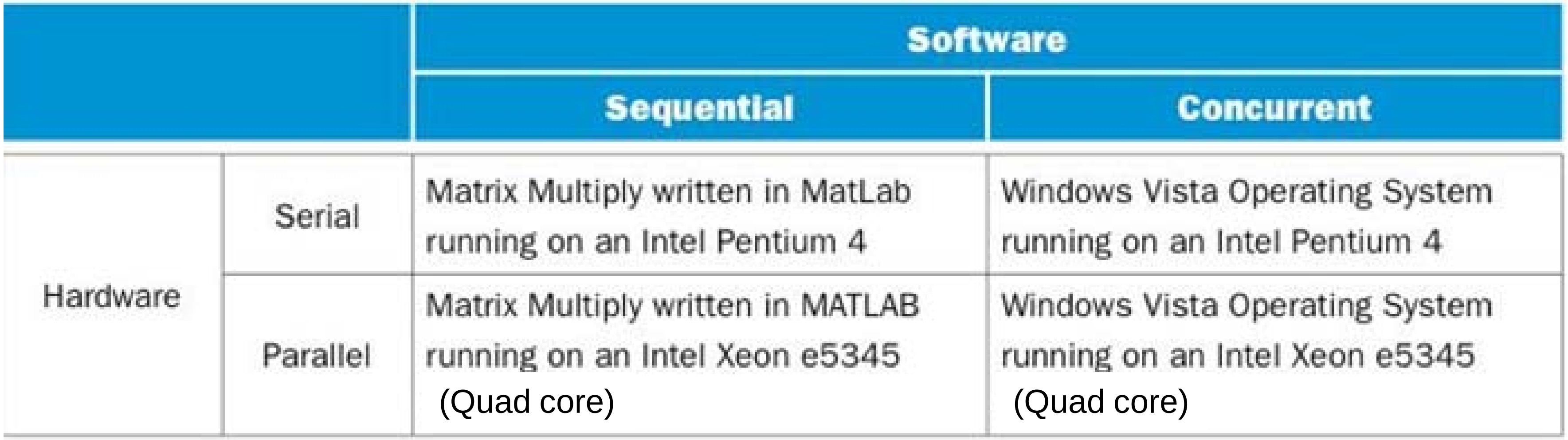
Chapter 7

Multicores, Multiprocessors, and Clusters

Introduction

- Goal: connecting multiple computers to get higher performance
  - Multiprocessors
  - Scalability, availability, power efficiency
- Job-level (process-level) parallelism
  - High throughput for independent jobs
- Parallel processing program
  - Single program run on multiple processors
- Multicore microprocessors
  - Chips with multiple processors (cores)

Hardware and Software



		Software	
		Sequential	Concurrent
Hardware	Serial	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel	Matrix Multiply written in MATLAB running on an Intel Xeon e5345 (Quad core)	Windows Vista Operating System running on an Intel Xeon e5345 (Quad core)

- Sequential/concurrent software can run on serial/parallel hardware
  - Challenge: making effective use of parallel hardware

The trouble with multi-core

- No clear notion how to program multi-core processors
- No further scaling in single core → power wall
- Intel would dedicate »all of our future product designs to multi-core environments«
- Now-defunct companies
  Ardent, Convex, Encore, Floating Point Systems, Inmos, Kendall Square Research, MasPar, nCUBE, Sequent, Tandem, Thinking Machines, …

Research since the 1960s

- Search for the right computer language:
  Hundreds, but none made it as fast, efficient and flexible as traditional sequential languages
- Design the hardware properly:
  No one has yet succeeded.
- Software to automatically parallelize sequential program:
  Depending on program, some benefit for 2, 4, (8) cores

Progress in some communities

- Data base systems for transaction processing (ATM, reservation systems)
- It is much easier to parallelize programs that deal with lots of users doing the same thing rather than a single user doing something very complicated
  ➔ inherent task-level parallelism
- Computer graphics (animated movies, special effects) individual scenes computed in parallel
  GPUs contain hundreds of processors, each tackling a small piece of rendering an image
  ➔ data-level parallelism
- Scientific computing (whether, crash simulations)
  ➔ data-level parallelism

Reasons for optimism

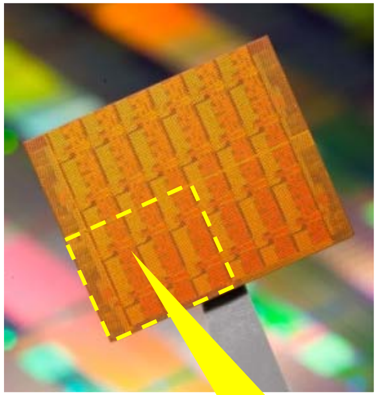
- The whole computer industry is working on the problem
- Shift to parallelism is starting and growing slowly
- Programmers start with dual- and quad-core-processors
- Degree of motivation: No further waiting for next generation processor
- Synergy between many-core processing and software as a service (cloud computing, applications running in a remote data center with millions of users)
  ➔ task-level parallelism
- Research working on a few important applications (e.g. speech recognition (understanding) in crowded, noisy, reverberant environments)

Hardware

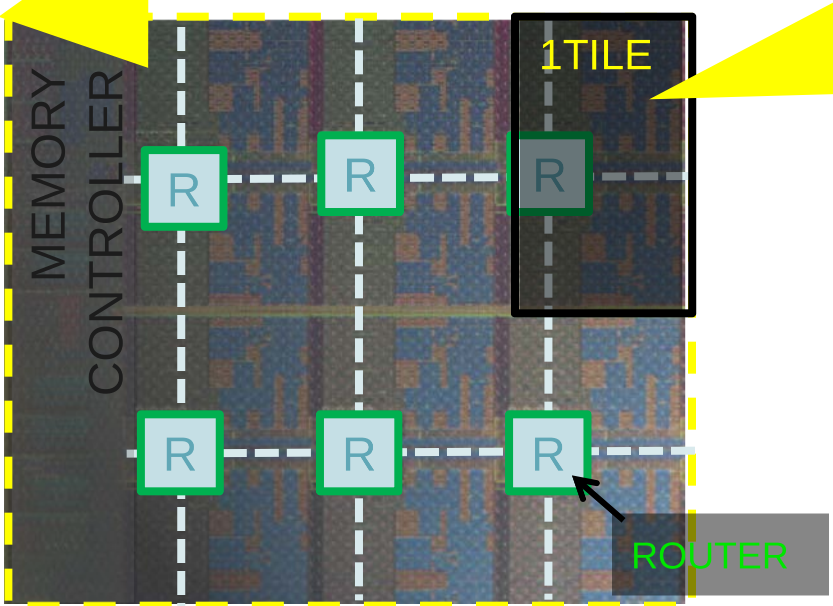
- Microprocessors with a large number of cores not yet manufactured
  ➔ no platform to run experimental software on

- FPGAs to simulate future computers, e.g. RAMP (Research Accelerator for Multiple Processors) http://ramp.eecs.berkeley.edu
  - 768–1008 MicroBlaze (32 Bit DLX-µP) cores in 64–84 FPGAs (Virtex-II Pro) on 16–21 boards

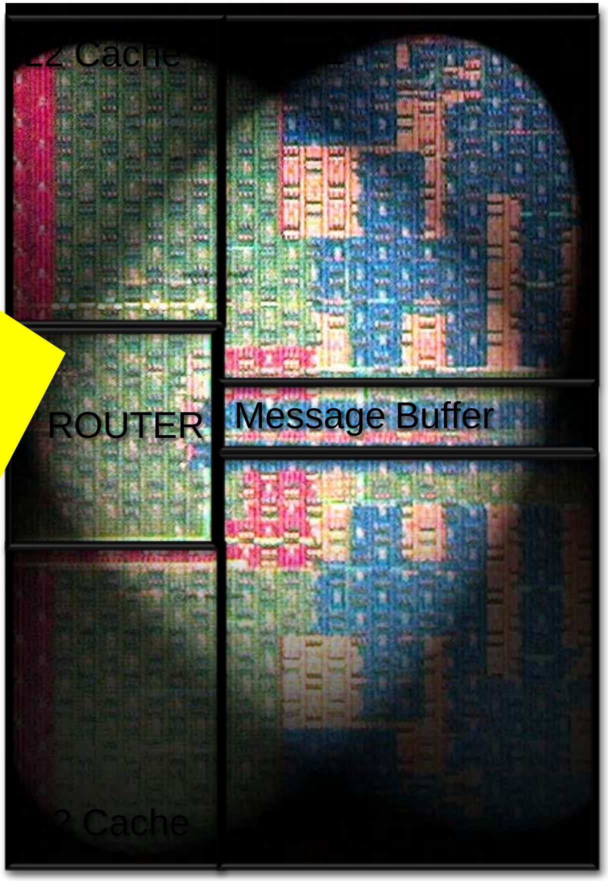
Intel: 48-Core Single-Chip Cloud Computer



- 24 Dual-core tiles
- 24 Routers
- Mesh network with 256 GB/s bisection bandwidth
- 4 Integrated DDR 3 memory controllers
- 1.3 Billion transistors



Dual-core SCC Tile



bwlsccrlb01
172.28.248.237

2020 (David Patterson)

1. Practical number of cores hits limit:
   Broad impact on IT: µP (products) still getting cheaper, but no more computational power.
   ➔ Netbook-path + cloud computing
2. Only multimedia apps can exploit data-level parallelism. Are such applications able to sustain the growth of IT?
   ➔ GPU-path
3. Someone figures out how to make dependable parallel software providing the future for the next 30 years
   ➔ optimistic path

What We've Already Covered

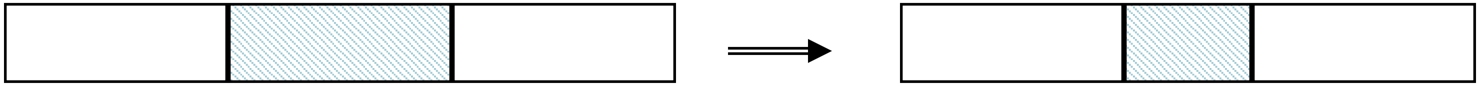
- §2.11: Parallelism and Instructions
  - Synchronization (atomic exchange)
- §3.6: Parallelism and Computer Arithmetic
  - Associativity (floating point)
- §4.10: Parallelism and Advanced Instruction-Level Parallelism (pipel., superscalar)
- §5.8: Parallelism and Memory Hierarchies
  - Cache Coherence
- §6.9: Parallelism and I/O:
  - Redundant Arrays of Inexpensive Disks

Parallel Programming

- Parallel software is the problem
- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it’s easier!
- Difficulties
  - Partitioning, load balancing
  - Coordination, synchronization
  - Communications overhead
  - Sequential dependencies

§7.2 The Difficulty of Creating Parallel Processing Programs

Amdahl’s Law



- Sequential part can limit speedup
- Example: 100 processors, 90× speedup?
  - $T_{new} = T_{parallelizable}/100 + T_{sequential}$
  - $\text{Speedup} = \dfrac{1}{(1 - F_{parallelizable}) + F_{parallelizable}/100} = 90$
  - Solving: $F_{parallelizable} = 0.999$
- Need sequential part to be 0.1% of original time

Scaling Example

- Workload: sum of 10 scalars, and 10 × 10 matrix sum
  - Speed up from 10 to 100 processors
- **Single processor**: Time = (10 + 100) × t_{add}
- **10 processors**
  - Time = 10 × t_{add} + 100/10 × t_{add} = 20 × t_{add}
  - Speedup = 110/20 = 5.5 (55% of potential)
- **100 processors**
  - Time = 10 × t_{add} + 100/100 × t_{add} = 11 × t_{add}
  - Speedup = 110/11 = 10 (10% of potential)
- ***Assumes load can be balanced across processors***

Scaling Example (cont)

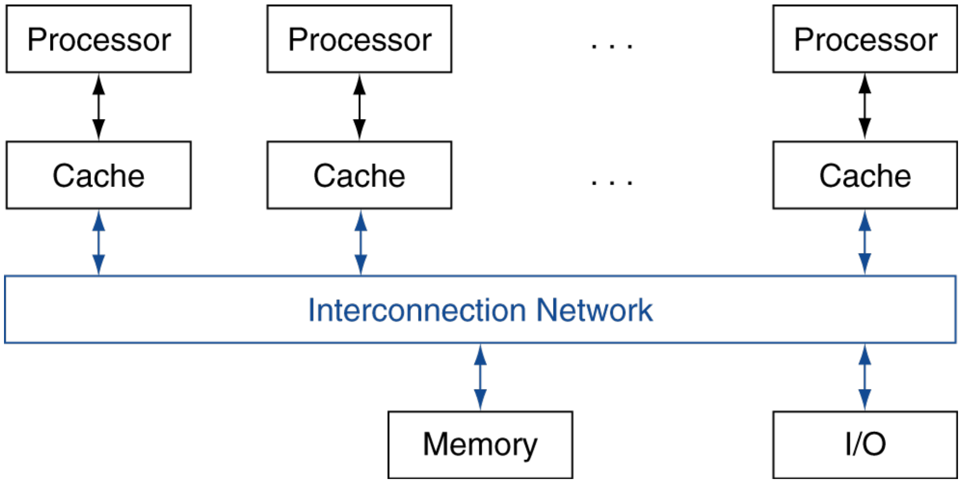
- What if matrix size is 100 × 100?
- Single processor: Time = $(10 + 10000) \times t_{add}$
- **10 processors**
  - Time = $10 \times t_{add} + 10000/10 \times t_{add} = 1010 \times t_{add}$
  - Speedup = 10010/1010 = 9.9 (99% of potential)
- **100 processors**
  - Time = $10 \times t_{add} + 10000/100 \times t_{add} = 110 \times t_{add}$
  - Speedup = 10010/110 = 91 (91% of potential)
- ***Assuming load balanced***

Strong vs Weak Scaling

- **Strong scaling**:
  speedup can be achieved on a multiprocessor without increasing the size of the problem
- **Weak scaling**:
  speedup is achieved on a multiprocessor by increasing the size of the problem proportionally to the increase in the number of processors

Shared Memory

- SMP: shared memory multiprocessor
  - Hardware provides single physical address space for all processors
  - Synchronize shared variables using locks
  - Memory access time
    - UMA (uniform) vs. NUMA (nonuniform)



Summing 100,000 Numbers on 100 Proc. SMP

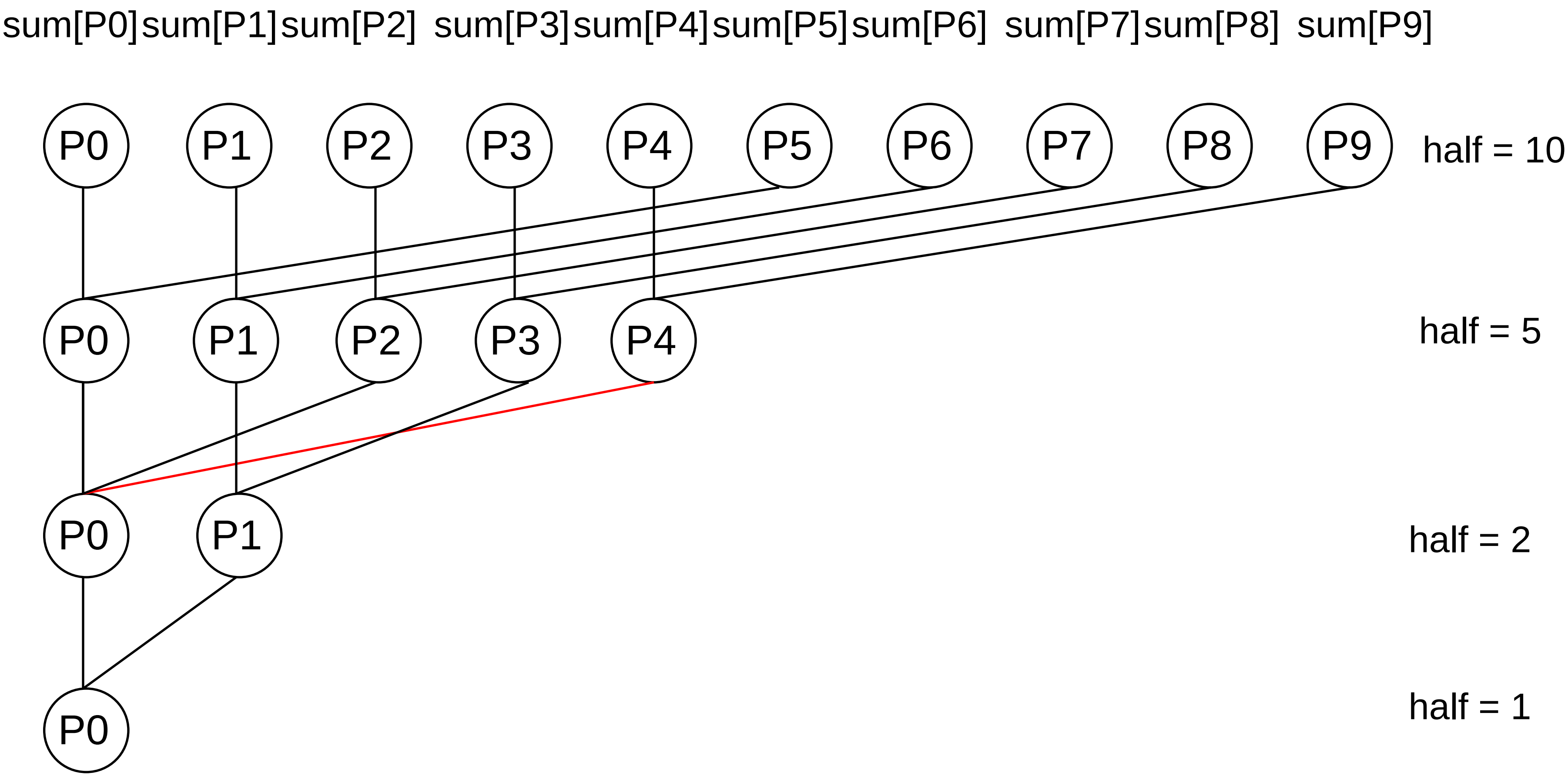
- Processors start by running a loop that sums their subset of vector A numbers (vectors A and `sum` are shared variables, `Pn` is the processor's number, `i` is a private variable)

```
sum[Pn] = 0;
for (i = 1000*Pn; i< 1000*(Pn+1); i = i + 1)
  sum[Pn] = sum[Pn] + A[i];
```

- The processors then coordinate in adding together the partial sums (`half` is a private variable initialized to `100` (the number of processors)) – reduction

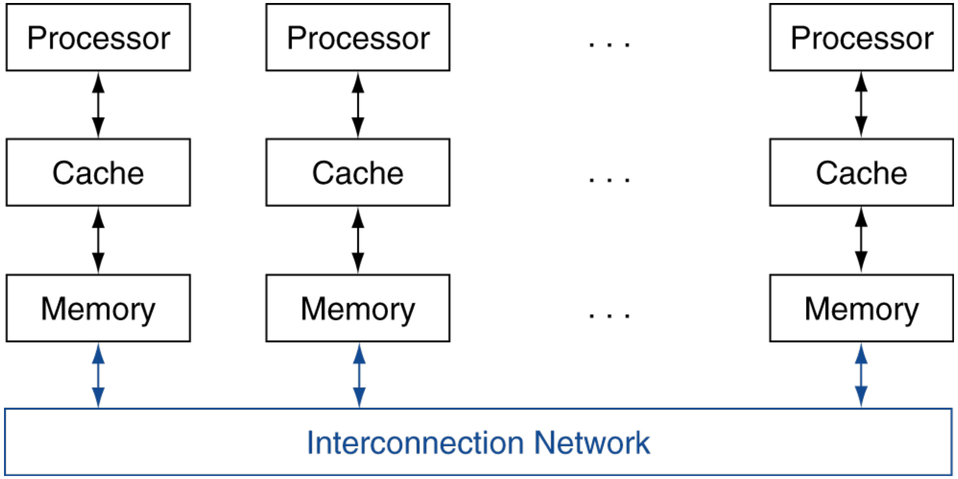
```
repeat
  synch();                  /*synchronize first
  if (half%2 != 0 && Pn == 0)
      sum[0] = sum[0] + sum[half-1];
  half = half/2
  if (Pn<half) sum[Pn] = sum[Pn] + sum[Pn+half]
until (half == 1);          /*final sum in sum[0]
```

An Example with 10 Processors



Message Passing

- Each processor has private physical address space
- Hardware sends/receives messages between processors



Loosely Coupled Clusters
Network of Workstations (NOW) Clusters

- Network of independent computers
    - Each has private memory and OS
    - Connected using I/O system
        - E.g., Ethernet/Switch, Internet
- Suitable for applications with independent tasks
    - Web servers, databases, simulations, …
- High availability, scalable, affordable
- Problems
    - Administration cost
    - Low interconnect bandwidth
        - c.f. processor/memory bandwidth on an SMP

Sum Reduction (Again)

- Sum 100,000 on 100 processors
- First distribute 100 numbers to each
  - The do partial sums

```
sum = 0;
for (i = 0; i<1000; i = i + 1)
  sum = sum + AN[i];
```

- Reduction
  - Half the processors send, other half receive and add
  - The quarter send, quarter receive and add, …

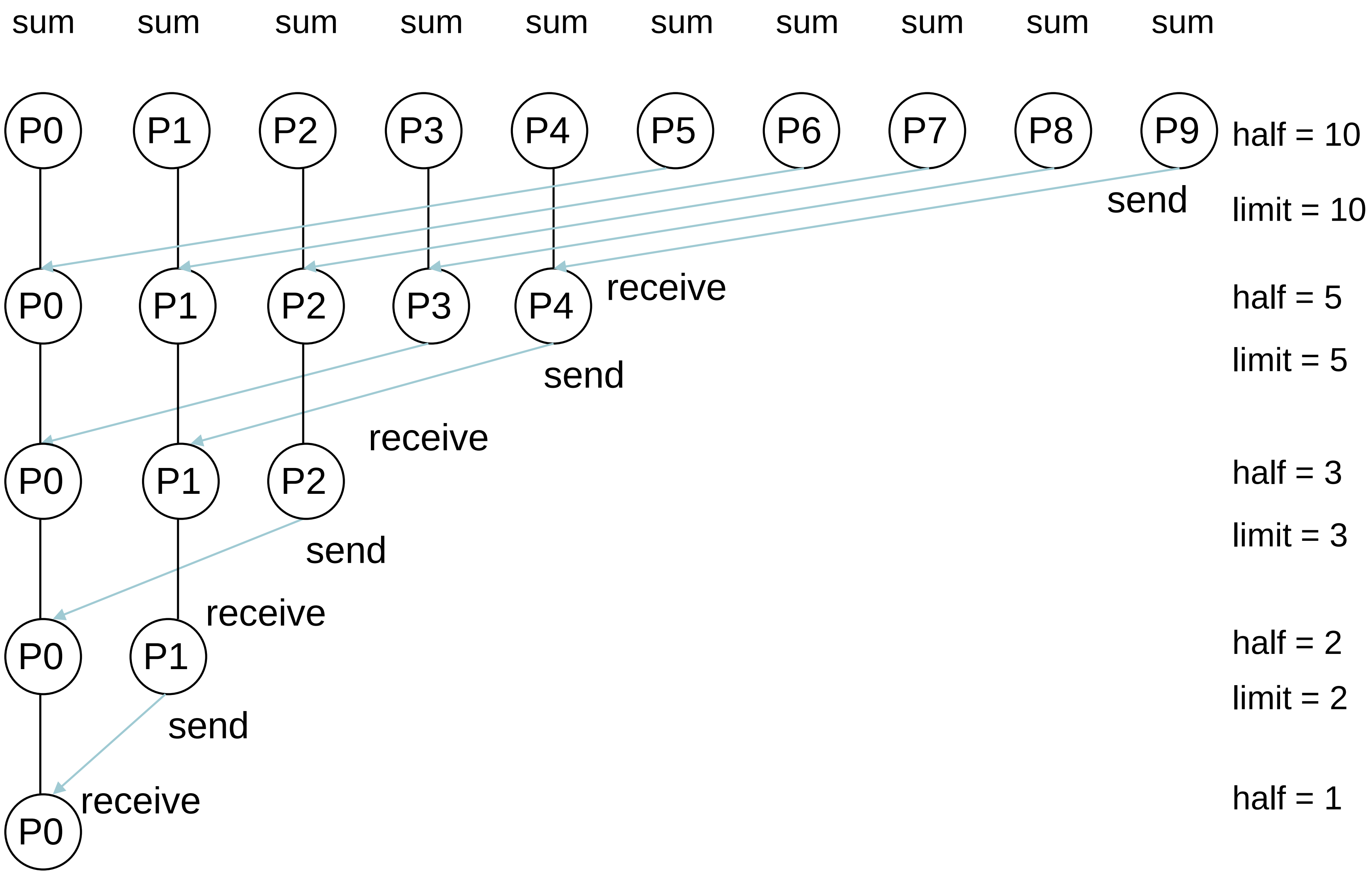
Sum Reduction (Again)

- Given send() and receive() operations

```
limit = 100; half = 100;/* 100 processors */
repeat
  half = (half+1)/2; /* send vs. receive
                        dividing line */
  if (Pn >= half && Pn < limit)
    send(Pn - half, sum);
  if (Pn < (limit/2))
    sum = sum + receive();
  limit = half; /* upper limit of senders */
until (half == 1); /* exit with final sum */
```

  - Send/receive also provide synchronization
  - Assumes send/receive take similar time to addition (which is unrealistic)

An Example with 10 Processors



Pros and Cons of Message Passing

- Message sending and receiving is ***much*** slower than addition
- But message passing multiprocessors are much easier for hardware designers to design
  - Don’t have to worry about cache coherency for example
- The advantage for programmers is that communication is explicit, so there are fewer “performance surprises” than with the implicit communication in cache-coherent SMPs.
- However, its harder to port a sequential program to a message passing multiprocessor since every communication must be identified in advance.
  - With cache-coherent shared memory the hardware figures out what data needs to be communicated

Grid Computing

- Separate computers interconnected by long-haul networks
  - E.g., Internet connections
  - Work units farmed out, results sent back
- Can make use of idle time on PCs
  - E.g., SETI@home, World Community Grid

Multithreading on a Chip

- Find a way to “hide” true data dependency stalls, cache miss stalls, and branch stalls by finding instructions (from other process threads) that are independent of those stalling instructions
- Hardware multithreading – increase the utilization of resources on a chip by allowing multiple processes (threads) to share the functional units of a single processor
  - Processor must duplicate the state hardware for each thread – a separate register file, PC, instruction buffer, and store buffer for each thread
  - The caches, TLBs, BHT, BTB, RUU can be shared (although the miss rates may increase if they are not sized accordingly)
  - The memory can be shared through virtual memory mechanisms
  - Hardware must support *efficient* thread context switching

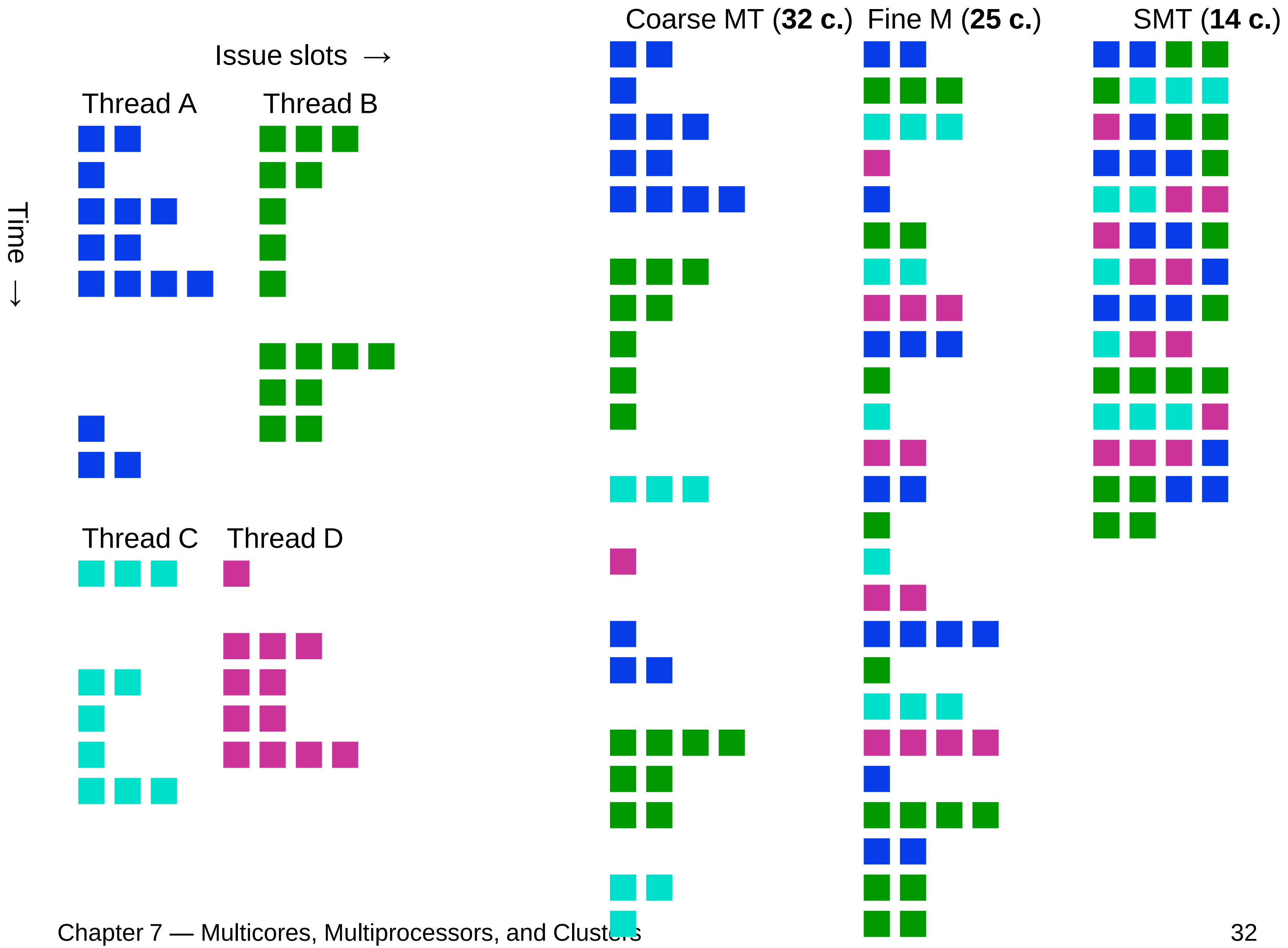
Types of Multithreading

- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn’t hide short stalls (eg, data hazards)

Simultaneous Multithreading (SMT)

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 **H**yper**T**hreading
  - Two threads: duplicated registers, shared function units and caches

Threading on a 4-way SS Processor Example



Coarse MT (**32 c.**) Fine M (**25 c.**) SMT (**14 c.**)

Issue slots →

Thread A

Thread B

Time →

Thread C

Thread D

Future of Multithreading

- **Will it survive? In what form?**
- Power considerations ⇒ simplified microarchitectures
  - Simpler forms of multithreading
- Tolerating cache-miss latency
  - Thread switch may be most effective
- Multiple simple cores might share resources more effectively

Instruction and Data Streams

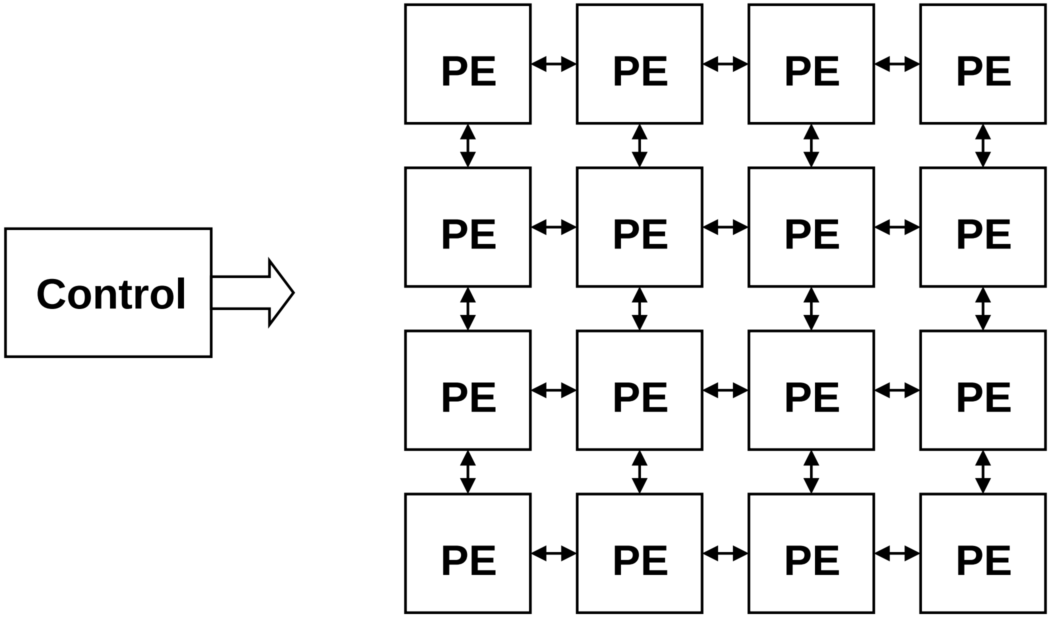
- An alternate (Flynn) classification

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

- SPMD: Single Program Multiple Data
  - A single parallel program on a MIMD computer running across all processors

§7.6 SISD, MIMD, SIMD, SPMD, and Vector

SIMD Processors



- Single control unit (one copy of the code)
- Multiple datapaths (Processing Elements – PEs) running in parallel

SIMD

- Operate element wise on vectors of data
  - E.g., MMX and SSE instructions in x86
    - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time
  - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications

Examples of SIMD Machines

	Maker	Year	# PEs	# b/ PE	Max memory (MB)	PE clock (MHz)	System BW (MB/s)
Illiac IV	UIUC	1976	64	64	1	13	2,560
DAP	ICL	1980	4,096	1	2	5	2,560
MPP	Goodyear	1982	16,384	1	2	10	20,480
CM-2	Thinking Machines	1987	65,536	1	512	7	16,384
MP-1216	MasPar	1989	16,384	4	1024	25	23,000

- Did SIMDs die out in the early 1990s ?

  No, today they are everywhere!

Vector Processors

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- Example: Vector extension to MIPS
  - 32 × 64-element registers (64-bit elements)
  - Vector instructions
    - `lv`, `sv`: load/store vector
    - `addv.d`: add vectors of double
    - `addvs.d`: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

Example: DAXPY (Y = a × X + Y)

- Conventional MIPS code

```
      l.d   $f0,a($sp)     ;load scalar a
      addiu r4,$s0,#512    ;upper bound of what to load
loop: l.d   $f2,0($s0)     ;load x(i)
      mul.d $f2,$f2,$f0    ;a × x(i)
      l.d   $f4,0($s1)     ;load y(i)
      add.d $f4,$f4,$f2    ;a × x(i) + y(i)
      s.d   $f4,0($s1)     ;store into y(i)
      addiu $s0,$s0,#8     ;increment index to x
      addiu $s1,$s1,#8     ;increment index to y
      subu  $t0,r4,$s0     ;compute bound
      bne   $t0,$zero,loop ;check if done
```

- Vector MIPS code

```
      l.d     $f0,a($sp)   ;load scalar a
      lv      $v1,0($s0)   ;load vector x
      mulvs.d $v2,$v1,$f0  ;vector-scalar multiply
      lv      $v3,0($s1)   ;load vector y
      addv.d  $v4,$v2,$v3  ;add y to product
      sv      $v4,0($s1)   ;store the result
```

Pipeline stalls once per vector operation , rather than once by vector element

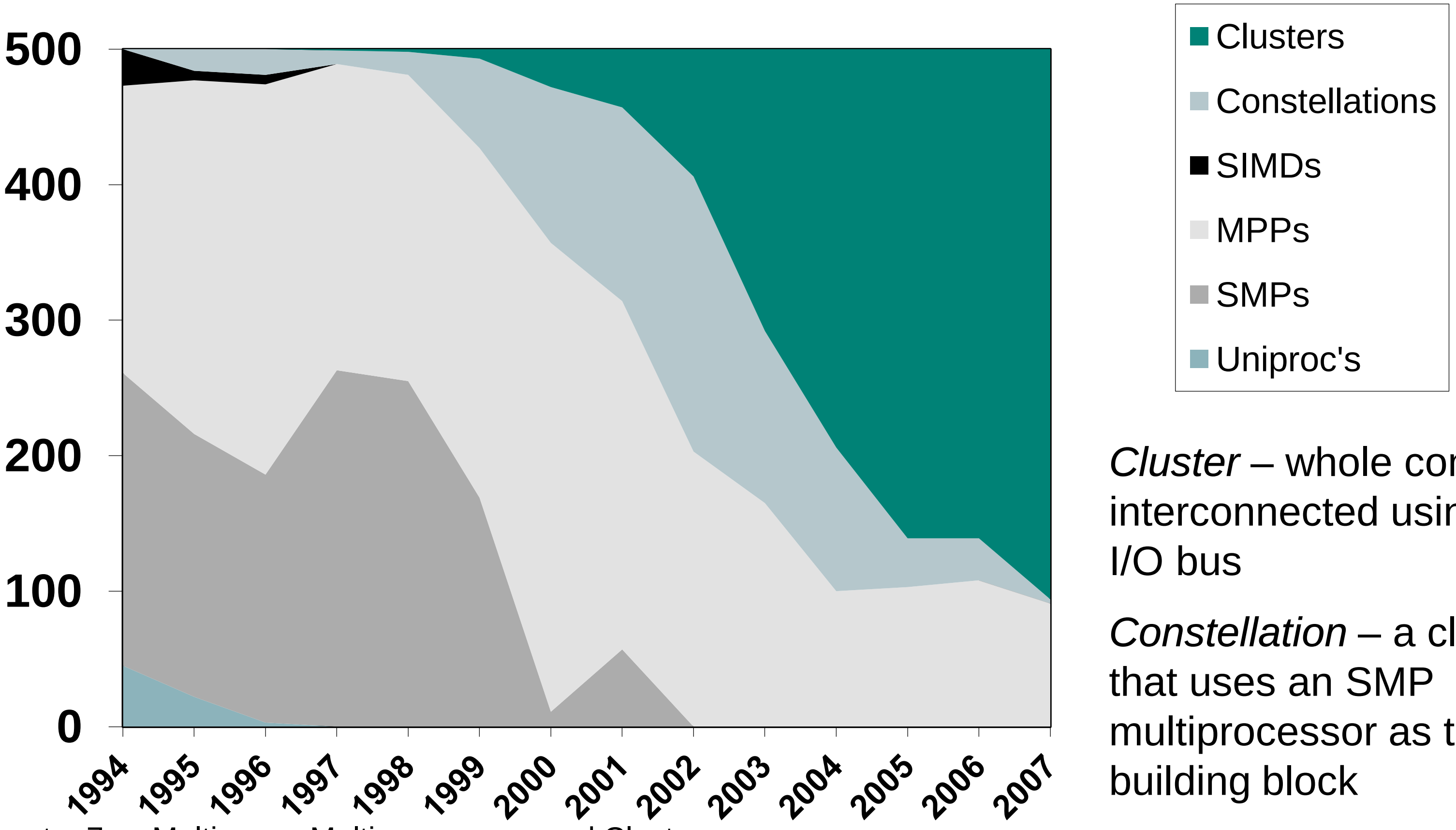
Vector vs. Scalar

- Vector architectures and compilers
  - Simplify data-parallel programming
  - Explicit statement of absence of loop-carried dependences
    - Reduced checking in hardware
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- More general than ad-hoc media extensions (such as MMX, SSE)
  - Better match with compiler technology

Supercomputer Style Migration (Top500)

http://www.top500.org

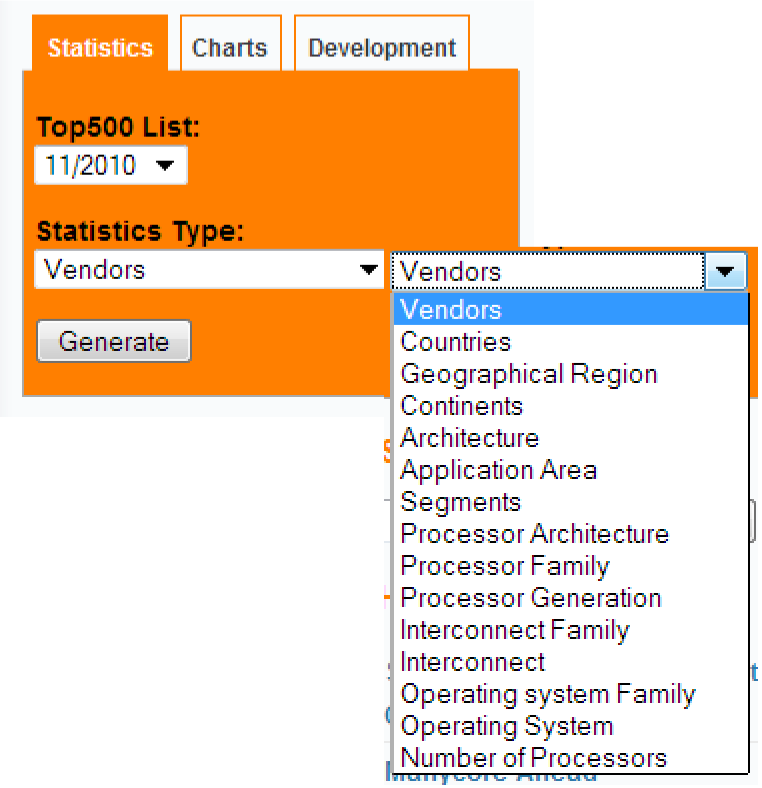
- Uniprocessors and SIMDs disappeared while Clusters and Constellations grew from 3% to 80%. Now its 98% Clusters and MPPs.
- **2010: 2 Constellations, 84 MPP, 414 Clusters**



Cluster – whole computers interconnected using their I/O bus

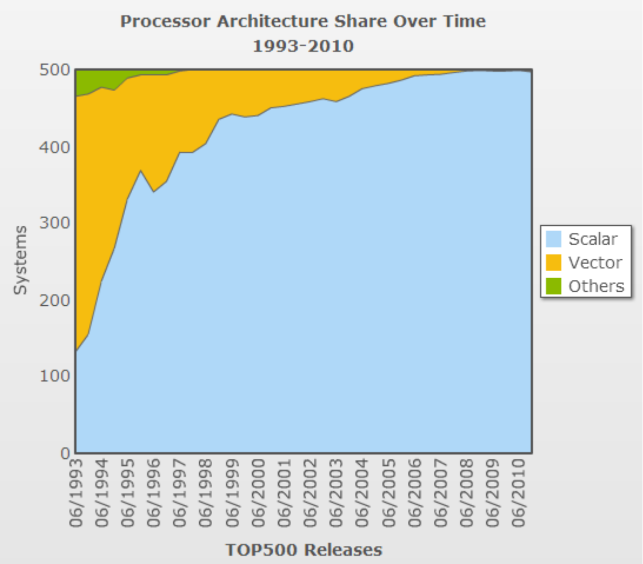
Constellation – a cluster that uses an SMP multiprocessor as the building block

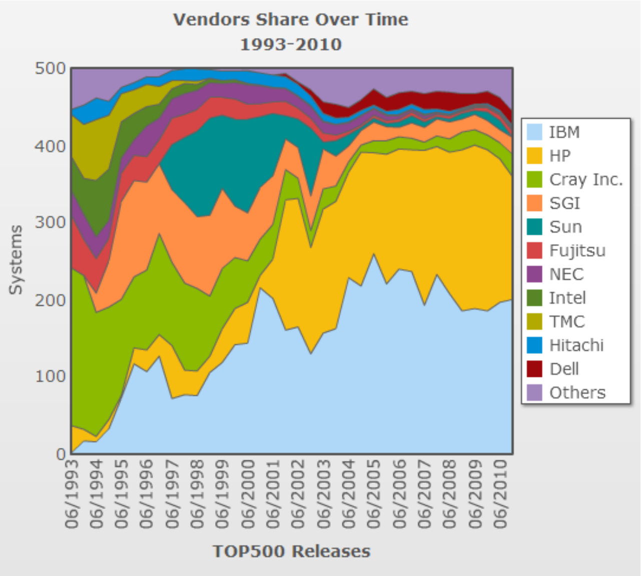
Top 500 Statistics / Charts / Development



Top500 List: 11/2010

Statistics Type: Vendors


Processor Architecture Share Over Time
1993-2010

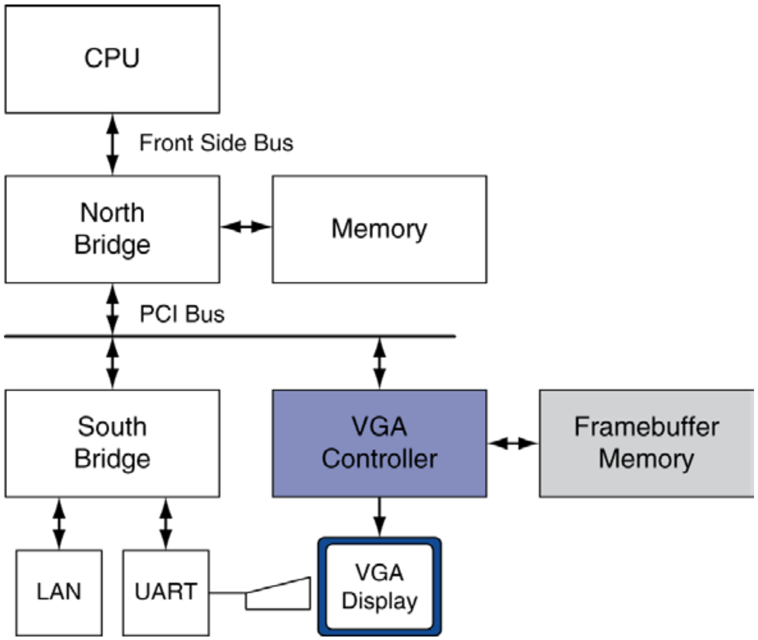

Vendors Share Over Time
1993-2010

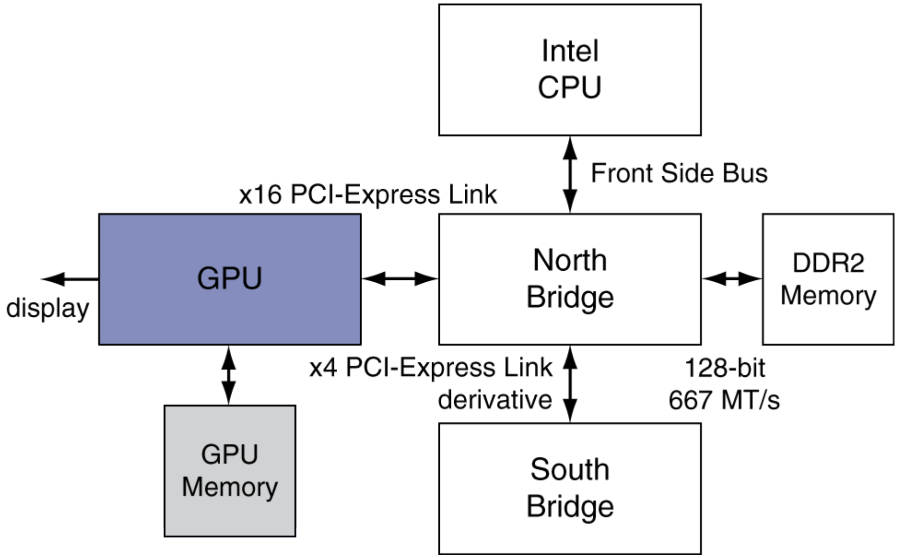
Vendors Share Over Time
1993-2010

History of GPUs

- Early video cards
  - Frame buffer memory with address generation for video output
- 3D graphics processing
  - Originally high-end computers (e.g., SGI)
  - Moore's Law ⇒ lower cost, higher density
  - 3D graphics cards for PCs and game consoles
- Graphics Processing Units
  - Processors oriented to 3D graphics tasks
  - Vertex/pixel processing, shading, texture mapping, rasterization

Graphics in the System



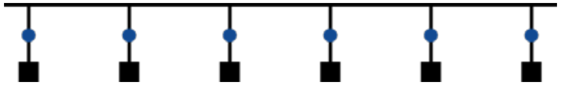


GPU Architectures

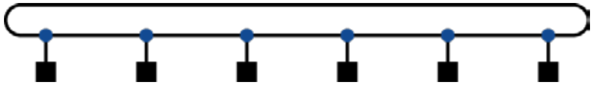
- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code
- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)

Interconnection Networks

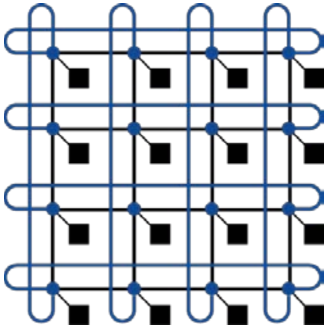
- Network topologies
  - Arrangements of processors, switches, and links



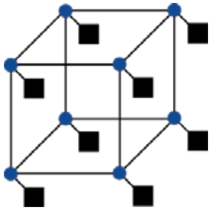
Bus



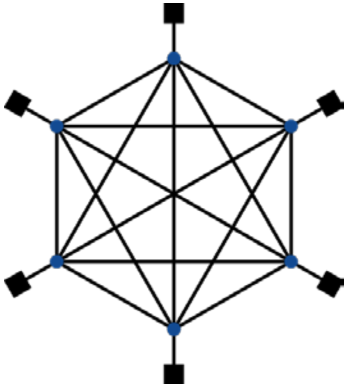
Ring



2D Mesh



N-cube (N = 3)

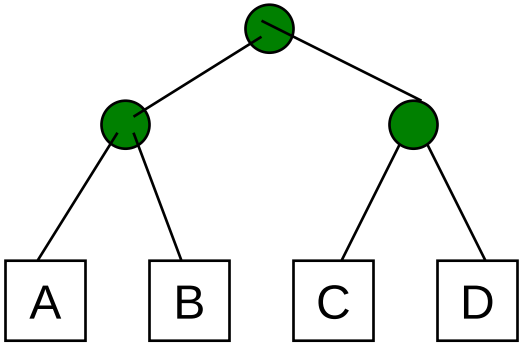


Fully connected

§7.8 Introduction to Multiprocessor Network Topologies

“Fat” Trees

- Trees are good structures. People in CS use them all the time. Suppose we wanted to make a tree network.



- Any time A wants to send to C, it ties up the upper links, so that B can't send to D.
- The solution is to 'thicken' the upper links.
  - Have more links as you work towards the root of the tree

Network Characteristics

- Performance
  - Latency per message (unloaded network)
  - Throughput
    - Link bandwidth
    - Total network bandwidth (represents best case)
      bandwidth of each link * number of links
    - Bisection bandwidth (closer to worst case)
      divide the machine in two parts, each with half the nodes and sum the bandwidth of the links that cross the dividing line
  - Congestion delays (depending on traffic)
- Cost
- Power
- Routability in silicon (NoC)

Parallel Benchmarks

- Linpack: matrix linear algebra
- SPECrate: parallel run of SPEC CPU programs
  - Job-level parallelism
- SPLASH: Stanford Parallel Applications for Shared Memory
  - Mix of kernels and applications, strong scaling
- NAS (NASA Advanced Supercomputing) suite
  - computational fluid dynamics kernels
- PARSEC (Princeton Application Repository for Shared Memory Computers) suite
  - Multithreaded applications using Pthreads and OpenMP

Code or Applications?

- Traditional benchmarks
  - Fixed code and data sets
- Parallel programming is evolving
  - Should algorithms, programming languages, and tools be part of the system?
  - Compare systems, provided they implement a given application
  - E.g., Linpack, Berkeley Design Patterns
- Would foster innovation in approaches to parallelism

Fallacies

- Amdahl’s Law doesn’t apply to parallel computers
  - Since we can achieve linear speedup
  - But only on applications with weak scaling
- Peak performance tracks observed performance
  - Marketers like this approach!
  - But compare Xeon with others in example
  - Need to be aware of bottlenecks

Pitfalls

- Not developing the software to take account of a multiprocessor architecture

Concluding Remarks

- Goal: higher performance by using multiple processors
- Difficulties
  - Developing parallel software
  - Devising appropriate architectures
- Many reasons for optimism
  - Changing software and application environment
  - Chip-level multiprocessors with lower latency, higher bandwidth interconnect
- An ongoing challenge for computer architects!