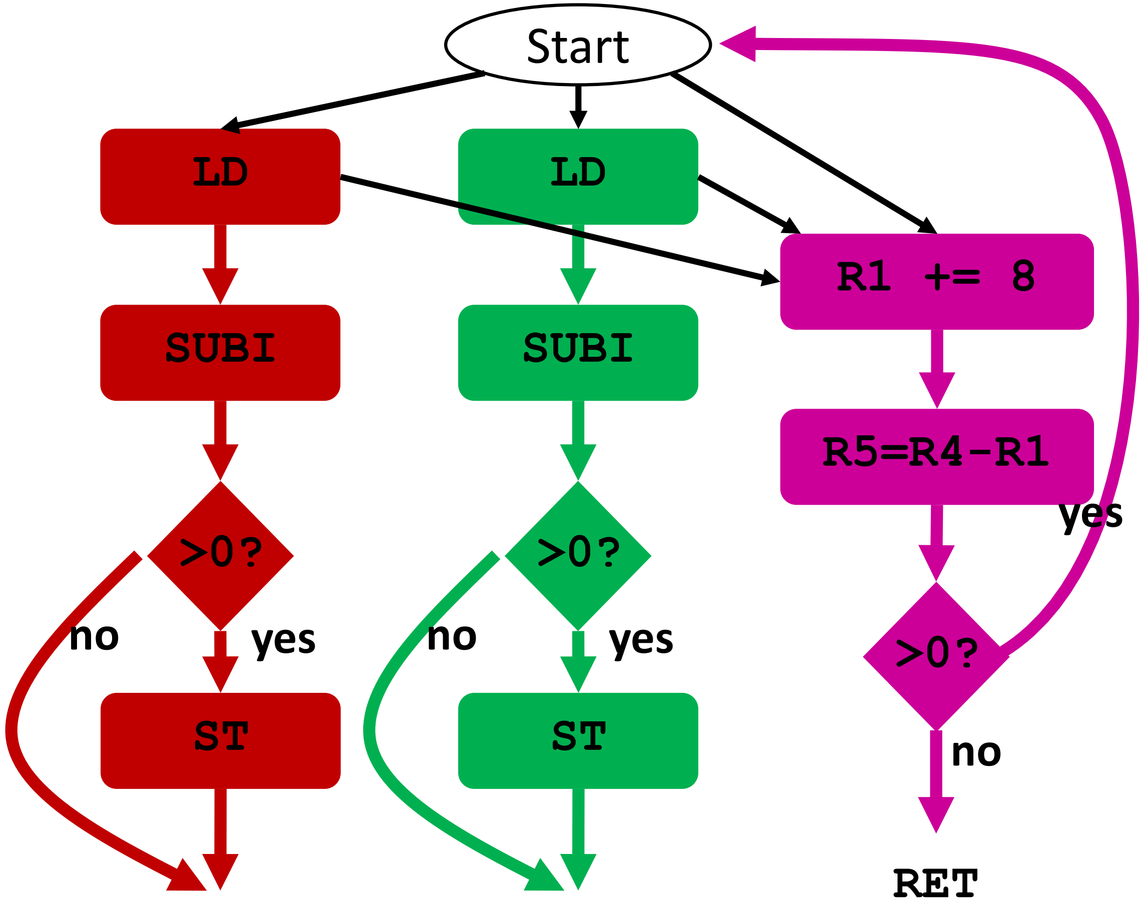
```
void decrement_all(
    int *array,
    int size) {
  for (int i = 0;
    i < size;
    i++) {
     int x = array[i] - 1;
     if (x > 0) {
       array[i] = x;
     }
  }
}
```

Loop unrolled x2

Instruction-level parallelism (ILP)

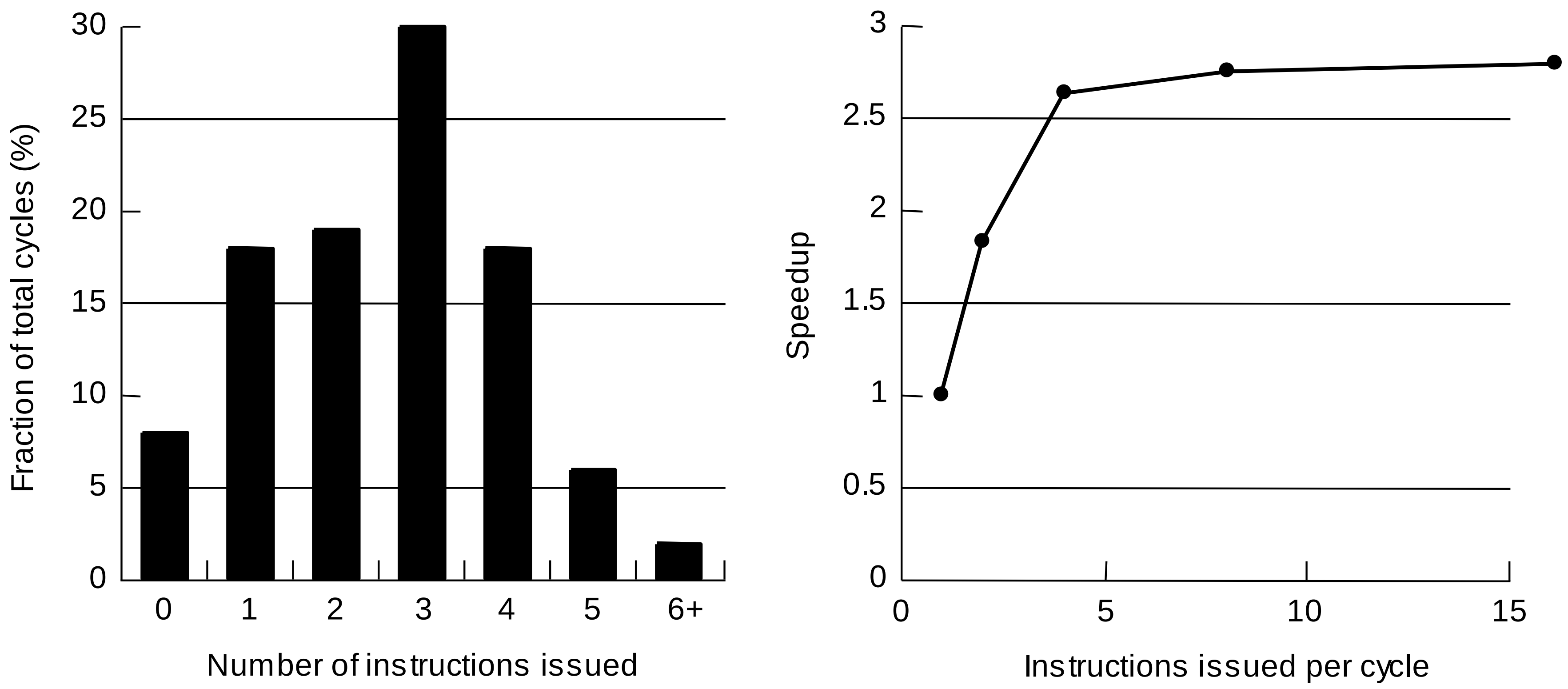
- Different instructions within a stream can be executed in parallel
- Pipelining, out-of-order execution, speculative execution, VLIW

```
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
```



Limits of conventional ILP

Instruction-level parallelism peaks @ ~4 ins / cycle



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

Dataflow

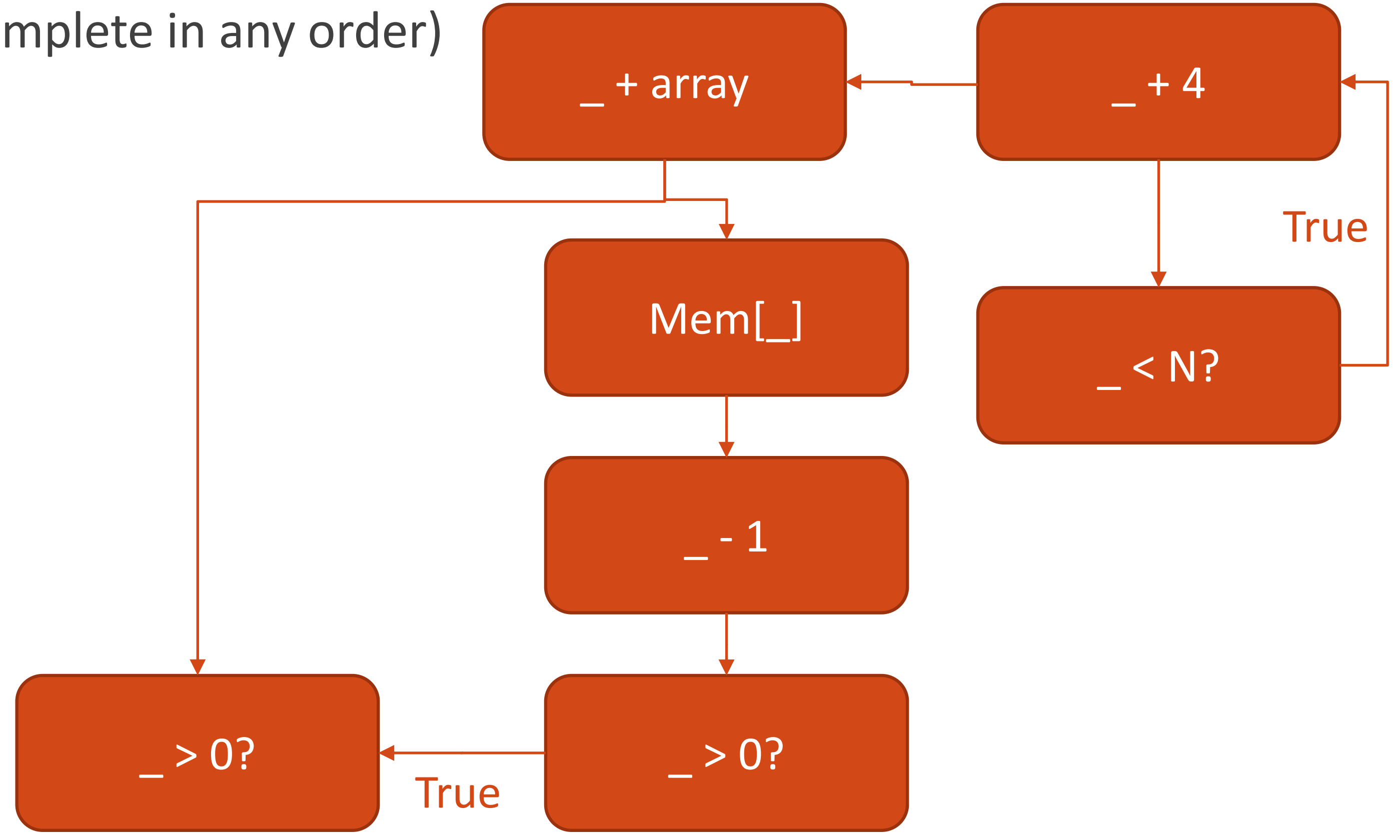
Operations communicate directly to dependent ops

No program counter! (Iterations may complete in any order)

Looks similar to ILP – not a coincidence

```
ITER:     _ + 4 → CHECK / LOOP
CHECK:    _ < N? → ITER

LOOP:     _ + array → LD / ST[0]
LD:       Mem[_] → SUB
SUB:      _ - 1 → CMPZERO
CMPZERO:  _ > 0? → ST[1]
ST:       Mem[_] := _
```

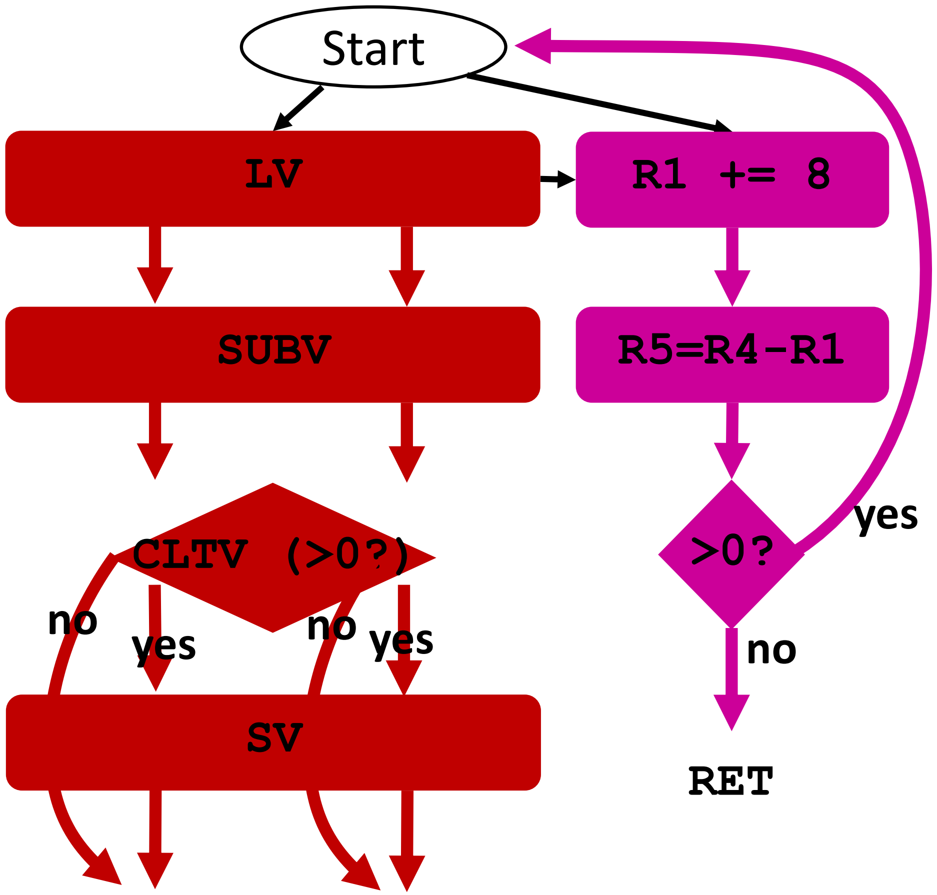


Data parallel

- 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
```

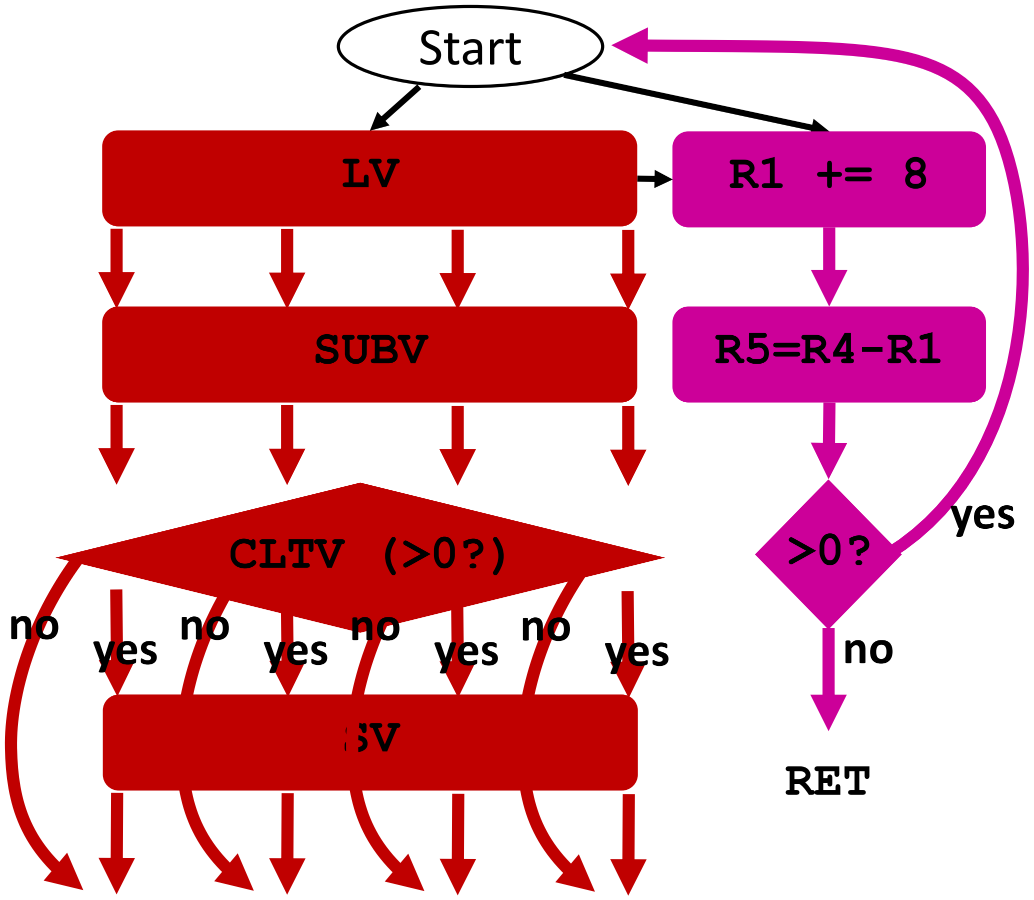


Data parallel

- 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
```

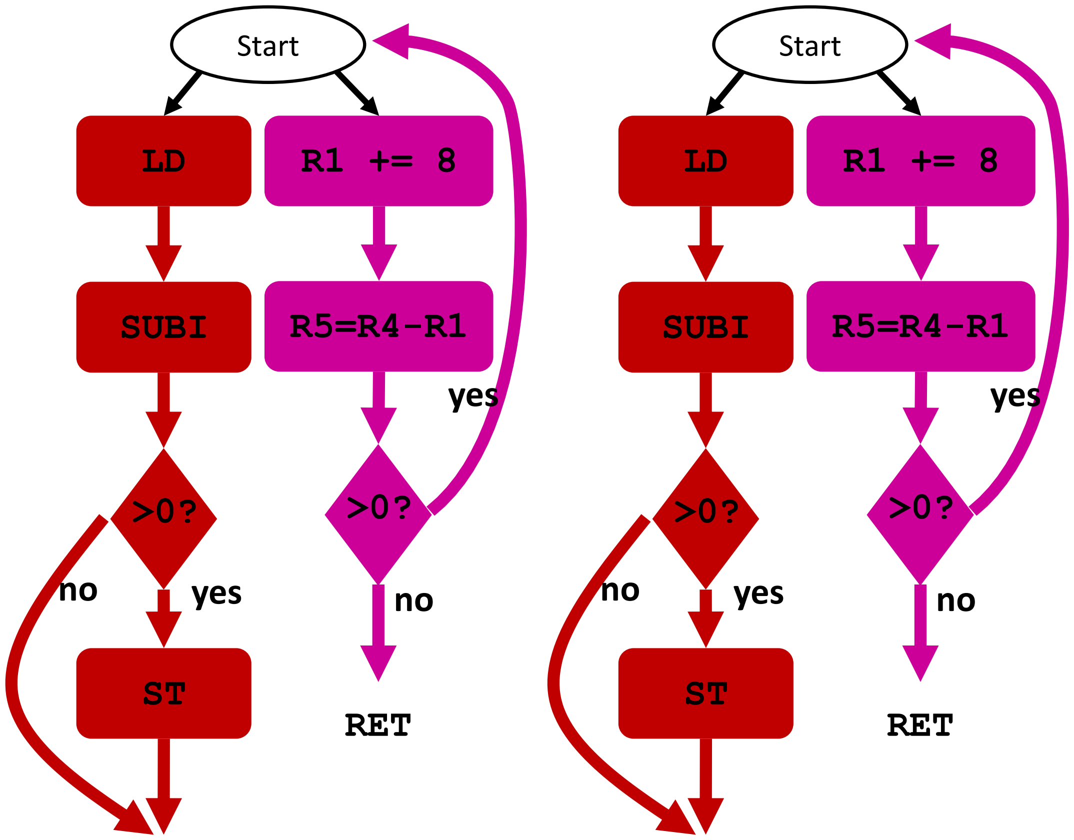


Task/Thread parallelism

- 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
```



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 programming models

Programming Model

What programmer uses in coding applications

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

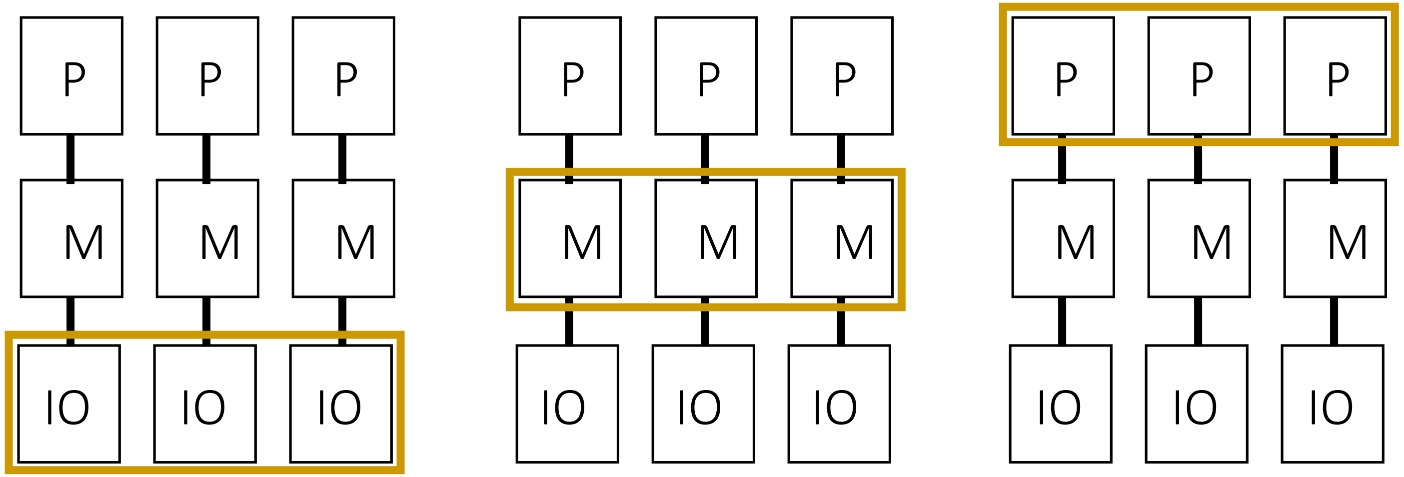
Examples:

- 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

Programming model can be realized in hardware, OS software, or user software

- Lots of debate about where to implement what functionality (hw vs sw)

Where Communication Happens



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

History: Arch vs Programming Models

Historically: architecture == programming model

- 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 (**compatibility** again)
  - Except processes run on different processors
  - Good throughput on multi-programmed workloads

Naturally provided on wide range of platforms

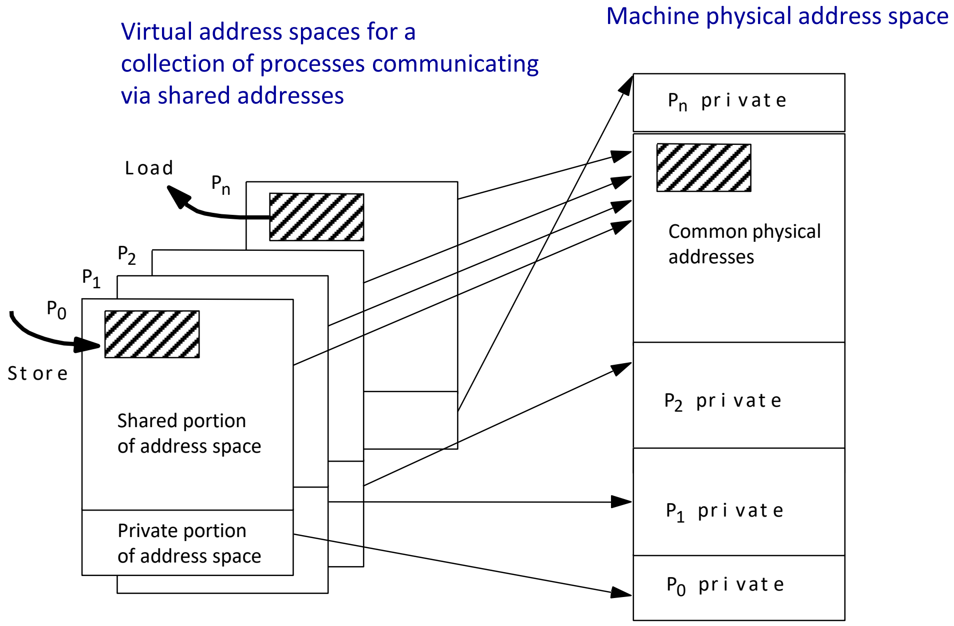
- History dates at least to precursors of mainframes in early 60s
- Wide range of scale: few to hundreds of processors

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

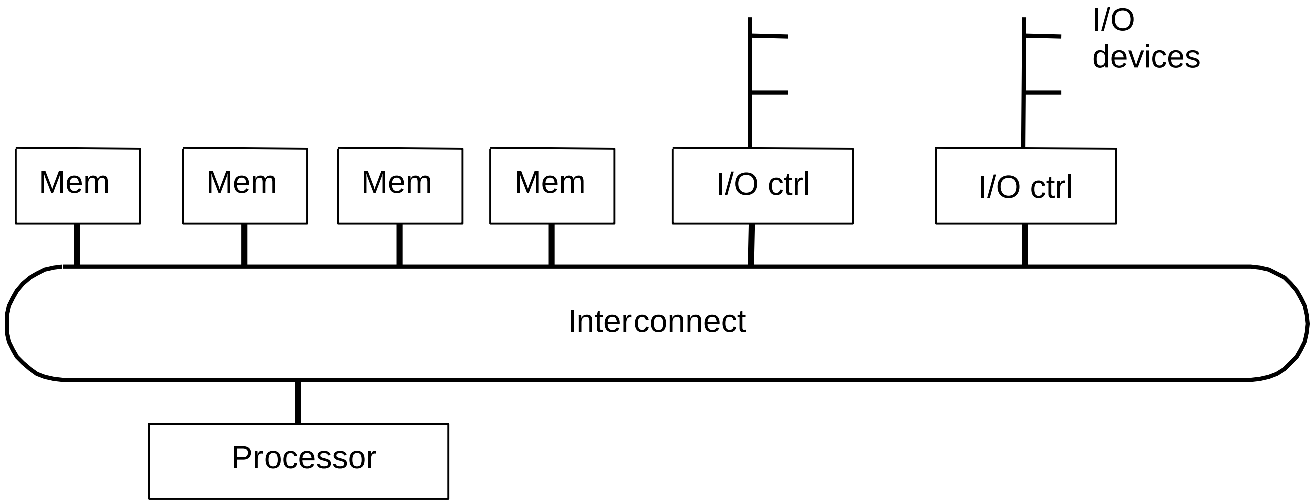


- Writes to shared address visible to other threads, processes
- OS uses shared memory to coordinate processes

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

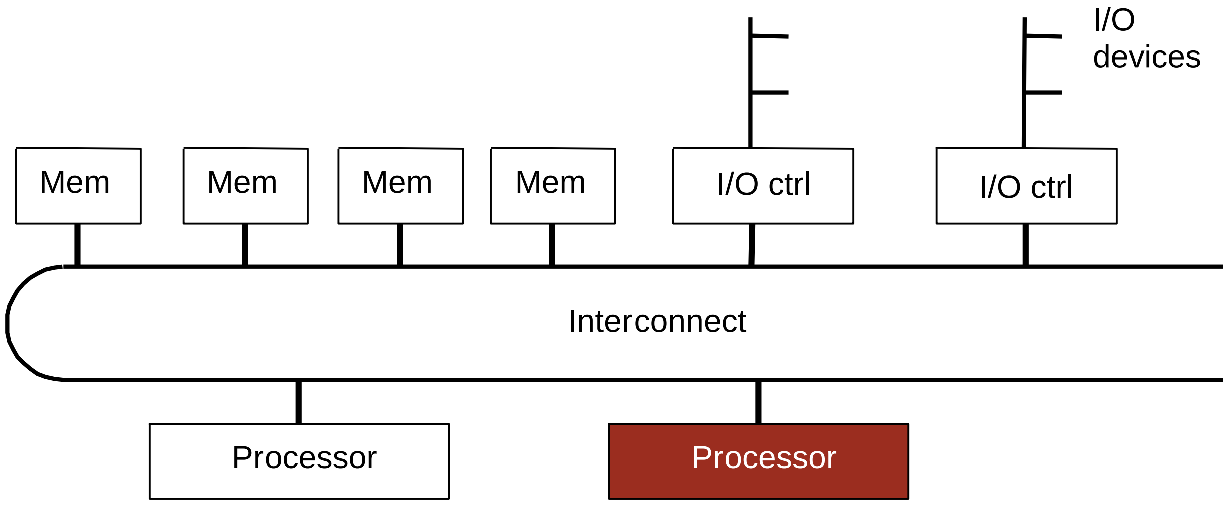


Memory capacity increased by adding modules, I/O by controllers

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



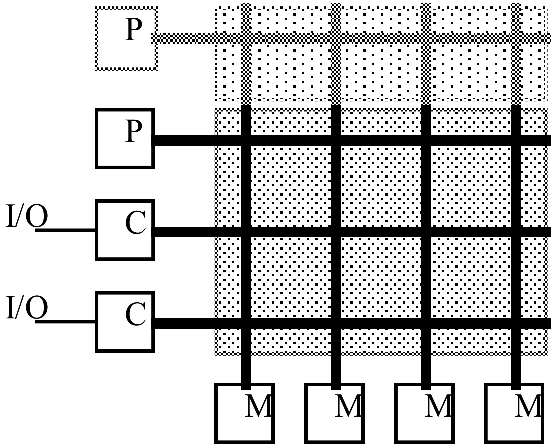
Memory capacity increased by adding modules, I/O by controllers

➔ **Add processors for processing!**

SAS History

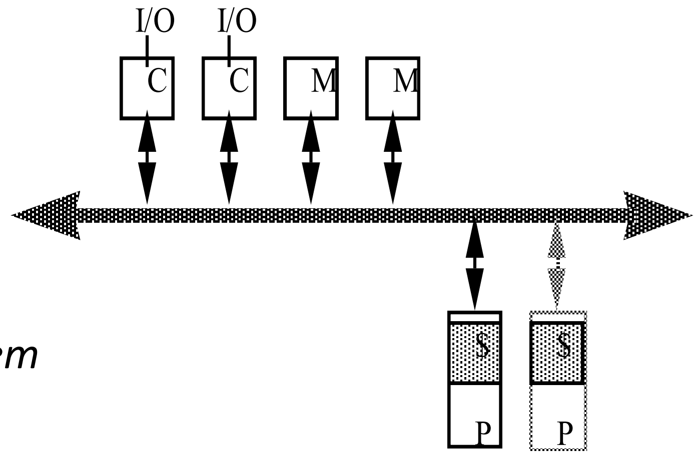
"Mainframe" approach:

- 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

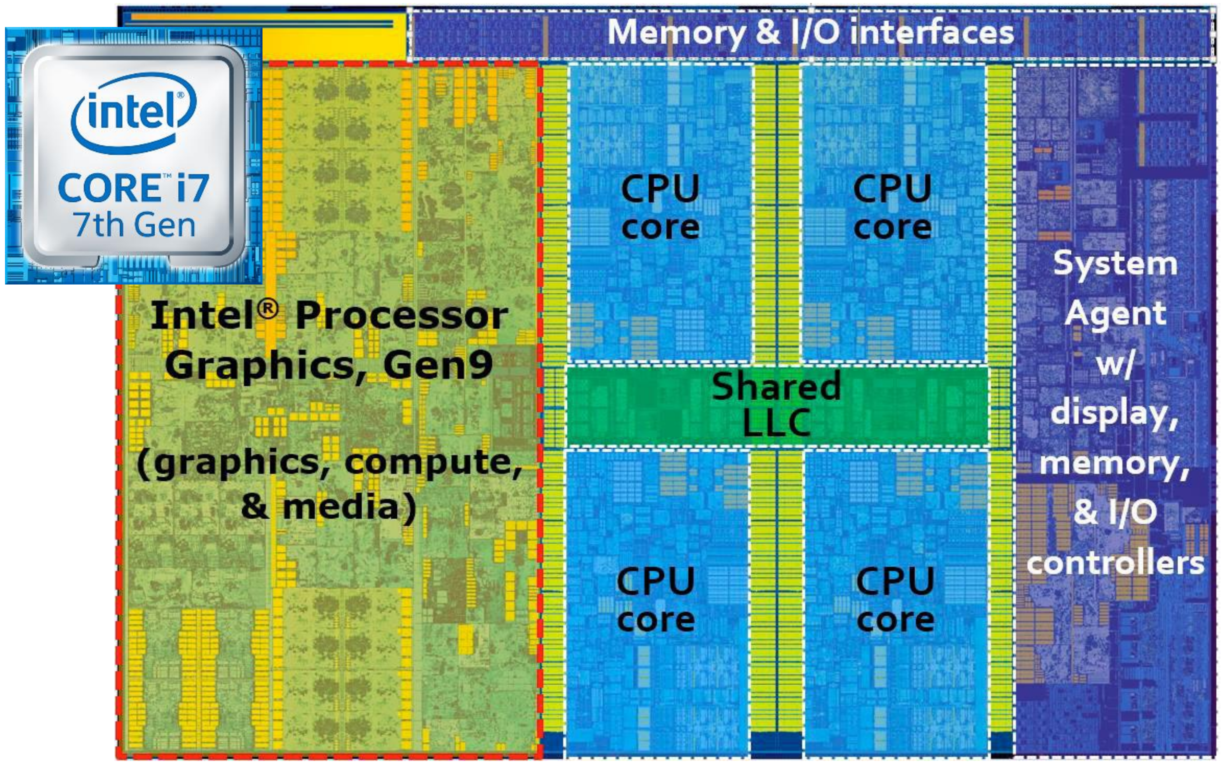


"Minicomputer" approach:

- 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*



Recent ('17) x86 Example

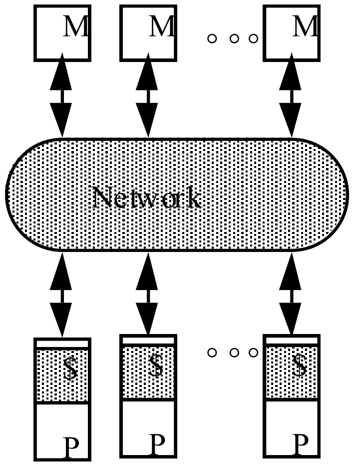


Intel's Core i7 7th generation

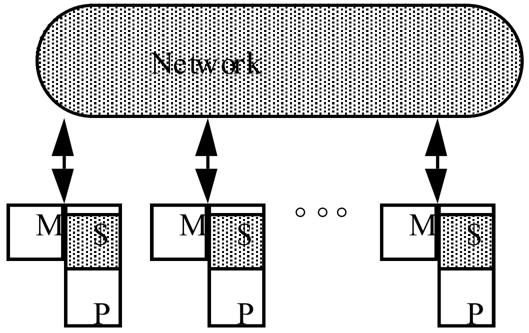
- Highly integrated, commodity systems
- On-chip: low-latency, high-bandwidth communication via shared cache
- Current scale = ~4 processors (up to 12 on some models, more on server parts)

Scaling Up

- Problem is interconnect: cost (crossbar) or bandwidth (bus)
- “Dance-hall” topologies: Latencies to memory uniform, but **uniformly large**
  - “Resource disaggregation” is the modern incarnation of this idea
- Distributed memory or non-uniform memory access (NUMA)
  - Construct shared address space out of simple message transactions across a general-purpose network
  - Cache nonlocal data to reduce data movement? ➔ Must decide coherence story (hardware vs software)



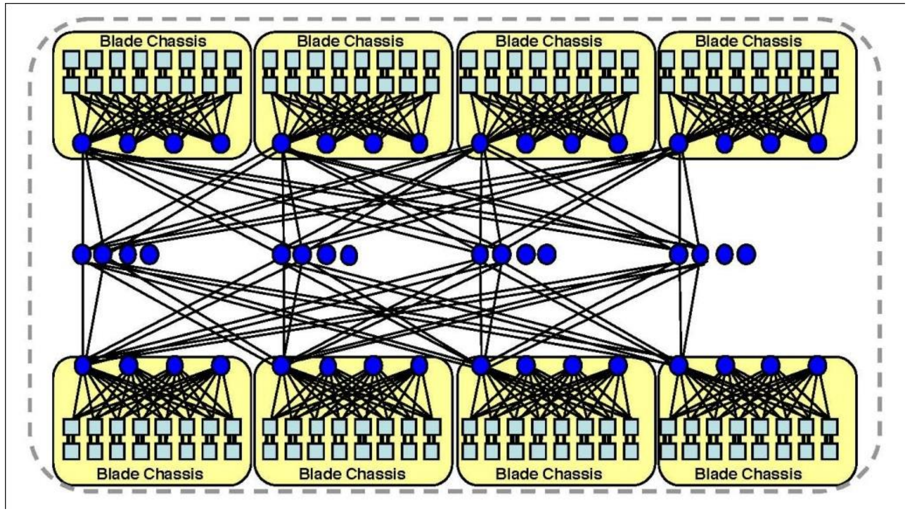
“Dance hall”



Distributed memory

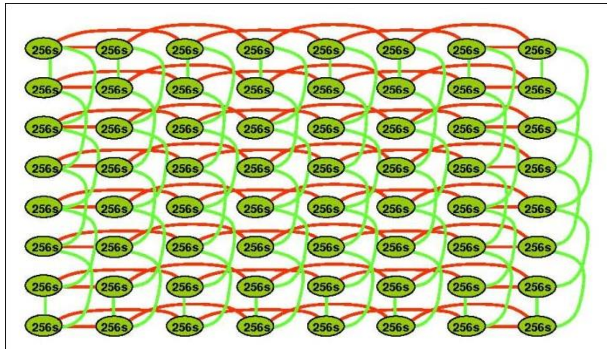
Example: SGI Altix UV 1000 (‘09)

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)

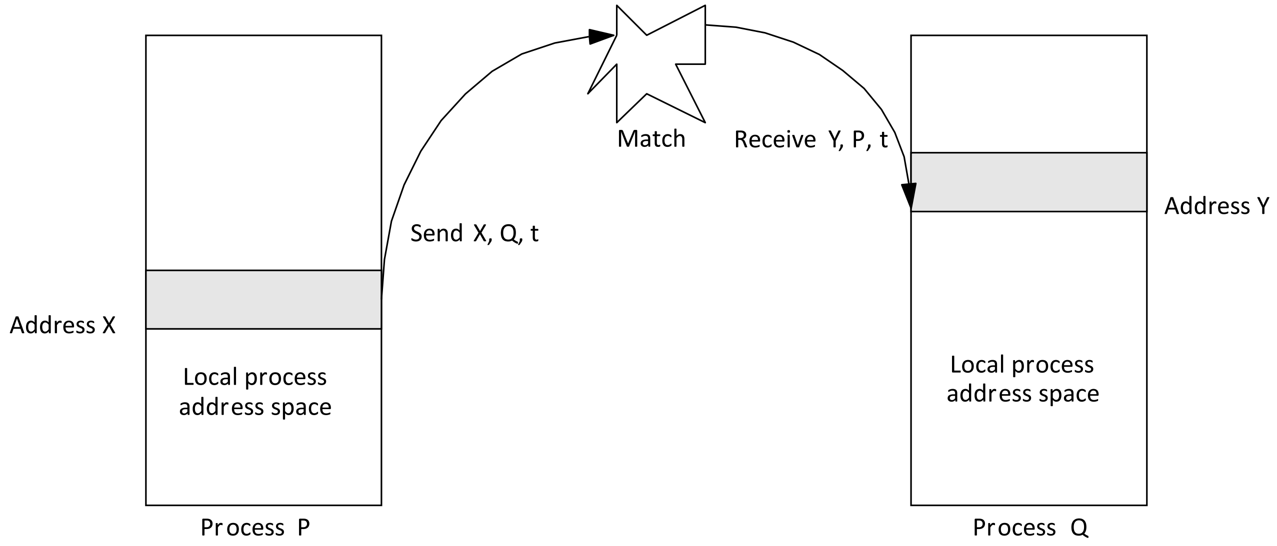
High-level block diagram similar to distributed-mem SAS

- 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

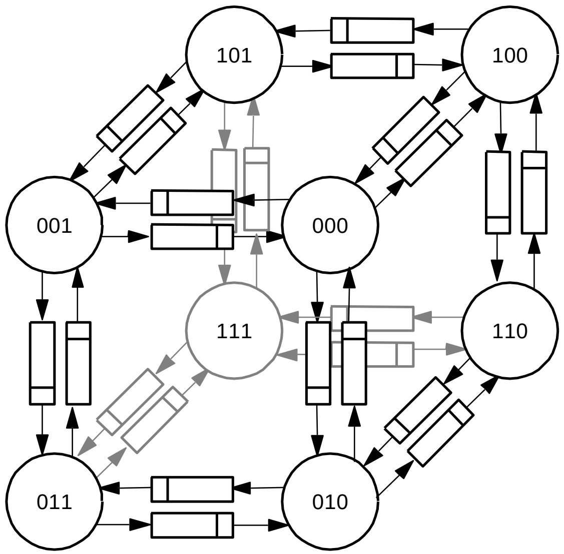
History of Message Passing

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: IBM Blue Gene/Q ('11)

81,920 cores / 5,120 nodes

Each 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)

Towards Architectural Convergence

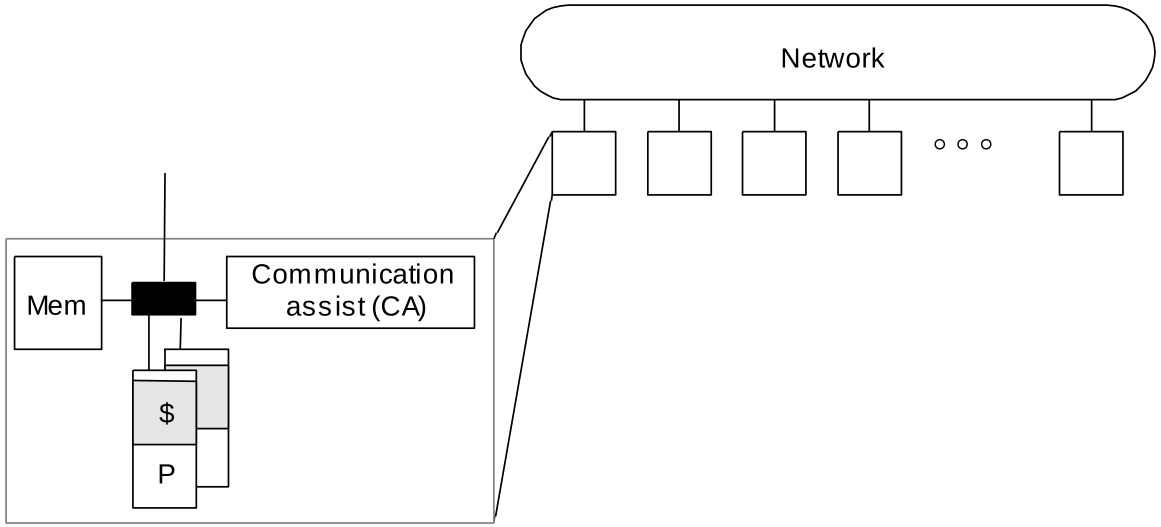
Evolution and role of software have blurred boundary
- Send/recv supported on SAS machines via buffers
- Can construct global address space on MP using hashing
- Page-based (or finer-grained) shared virtual memory

Hardware converging too
- Tightly integrated network interface (in hardware)
- 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...**

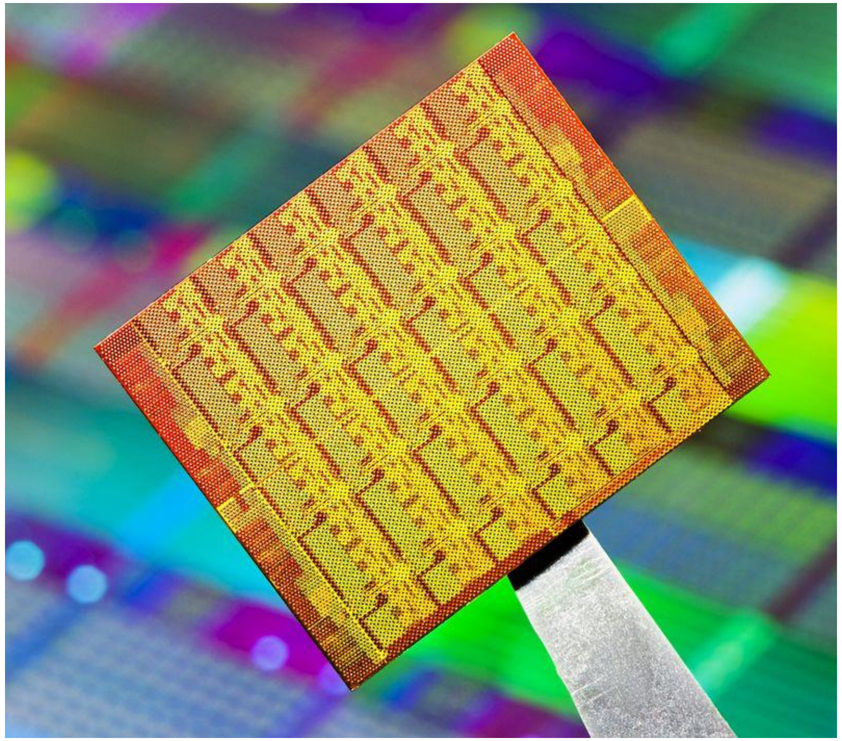
Intel Single-chip Cloud Computer (‘09)

48 cores

2D mesh network

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

4 DDR3 controllers



Shared memory + message passing hardware

No hardware coherence

Coherence available through software library

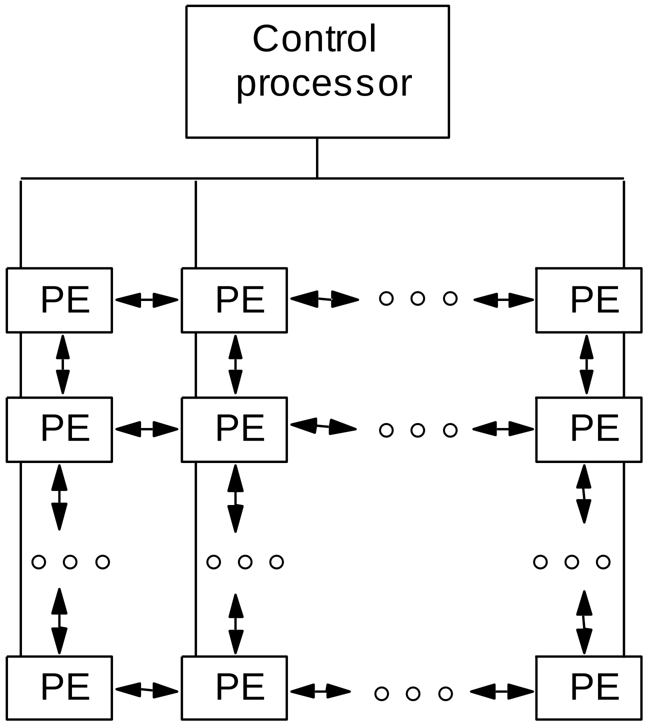
Data-Parallel Systems

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, dumb, fast processors with little memory each
- Attached to a control processor that issues instructions
- Specialized communication for cheap global synchronization
- Each processor can be implemented in fast, specialized circuits



History of data-parallel arch

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 out-of-order (somewhat) weakened this argument
- Simple, regular applications have good locality, can do well anyway
- ➔ MIMD machines also effective for data parallelism and more general
- Loss of generality due to hardwiring data parallelism

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

Data-parallelism is key to most accelerator designs

Example: Nvidia Pascal 100 (‘16)

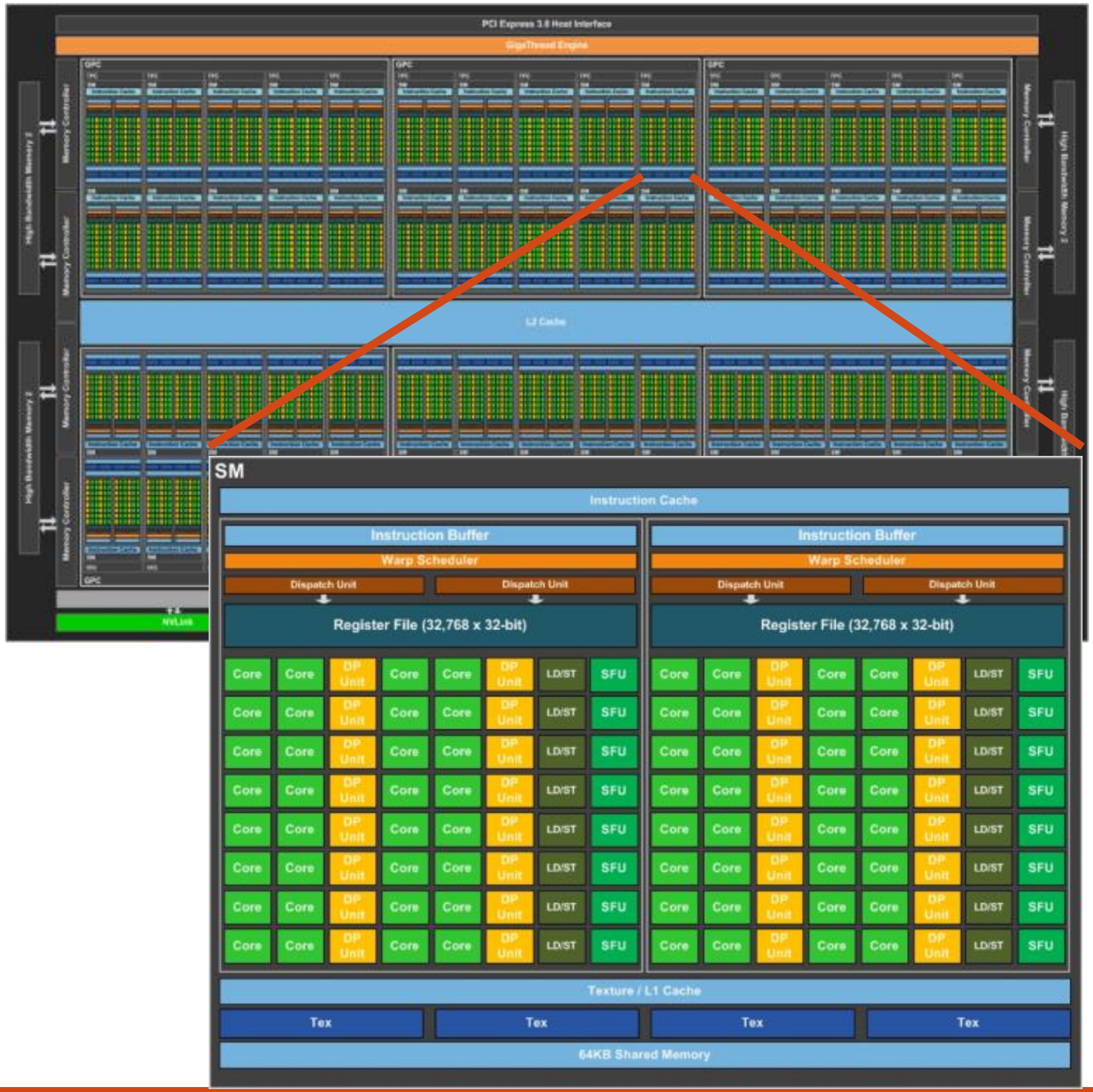
60x streaming multiprocessors (SMs)

64 “CUDA cores” each

→ 3840 total “cores”

732 GB/s mem bw using 3D stacking technology

256KB registers / SM

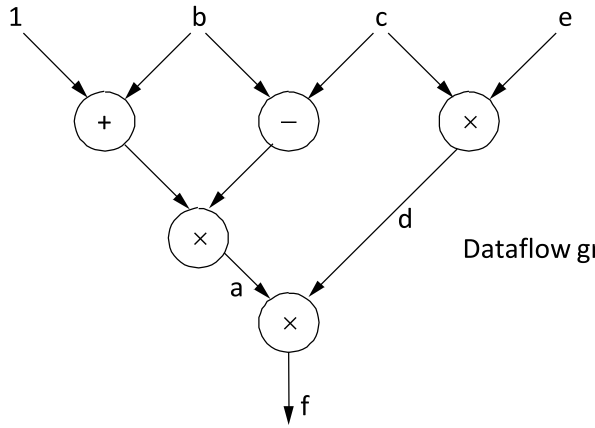


Dataflow Architectures

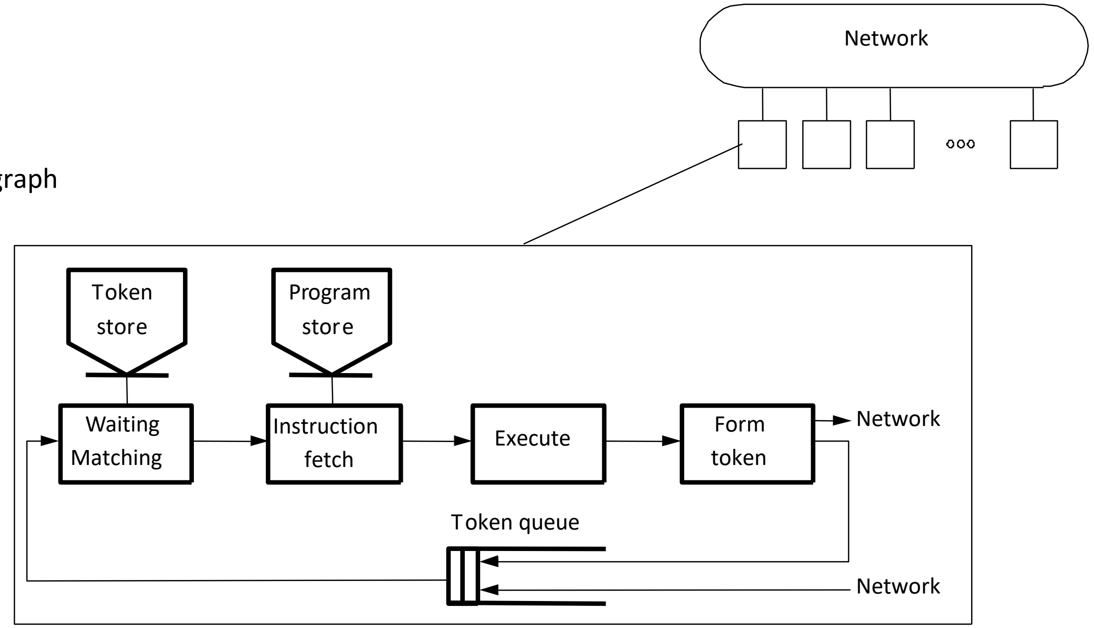
Represent computation as a graph of essential dependences

- 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



History of Dataflow

Key characteristics:

- Ability to name operations, synchronization, dynamic scheduling

Problems:

- Operations have locality & should be grouped together!!! [Swanson+, MICRO'03]
- Dataflow exposes *too much parallelism* [Culler & Arvind, ISCA'88]
- Handling 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)
- Much of the benefit of dataflow can be realized in software!
  - Loses super fine-grain operations ➔ much less parallelism

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

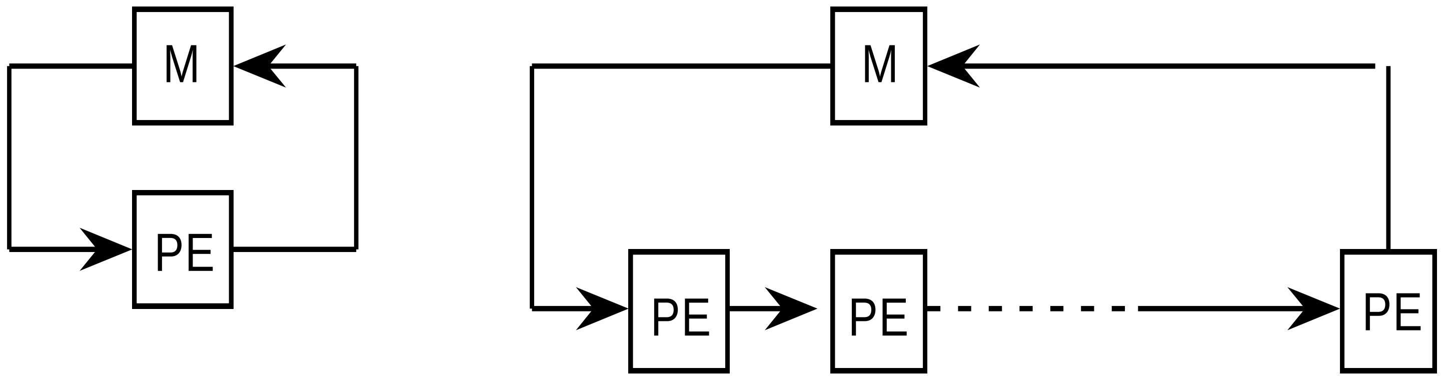
- Dataflow at multiple granularities
- Dataflow amongst many von Neumann tasks

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

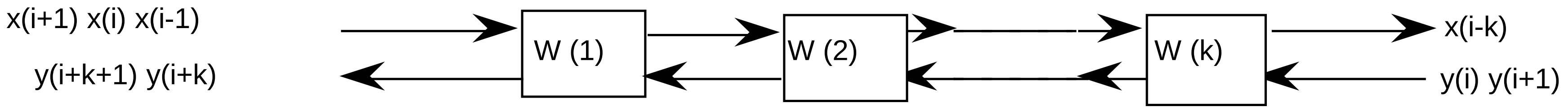
Different from SIMD: each PE may do something different

Different from dataflow: highly regular structure to computation

Initial motivation: VLSI enables inexpensive special-purpose chips, can represent algorithms directly by chips connected in regular pattern

Example & Lasting Contributions of Systolic

Example: Systolic array for 1-D convolution



$$y(i) = \sum_{j=1}^{k} w(j)*x(i-j)$$

- **Practical realizations (e.g. iWARP from CMU in late 80s) use 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**
  - E.g., Google's tensor processing unit (TPU)

MIT RAW Processor ('02)

Tiled mesh multicore

Very simple cores

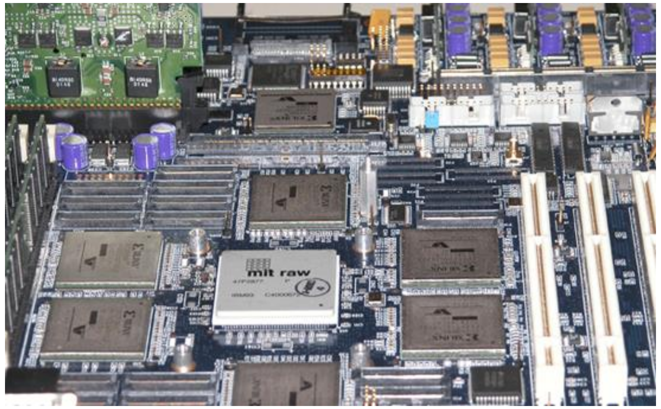
No hardware coherence

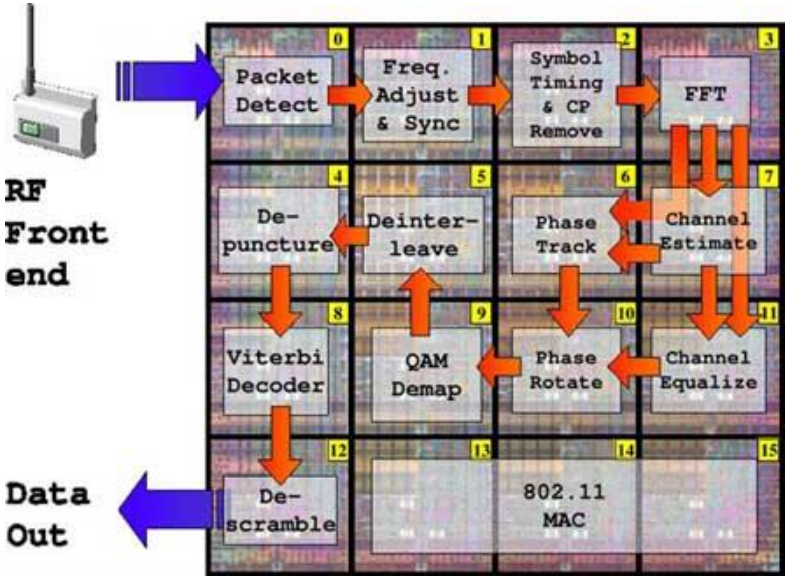
Register-to-register messaging

Programmable routers

Programs split across cores

Looks like a systolic array!





Comparison of Parallel Arch Schools

	Naming	Operations	Ordering	Processing Granularity
Sequential	All of memory	Load/store	Program	Large (ILP)
Shared memory	All of memory	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

Fundamental Issues in Parallel Architecture

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+, 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

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)

T_1	T_2

Old time: $T = T_1 + T_2$

New program (enhanced)

$T_1' = T_1$	$T_2' <= T_2$

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)

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

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:

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

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

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 for Absolute Limits

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.

Amdahl's Law for Ideal Parallel Speedup

Amdahl's Law

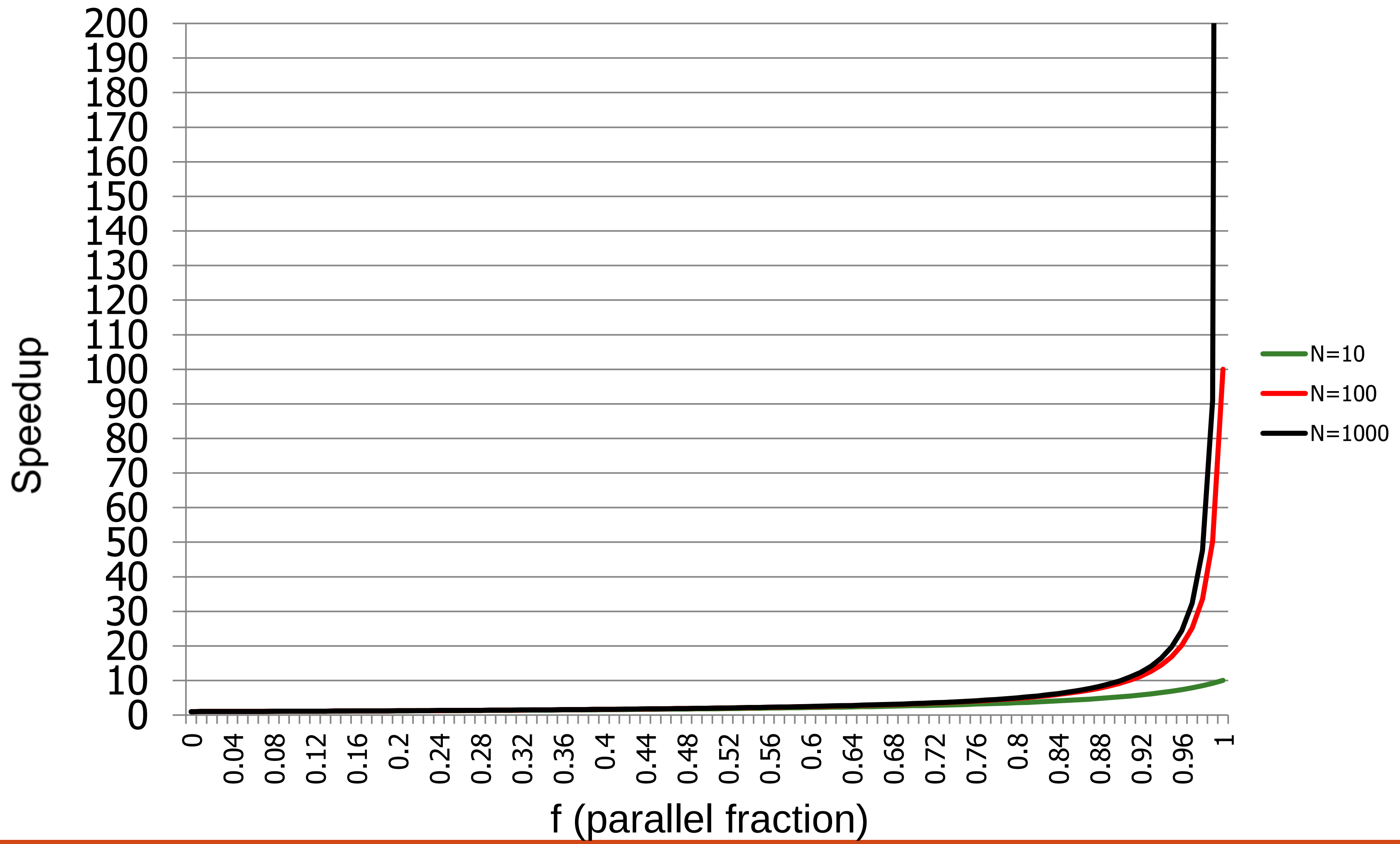
- f: Parallelizable fraction of a program
- P: Number of processors

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

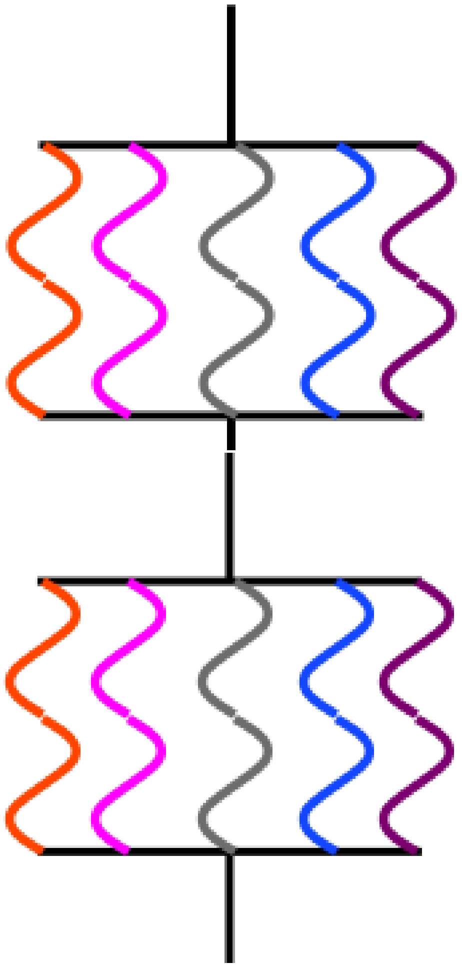
- Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

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

Corollary: The Sequential Bottleneck



Why the Sequential Bottleneck?



All parallel machines have the sequential bottleneck

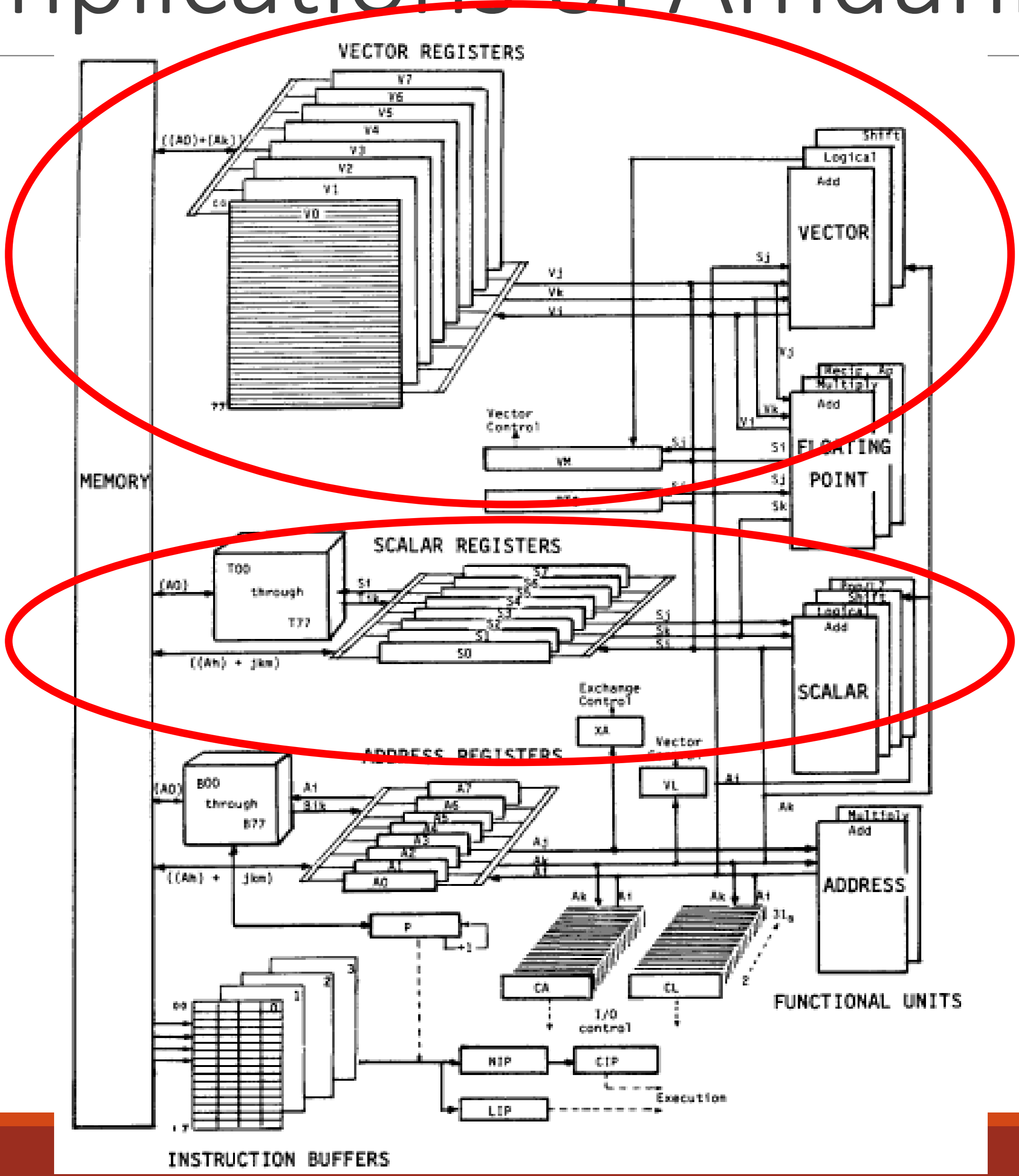
Causes:

- Non-parallelizable operations on data

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

- Synchronization: threads cannot run in parallel all the time
- Load imbalance: “stragglers” slow down program phases
- Resource sharing: threads contend on a common resource

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!

Implications of Amdahl’s Law on Design

Accelerate the sequential bottleneck! [Hill & Marty, IEEE Computer’08]

- Renewed focus on **sequential processor microarchitecture**, despite diminishing returns
  - Dynamically re-configure processor into many small cores vs few big cores? [Ipek+, ISCA’07]
- Specialize **communication & synchronization** to reduce stalls
- Hardware support for fine-grain scheduling to reduce **load imbalance**
- Architectural features to limit **resource contention** (e.g., cache/bandwidth partitioning)
- Accelerate **critical sections**, e.g., by migrating them to a faster core [Suleman+, ASPLOS’09]

Amdahl’s Law in the accelerator era

- Amdahl’s Law applies equally well to accelerator design
- Speedup from a heterogeneous SoC limited by fraction of program it accelerates
- ➔ Hard limits to performance gain from accelerators

Difficulty in Parallel Programming

Little difficulty if parallelism is natural
- "Embarrassingly parallel" applications
- Multimedia, physical simulation, graphics
- Large web services

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
- E.g., hardware transactional memory E.g., hardware transactional memory [Hammond+, ISCA'04]