Chapter 7A

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
- Cluster: a set of computers connected over a LAN that functions as a single large multiprocessor
  - Scientific problems, web servers, databases

Introduction

- Now programmers must become parallel programmers
- Challenge
  - How to create HW and SW that will make it easy to write correct parallel processing programs that will execute efficiently in performance and power as the number of cores per chip scales geometrically (?).

Categorization

- Hardware
  - Serial: e.g., Pentium 4
  - Parallel: e.g., quad-core Xeon e5345
- Software
  - Sequential: e.g., matrix multiplication, compiler
  - Concurrent: e.g., operating system

		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 (Clovertown)	Windows Vista Operating System running on an Intel Xeon e5345 (Clovertown)

Hardware and Software

- Sequential/concurrent software can run on serial/parallel hardware
  - Challenge: making effective use of parallel hardware
- In this chapter, we will use parallel processing program or parallel software to mean either sequential or concurrent software running on parallel computer

Sections in chapter 7

- The sections
  - 7.2: difficulty of creating parallel programs
  - 7.3: shared memory multiprocessor
  - 7.4: clusters (message passing multiprocessors)
  - 7.5: multithreading
  - 7.6: an older classification scheme (SIMD, vector)
  - 7.7: graphic processing unit (GPU)
  - 7.8: network topologies
  - 7.9: multiprocessor benchmarks

What We've Already Covered

- What We've Already Covered
  - §2.11: Parallelism and Instructions
    - Synchronization
  - §3.6: Parallelism and Computer Arithmetic
    - Associativity
  - §4.10: Parallelism and Advanced Instruction-Level Parallelism
  - §5.8: Parallelism and Memory Hierarchies
    - Cache Coherence
  - §6.9: Parallelism and I/O:
    - Redundant Arrays of Inexpensive Disks

§3.6: Parallelism and Computer Arithmetic

- Integer addition is associative
  - If you were to add a million numbers together, you would get the same result whether you used 1 processor or 100 processor.
- Floating-point addition is not associative
  - because floating-point numbers are approximation and
  - because computer arithmetic has limited precision
- Parallel code with floating-point numbers should confirm it with numerical analysis
  - Validated numerical libraries such as
    - LAPACK: linear algebra
    - SCALAPACK: scalable LAPACK

Parallel Programming

- Too few important application programs have been written to complete tasks sooner on multiprocessors.
- It is difficult to write software that uses multiple processors to complete one task faster, and the problem gets worse as the number of processors increases.
- Why have parallel processing programs been so much harder to develop than sequential programs?

Parallel Programming

- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it's easier!
- Uniprocessor design techniques such as superscalar and out-of-order execution exploit ILP
  - Normally without involvement of programmer
  - Reduces the demand for rewriting programs for multiprocessors

Parallel Programming

- Why is it difficult to write parallel processing programs that is fast, especially as the number of processors increases?
  - Eight reporters try to write a single story in hopes of doing the work eight times faster.
- Difficulties
  - Partitioning,
  - Coordination
    - Scheduling, load balancing, Synchronization
  - Communications overhead

Speed-up Challenge

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

Strong vs Weak Scaling

- Strong scaling means measuring speed-up while keeping the problem size is fixed.
  - Strong scaling is defined as how the solution time varies with the number of processors for a fixed *total* problem size
- Weak scaling means that the problem size grows proportionally to the increase in the number of processors.
  - Weak scaling is defined as how the solution time varies with the number of processors for a fixed problem size *per processor*.

Speed-up challenge

- Workload: sum of 10 scalars, and 10 × 10 matrix sum
  - Speed up from 10 to 100 processors
- Assumes load can be balanced across processors
- Single processor: Time = (10 + 100) × t_{add}
- 10 processors
  - Time = 10 × t_{add} + 100/10 × t_{add} = 20t_{add}
  - Speedup = 110/20 = 5.5 (55% of potential)
- 100 processors
  - Time = 10 × t_{add} + 100/100 × t_{add} = 11t_{add}
  - Speedup = 110/11 = 10 (10% of potential)
- Strong scaling condition: the problem size is fixed while scaling up the system.

For bigger problem

- Scaling up the problem: 100x
  - What if matrix size is 100 × 100?
- Single processor: Time = (10 + 10000) × t_{add}
- 10 processors
  - Time = 10 × t_{add} + 10000/10 × t_{add} = 1010t_{add}
  - Speedup = 10010/1010 = 9.9 (99% of potential 10)
- 100 processors
  - Time = 10 × t_{add} + 10000/100 × t_{add} = 110t_{add}
  - Speedup = 10010/110 = 91 (91% of potential 100)
- For a larger problem, we get higher percent of the potential speedup

The size of the problem

- Weak scaling condition: the program size grows proportionally to the number of processors in the system.
  - The previous examples shows that the speedup for strong scaling is harder than that for weak scaling.
- Assume that the size of the problem, M, is the working set in the memory, and we have P processors.
  - For strong scaling, the memory per processor is approximately M/P.
  - For weak scaling, the memory per processor is approximately M.

Gustafson's law

- The key assumption here is that the total amount of work to be done in parallel *varies linearly with the number of processors, P*.
  - a (serial part) + b (parallel part) : on parallel machine
  - a + Pb : a single processor
  - Gain = a + Pb / (a + b)

    = α + P (1- α) if α = a/(a+b)

    = (1 – F) + PF if F = b/(a+b)

A driving metaphor

- Amdahl's law: fixed distance
  - Suppose a car is traveling between two cities 60 miles apart, and has already spent one hour traveling half the distance at 30 mph.
  - No matter how fast you drive the last half, it is impossible to achieve 90 mph average before reaching the second city.
- Gustafson's law: Given enough time and distance
  - Suppose a car has already been traveling for some time at less than 90mph.
  - Given enough time and distance to travel, the car's average speed can always eventually reach 90mph, no matter how long or how slowly it has already traveled.

Weak Scaling

- Weak Scaling is the most interesting for O(N) algorithms.
- In this case perfect weak scaling is a constant time to solution, independent of processor count.
- Deviations from this indicate that either
  - the algorithm is not truly O(N) or
  - the overhead due to parallelism is increasing,
  - or both.

Load Balancing

- To achieve the speed-up of 91 on the previous larger problem with 100 processors, we assumed the load was perfectly balanced.

- Show the impact on speed-up if one processor's load is higher than all the rest.
  - at 2x (1%): 10t + max (200t, 9800t/99)= 210t

    10010t/210t = 48 (reduced from 91)
  - at 5x (5%): 10t + max (500t, 9500t/99)= 510t

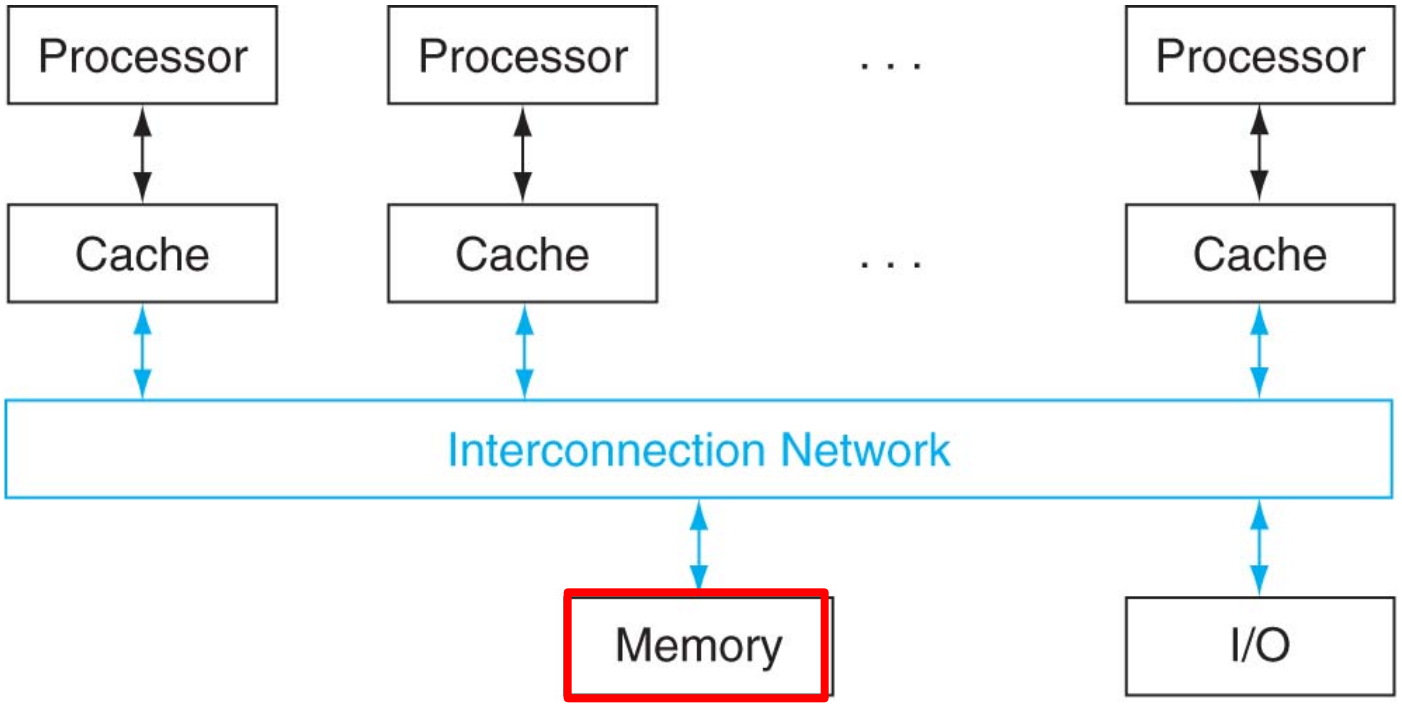
    10010t/510t = 20 (reduced from 91)
  - This example demonstrates the speed-up is very sensitive to load balancing

Shared Memory

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

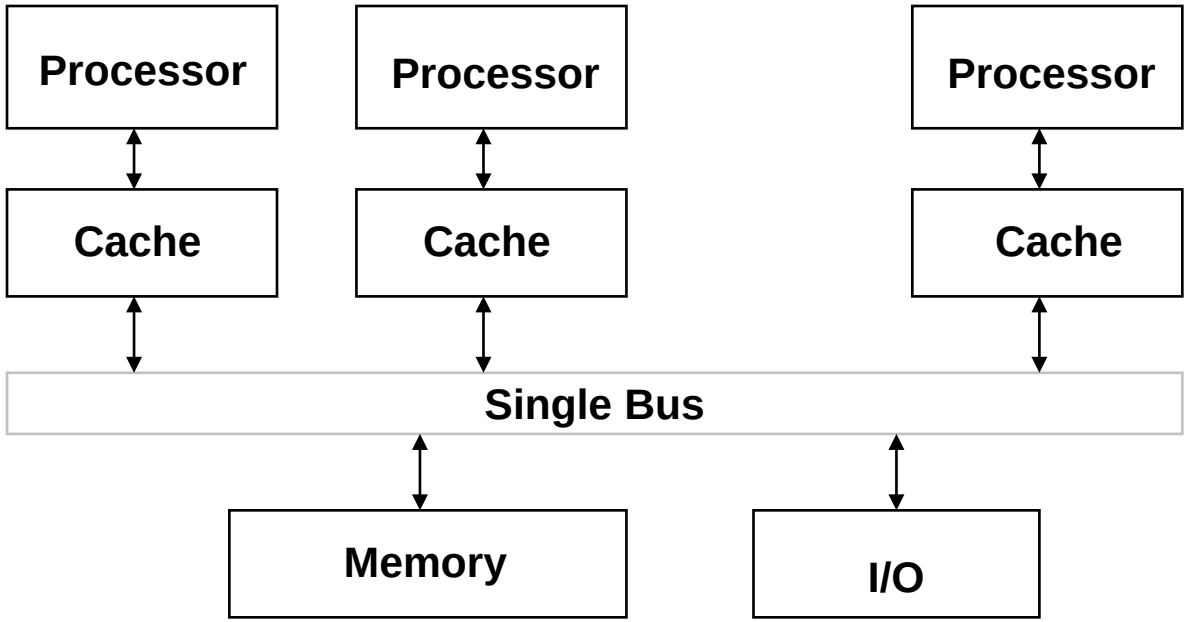
			# of Proc
Communication model	Message passing		8 to many
	Shared address	NUMA	8 to many
		UMA	2 to 64
Physical connection	Network		8 to many
	Bus		2 to 36

Organization of a SMP



- Shared memory does not mean that there is a single, centralized memory.
  - Symmetric shared-memory: UMA
  - Distributed shared-memory: NUMA

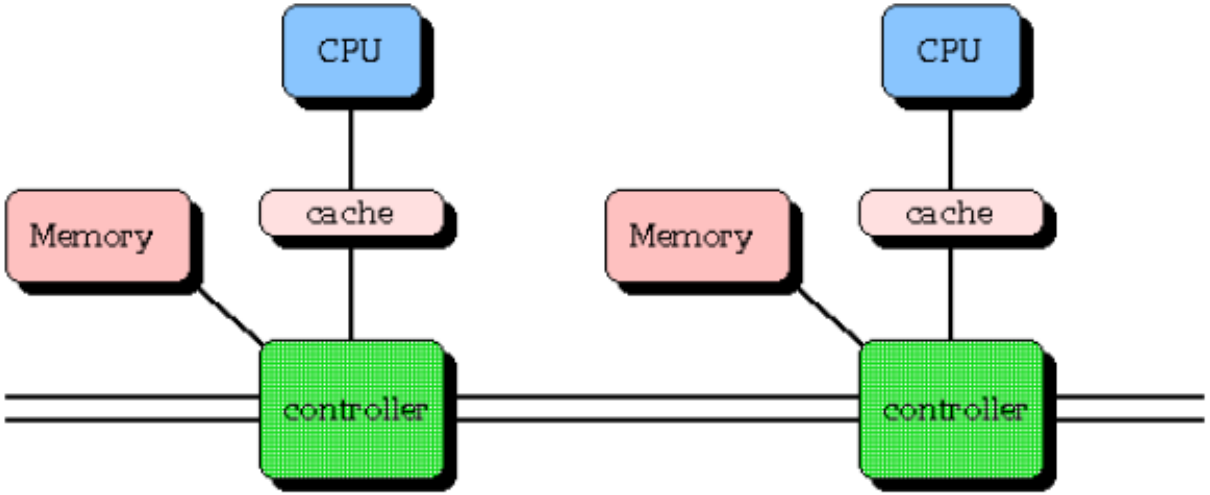
Single-Bus UMA SMP



- Caches are used to reduce latency *and* to lower bus traffic
- Must provide hardware to ensure that caches and memory are consistent (cache coherency)
- Must provide a hardware mechanism to support process synchronization (the process of coordinating the behavior of two or more processes, which may be running on different processors)

NUMA

- Often made by physically linked SMPs
  - One SMP can directly access memory of another SMP
  - Not all processors have equal access time to all memories
  - Memory access across link is slower
  - If cache coherence is maintained, called CC-NUMA



Shared Memory Multiprocessors

- Processors coordinate/communicate through shared variables in memory (via loads and stores)
  - Use of shared data must be coordinated via synchronization primitives (locks)
- UMA (uniform memory access) – aka SMP(?)(symmetric multiprocessors)
  - all accesses to main memory take the same amount of time no matter which processor makes the request or which location is requested
- NUMA (nonuniform memory access)
  - some main memory accesses are faster than others depending on the processor making the request and which location is requested
  - can scale to larger sizes than UMAs so are potentially higher performance

What does SMP stand for?

- SMP: symmetric memory multiprocessor
  - A computer architecture that provides fast performance by making multiple CPUs available to complete individual processes simultaneously (multiprocessing).
  - Unlike asymmetrical processing, any idle processor can be assigned any task, and additional CPUs can be added to improve performance and handle increased loads.
  - A variety of specialized operating systems and hardware arrangements are available to support SMP.
  - Specific applications can benefit from SMP if the code allows multithreading.
  - SMP uses a single operating system and shares common memory and disk input/output resources.
  - Both UNIX and Windows NT support SMP

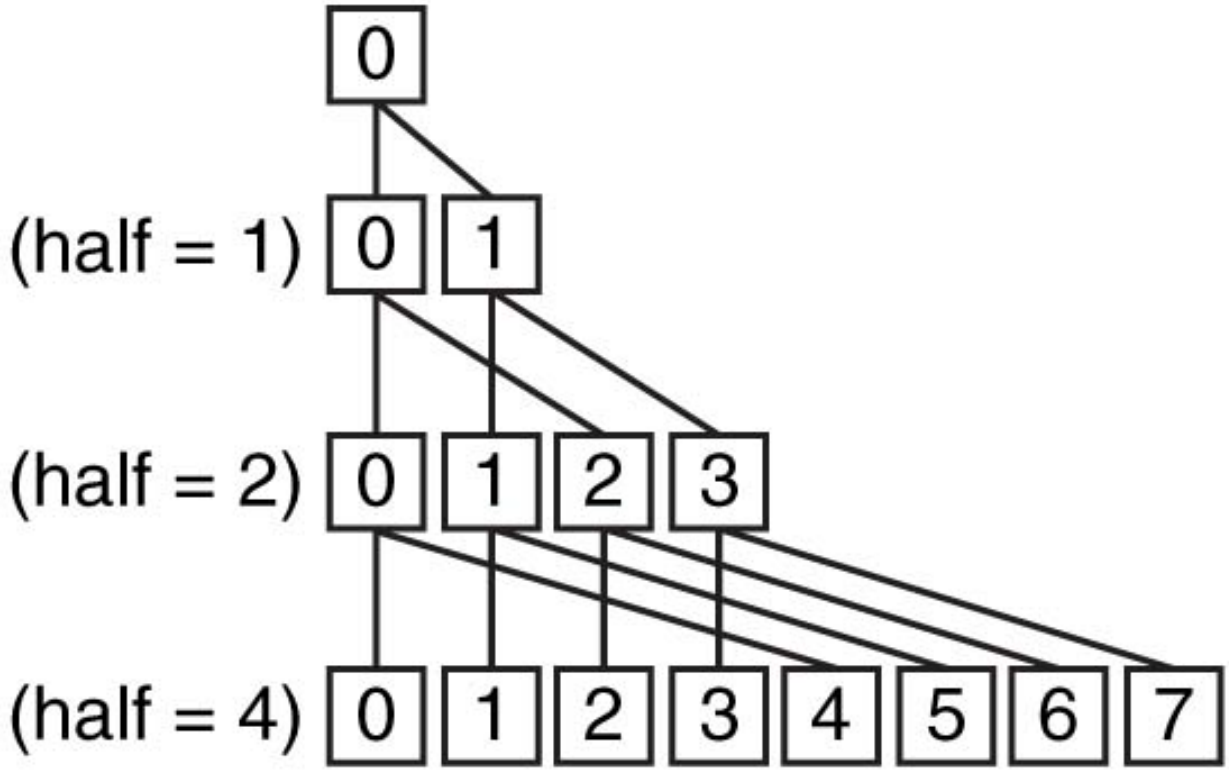
Example: Sum Reduction

- Sum 100,000 numbers on 100 processor UMA
  - Each processor has ID: 0 ≤ Pn ≤ 99
  - Partition 1000 numbers per processor
  - Initial summation on each processor

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

- Now need to add these 100 partial sums
  - Reduction: a function that processes a data structure and returns a single value
  - Half the processors add pairs, then quarter, …
    - An inverse tree
  - Need to synchronize between reduction steps

Last four levels of a reduction



The last four levels of a reduction that sums results from each processor, from bottom to top. For all processors whose number i is less than half, add the sum produced by processor number (i + half) to its sum.

Example: Sum Reduction

Pn is the number identifying the processor
Code for Pn; i and half are private variables

```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i = i + 1)
  sum[Pn] = sum[Pn] + A[i];
half = 100;
repeat
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets an additional element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```

100
50
25
12
6
3
1

Multiprocessor Organizations

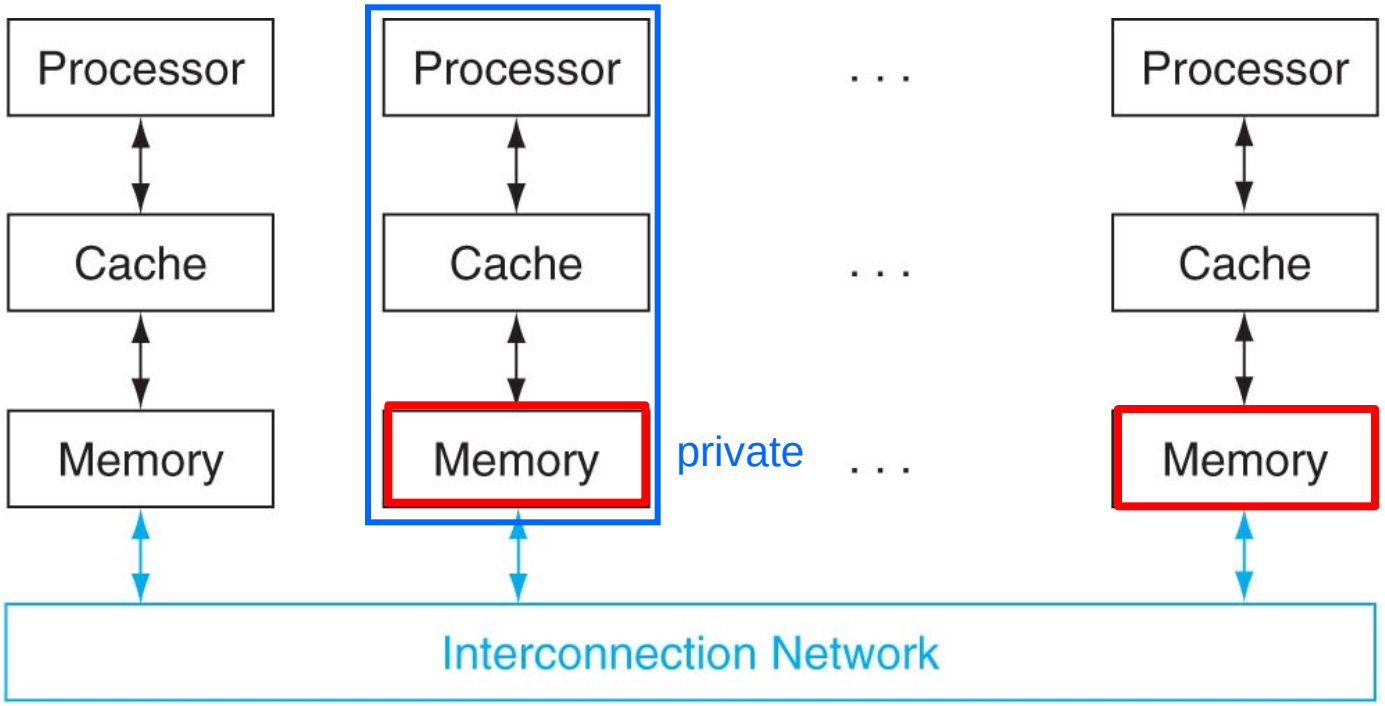
- Processors connected by a single bus
- Processors connected by a network

			# of Proc
Communication model	Message passing		8 to many
	Shared address	NUMA	8 to many
		UMA	2 to 64
Physical connection	Network		8 to many
	Bus		2 to 36

Message Passing & Clusters

- An alternative multiprocessor communicates via explicit message passing
  - Each processor has private physical address space
  - SW and HW interfaces for send/receive messages between processors
    - The message can be thought of as a remote procedure call.
- Some concurrent applications run well on parallel HW, independent of shared-address or message-passing
- Clusters: collections of computers connected via I/O over standard network switches to form a message-passing multiprocessors
  - Each runs a distinct copy of the operating system

A Message Passing Multiprocessor



Classic organization of a multiprocessor with multiple private address spaces, traditionally called a message-passing multiprocessor. Note that unlike the SMP, the interconnection network is not between the caches and memory but is instead between processor-memory nodes.

Clusters: Loosely Coupled

- Network of independent computers
  - Message passing parallel computer
  - 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 (prefer virtual machines) : n times
  - Low interconnect bandwidth, compared to memory bus bandwidth of an SMP
  - N independent memories and N OS copies

Memory Efficiency

- A single shared memory processor has 20 GB of main memory, five clustered computers each have 4 GB, and the OS occupies 1 GB.
- How much more space is there for users with shared memory?
  - A SMP = (20 -1) = 19 GB
  - The cluster = 5* (4-1) = 15 GB
  - The share memory computer has 4 GB more space than that of the cluster = 1.25X

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

```
100
 50
 25
 13
  7
  4
  2
  1
```

```
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

If there is an odd number of nodes, the middle node does not participate in send/receive

Sum Reduction (Again)

- half=50, receive:0,1,..,49 send: from 50,52,..,99 to 0,1,..,49
- half=25, receive:0,1,..,24 send: from 25,26,..,49 to 0,1,..,24
- half=13, receive:0,1,..,12 send: from 13,14,..,24 to 0,1,..,11
- half=7, receive:0,1,..,6 send: from 7,8,..,12 to 0,1,..,5
- half=4, receive:0,1,..,3 send: from 4,5,6 to 0,1,2
- half=2, receive:0,1 send: from 2,3 to 0,1
- half=1, recieve:0 send: from 1 to 0

```
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);  // stall until send
  if (Pn < (limit/2))      //
    sum = sum + receive(); //  stall until receive
  limit = half; /* upper limit of senders */
until (half == 1); /* exit with final sum */
```

Example: Elaboration

- In the previous example, message passing is assumed to be about as fast as addition.
- In reality, message sending and receiving is much slower
  - An optimization to better balance computation and communication might be to have fewer nodes receive more sums from other processors

Message Passing

- Much easier for hardware designer
  - Compared to implementation of cache coherent protocol
- Communication is explicit
  - Fewer performance surprise than with the implicit communication in cache-coherent shared memory computers
- Harder to port a sequential program to a message-passing computer
  - Since every communication must be identified in advance

Cluster

- A weakness of separate memory for user memory turns into a strength in system availability
  - Since the cluster software is a layer that runs on top of local OS running on each processor, it is much easier to disconnect and replace a broken machine.
- Given that clusters are constructed from whole computers and independent, scalable networks,
  - this isolation also make it easier to expand the system without bringing down the application that runs on top of the cluster.

Cluster

- The clusters are attractive to the service providers of the World Wide Web because of
  - low cost,
  - high availability,
  - improved power efficiency, and
  - rapid, incremental expandability
- The search engines depend on the clusters
- eBay, Google, Microsoft, Yahoo and other all have multiple datacenters each with clusters of tens of thousands of processors
  - The use of multiple processors in Internet services companies has been hugely successful.

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
  - SETI@home:
    - over 5 M computer users have signed
    - Operated at 257 TeraFLOPS by then end of 2006

Hardware Multithreading

- Hardware multithreading allows multiple threads to share the functional units of a single processor in a overlapping fashion.
  - Increase utilization of a processor by switching to another thread when one thread is stalled.
- Performing multiple threads of execution in parallel
  - Processor must duplicate the state of each thread (e.g., a copy of register file, a PC) and a separate page table for running independent programs)

Process Model (OS)

- Process model: two independent concepts
  - Resource grouping (PCB: process control block)
  - Execution (ready, running, blocked, exit)
- Processes are used to group resources together
- Threads are the entities scheduled for execution on the CPU
  - sometimes called lightweight processes.
- Multithreading is used to describe the situation of allowing multiple threads in the same process.

Process and Thread

- Per process items
  - Address space, Global variables
  - Open files, Child processes
  - Pending alarms: OS notifies after a specified time
  - Signals and signal handlers: A process handles signals just like OS does interrupts. A process can send signals only to members of its process group
  - Accounting information
- Per process items
  - Program counter
  - Registers
  - Stack
  - State: running, blocked, ready, or terminated

Beyond single thread ILP

- There can be much higher natural parallelism in some applications
  (e.g., Database or Scientific codes)
- Explicit Thread Level Parallelism or Data Level Parallelism
- Thread: instruction stream with own PC and data
  - thread may be a process part of a parallel program of multiple processes, or it may be an independent program
  - Each thread has all the state (instructions, data, PC, register state, stack, and so on) necessary to allow it to execute
- Data Level Parallelism: Perform identical operations on data, and lots of data

Thread Level Parallelism (TLP)

- ILP exploits implicit parallel operations within a loop or straight-line code segment
- TLP explicitly represented by the use of multiple threads of execution that are inherently parallel
- Goal: Use multiple instruction streams to improve
  - Throughput of computers that run many programs
  - Execution time of multi-threaded programs
- TLP could be more cost-effective to exploit than ILP

Hardware Multithreading

- Memory itself can be shared through the virtual memory mechanisms, which already support multiprogramming
- A thread switch should be much more efficient than a process switch.
  - A context switch: 100s to 1000s cycles
  - A thread switch: instantaneous with hardware support
- Two approaches to hardware multithreading
  - Fine-grained
  - Coarse-grained

Fine-Grained Multithreading

- Switches between threads on each instruction, causing the execution of multiples threads to be interleaved
- Usually done in a round-robin fashion, skipping any stalled threads
- CPU must be able to switch threads every clock

- Advantage: it can hide both short and long stalls.
- Disadvantage: it slows down execution of individual threads.
  - since a thread ready to execute without stalls will be delayed by instructions from other threads
- Used on Sun's Niagara

Coarse-Grained Multithreading

- Switches threads only on costly stalls
  - such as L2 cache misses
- Advantages
  - Relieves need to have very fast thread-switching
- Disadvantage
  - Hard to overcome throughput losses from shorter stalls, due to pipeline start-up costs
- Coarse-grained multithreading is better for reducing penalty of high cost stalls, where pipeline refill is much less than the stall time
- Used in IBM AS/400

Do both ILP and TLP? SMT

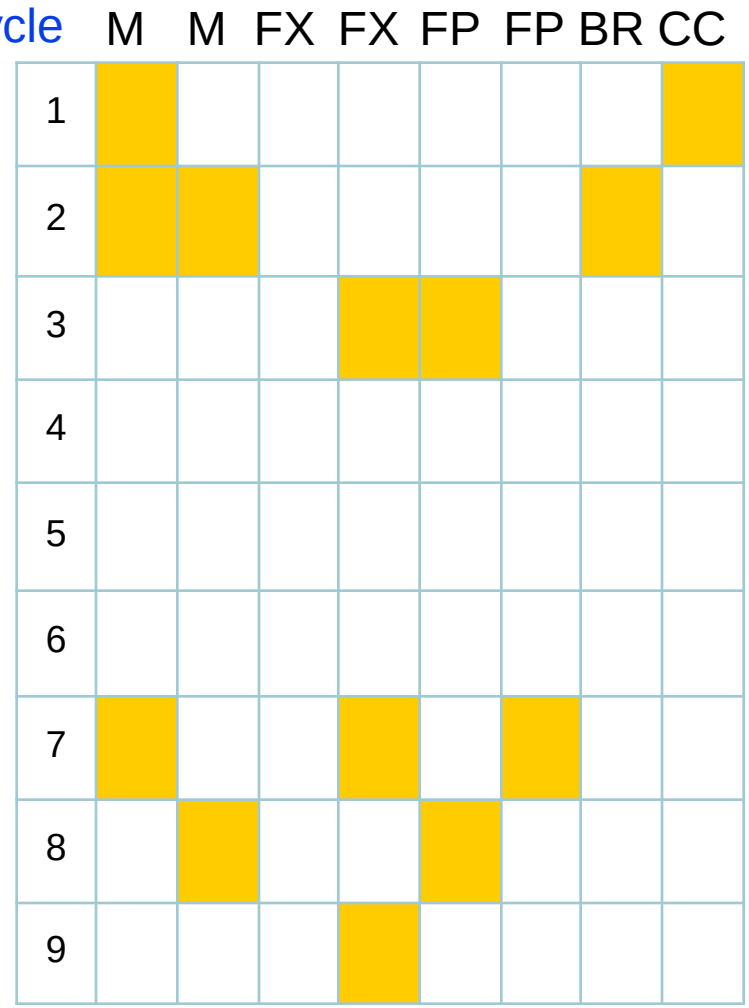
- TLP and ILP exploit two different kinds of parallel structure in a program
- Could a processor oriented at ILP to exploit TLP?
  - functional units are often idle in data path designed for ILP because of either stalls or dependences in the code
- Could the TLP be used as a source of independent instructions that might keep the processor busy during stalls?
- Could TLP be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?
- SMT is a variation of hardware multithreading that exploits both ILP and TLP

Simultaneous Multithreading

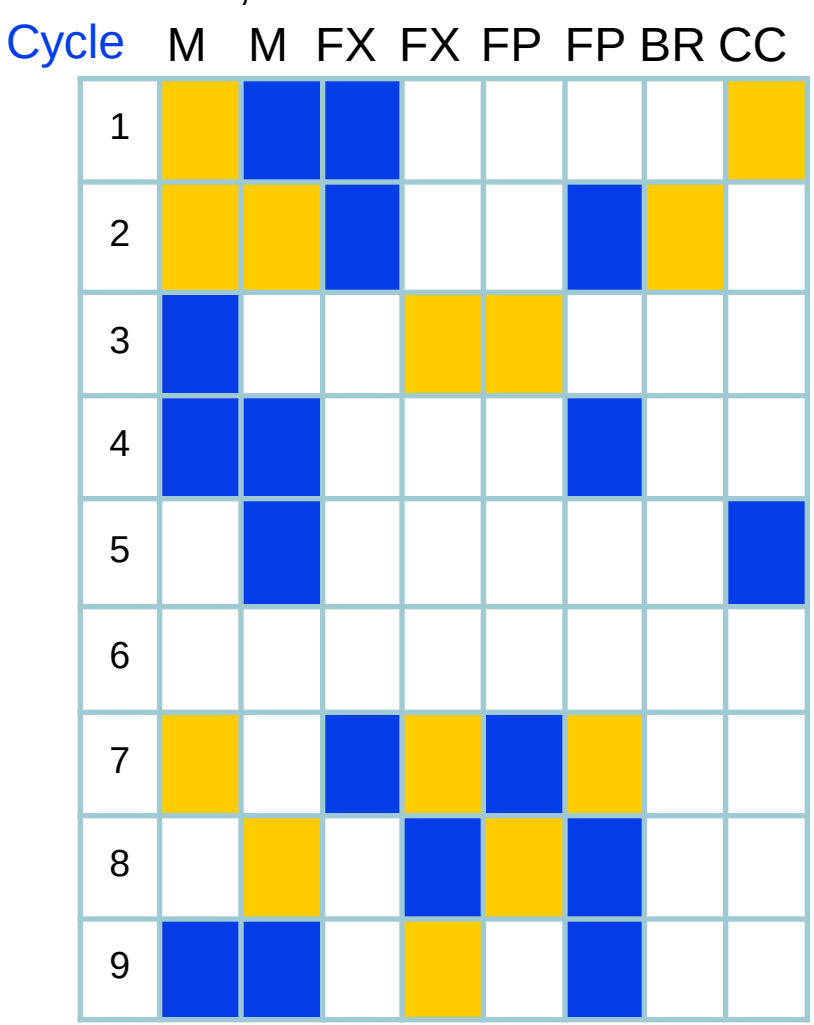
- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
  - Two threads: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

Simultaneous Multi-threading

One thread, 8 units



Two threads, 8 units

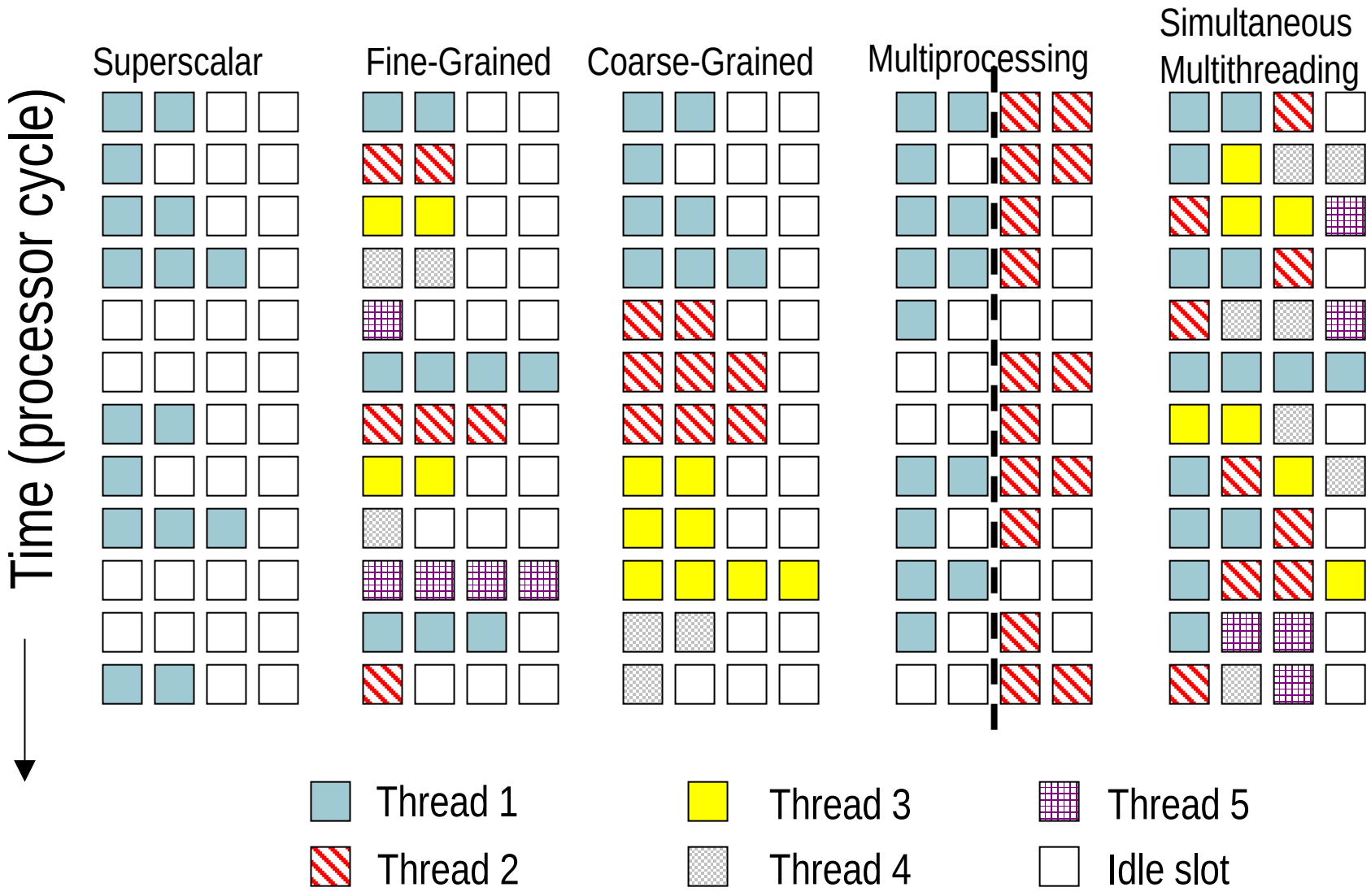


M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes

Why SMT?

- An insight that dynamically scheduled processor already has many HW mechanisms to support multithreading
  - Large set of virtual registers that can be used to hold the register sets of independent threads
  - Register renaming provides unique register identifiers
  - Out-of-order completion allows the threads to execute out of order, and get better utilization of the HW
- Just adding a per-thread renaming table and keeping separate PCs
  - Independent commitment can be supported by logically keeping a separate reorder buffer for each thread

Multithreaded Categories



Instruction and Data Streams

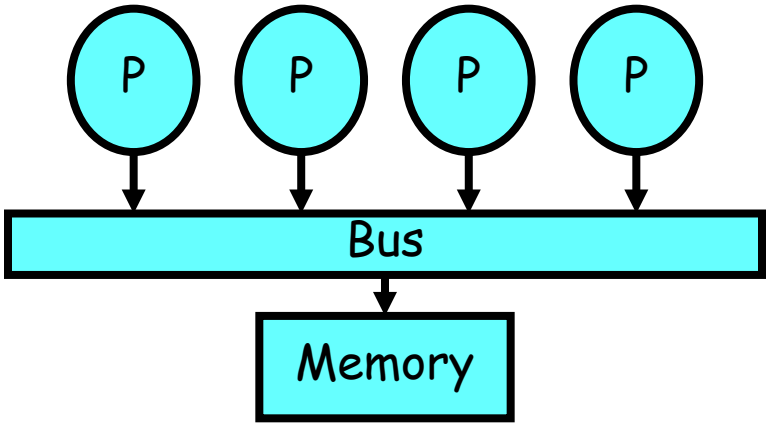
- Flynn's classification (1966)

		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 program runs across all processors
  - A parallel program on a MIMD computer
  - Conditional code for different processors

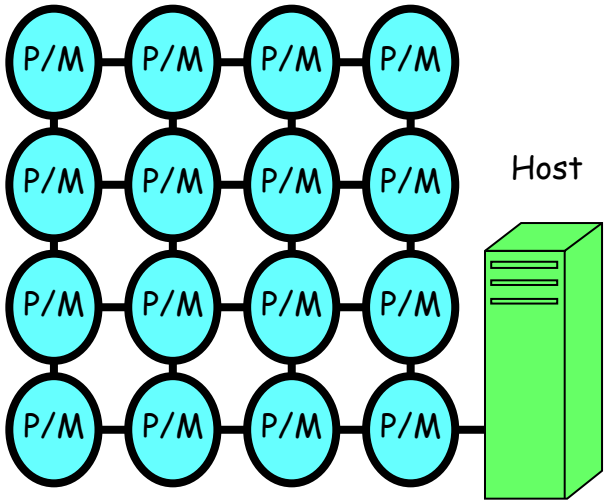
Examples of MIMD Machines

- Symmetric Multiprocessor
  - Multiple processors in box with shared memory communication
  - Current MultiCore chips like this
  - Every processor runs copy of OS



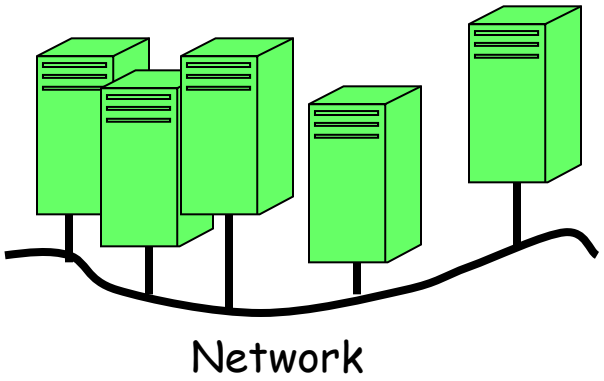
Examples of MIMD Machines

- Non-uniform shared-memory with separate I/O through host
  - Multiple processors
    - Each with local memory
    - general scalable network
  - Extremely light “OS” on node provides simple services
    - Scheduling/synchronization
  - Network-accessible host for I/O



Examples of MIMD Machines

- Cluster
  - Many independent machine connected with general network
  - Communication through messages



SIMD

- Operate element-wise on vectors of data
  - E.g., MMX and SSE instructions in x86
    - Multiple data elements in 128-bit wide registers
    - 128 = 8 x 16
  - MMX: multimedia extension
  - SSE: streaming SIMD extension
- 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

Vector Processors

- An older and more elegant interpretation of SIMD
  - Cray computers
- It is a great match to problems with lots of data parallelism
- Employs highly pipelined functional units
  - For example, rather than having 64 ALUs perform 64 addition simultaneously, like old array processors, the vector architectures pipelined ALU to get good performance with lower cost.

Vector Processors

- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Operate on them sequentially in registers
  - Results stored from registers to memory
- A key feature of vector architectures is a set of vector registers
  - 32 × vector registers (each: 64 64-bit elements)
- Example: Vector extension to MIPS
  - Vector instructions
    - `lv`, `sv`: load/store vector
    - `addv.d`: add vectors of double
    - `addvs.d`: add scalar to each element of vector of double

Example: DAXPY (Y = aX + Y)

- Conventional MIPS code

```
      l.d   $f0,a($sp)      ;load scalar a
      addiu r4,$s0,#512     ;upper bound: 64 x 8
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 ax
      lv      $v3,0($s1)   ;load vector y
      addv.d  $v4,$v2,$v3  ;add y to product
      sv      $v4,0($s1)   ;store the result to y
```

Comparison

- Significantly reduces instruction-fetch bandwidth
  - VMIPS: 6 instructions
  - MIPS: almost 600 instructions
  - This reduction saves power
- Frequency of pipeline hazards
  - In the MIPS code: two dependencies for each iteration for the loop
    - add.d must wait mul.d
    - s.d must wait add.d
  - In VMIPS: only for the first element in a vector
  - About 64x higher in the MIPS code
    - Can be reduced by using loop-unrolling, though.

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
```

Elaboration

- In the previous example, the loop size exactly matched the vector length (64).
- What if not matched?
- When loops are shorter
  - Vector architectures use a register that reduces the length of vector operations
- When loops are larger
  - We add bookkeeping code to iterate full-length vector operations and to handle the leftovers.
  - The latter process is called strip mining.

Vector vs. Scalar

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

Vector vs. Multimedia Extension

- The number of operations
  - X86 SSE: a few
  - Vector: dozens
- The number of elements in a vector operation is not in the opcode but in a separate register
- Data transfers
  - X86 SSE: need to be contiguous
  - Vector: support both strided and indexed accesses
- Flexibility in data widths in vector
  - 32 64-bit, 64 32-bit, 128 16-bit, 2556 8-bit
- Vector architecture: more efficient to execute data parallel processing programs.

Elaboration

- Given the advantages of vector, why aren't they more popular outside high-performance computing?
  - There were concerns about the larger state for vector registers increasing context switch time and difficulty of handling page faults in vector loads and stores
  - SIMD instructions achieved some of the benefits of instructions.
  - However, recently, Intel announced Advanced Vector Instruction (AVI) will expand the width of the SSE registers form 128 bits to 256 bits immediately and allow eventual expansion to 1024 bits (16 double-precision floating-point numbers)
  - Intel introduced a GPU named "Larrabee"

Elaboration

- Another advantage of vector and multimedia extensions is that it is relatively easy to extend a scalar instruction set architecture with these instructions to improve performance of data parallel operations.

Homework: chapter 7

- Due before starting the final exam on Dec. 8
- Exercise 7.8
- Exercise 7.10
- Exercise 7.16
- Exercise 7.19
- Exercise 7.23