Thread-Level Parallelism

Computer Organization

Wednesday, 09 October 2024

Many slides adapted from:
Computer Organization and Design,
Patterson & Hennessy
5th Edition, © 2014, MK
and from Prof. Mary Jane Irwin, PSU

TÉCNICO LISBOA

Summary

- Today
  - Multiprocessors
    - Programming overheads
    - Shared-memory multiprocessor
      - UMA vs NUMA
    - Memory coherence

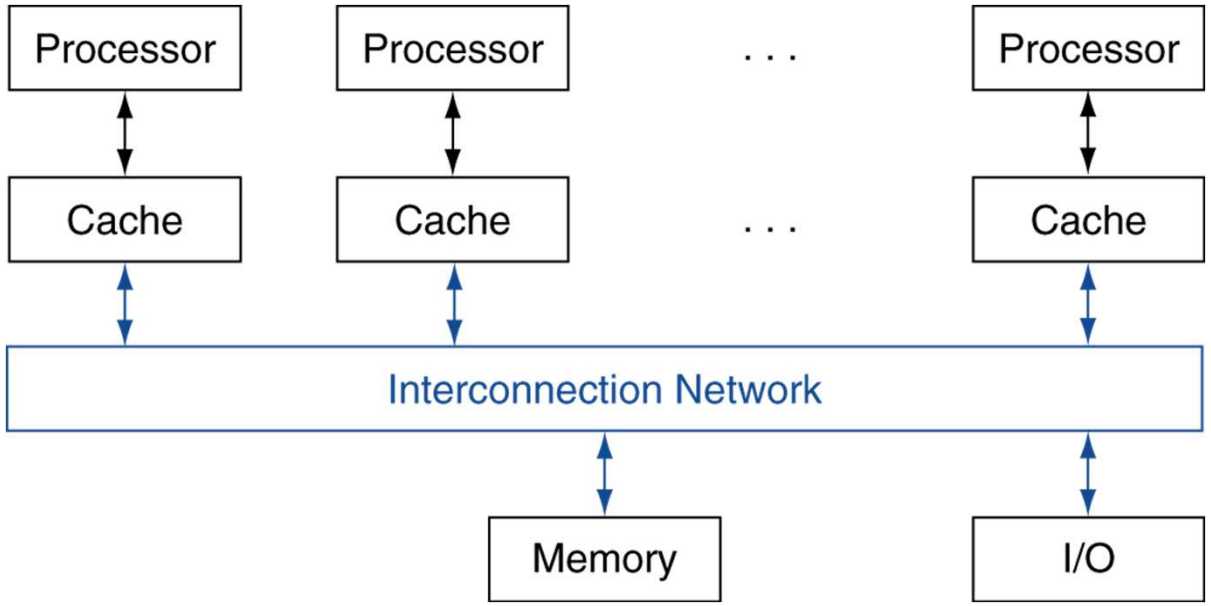
Forms of Parallelism

- Instruction-Level Parallelism
  - Pipelining
  - Multiple-issue processors

- I/O
  - DMA
  - RAID

Task-level Parallelism

- Multiprocessor – a computer system with at least two processors



Write-through
or
Write-back?

Minimize accesses to memory!

  - Can deliver high throughput for independent jobs via job-level parallelism or process-level parallelism
  - And improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel processing program

Multicores Now Common

- The power challenge has forced a change in the design of microprocessors
  - Since 2002 the rate of improvement in the response time of programs has slowed from a factor of 1.5 per year to less than a factor of 1.2 per year
- Today's microprocessors typically contain more than one core, Chip Multicore microProcessors (CMPs), in a single IC
  - The number of cores is expected to double every two years

Product	AMD Opteron	Intel Core i7	IBM Power 8	Sun Sparc T5
Cores per chip	16	10	12	16
Clock rate	2.4 GHz	3.5 GHz	4 GHz	3.6 GHz
Power	89W	140 W	100 W	94 W

Other Multiprocessor Basics

- Some of the problems that need higher performance can be handled simply by using a cluster – a set of independent servers (or PCs) connected over a local area network (LAN) functioning as a single large multiprocessor
  - Search engines, Web servers, email servers, databases, …
- A key challenge is to craft parallel (concurrent) programs that have high performance on multiprocessors as the number of processors increase – i.e., that scale
  - Scheduling, load balancing, time for synchronization, overhead for communication

Parallel Programming

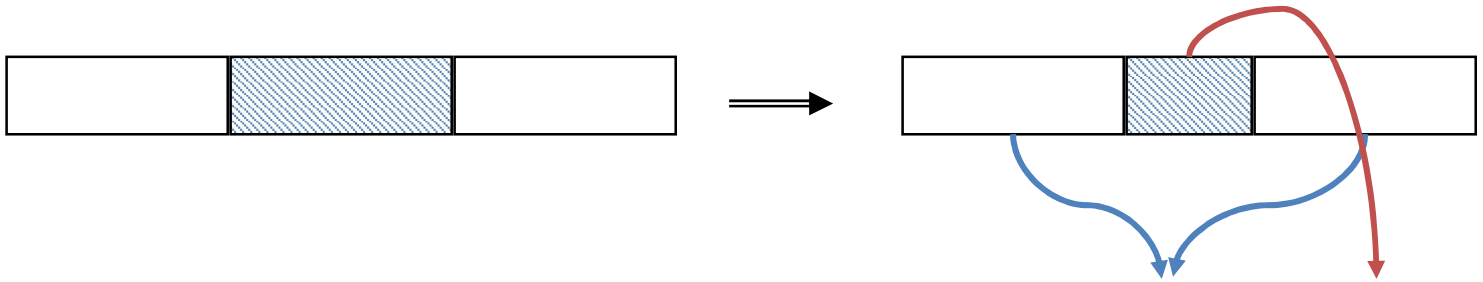
- Parallel software is the problem
- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it's easier!
- Difficulties
  - Partitioning
  - Coordination
  - Communications overhead

Encountering Amdahl's Law

- Speedup due to enhancement E is

$$\text{Speedup w/ E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/ E}}$$

  - Suppose that enhancement E accelerates a fraction F (F <1) of the task by a factor S (S>1) and the remainder of the task is unaffected



$$\text{ExTime w/ E} = \text{ExTime w/o E} \times ((1\text{-}F) + F/S)$$

$$\text{Speedup w/ E} = 1 / ((1\text{-}F) + F/S)$$

Example 1: Amdahl's Law

Speedup w/ E = 1 / ((1-F) + F/S)

- Consider an enhancement which runs 20 times faster but which is only usable 25% of the time.

  Speedup w/ E = 1/(.75 + .25/20) = 1.31

- What if its usable only 15% of the time?

  Speedup w/ E = 1/(.85 + .15/20) = 1.17

- Amdahl's Law tells us that to achieve linear speedup with 100 processors, none of the original computation can be scalar!
- To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less

  Speedup w/ E = 1/(.001 + .999/100) = 90.99

Example 2: Amdahl's Law

Speedup w/ E = 1 / ((1-F) + F/S)

- Consider summing 10 scalar variables and two 10 by 10 matrices (matrix sum) on 10 processors

  Speedup w/ E = 1/(.091 + .909/10) = 1/0.1819 = 5.5

- What if there are 100 processors?

  Speedup w/ E = 1/(.091 + .909/100) = 1/0.10009 = 10.0

- What if the matrices are 100 by 100 (or 10,010 adds in total) on 10 processors?

  Speedup w/ E = 1/(.001 + .999/10) = 1/0.1009 = 9.9

- What if there are 100 processors ?

  Speedup w/ E = 1/(.001 + .999/100) = 1/0.01099 = 91

Scaling

- To get good speedup on a multiprocessor while keeping the problem size fixed is harder than getting good speedup by increasing the size of the problem.
  - Strong scaling – when speedup can be achieved on a multiprocessor without increasing the size of the problem
  - Weak scaling – when speedup is achieved on a multiprocessor by increasing the size of the problem proportionally to the increase in the number of processors

- Load balancing is another important factor
  - Just a single processor with twice the load of the others cuts the speedup almost in half

Multiprocessor/Clusters Key Questions

- How is the workload distributed?
- How do tasks share data?
- How are tasks coordinated?
- How scalable is the architecture? How many processors can be supported?

These define program overheads:

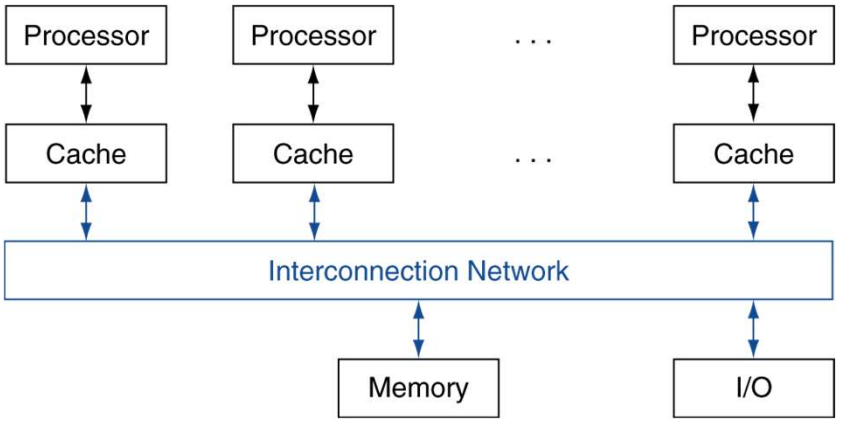
- Level of parallelism (ie, how many processors executing in parallel at any point)
- Communication / Synchronization
- Load balancing

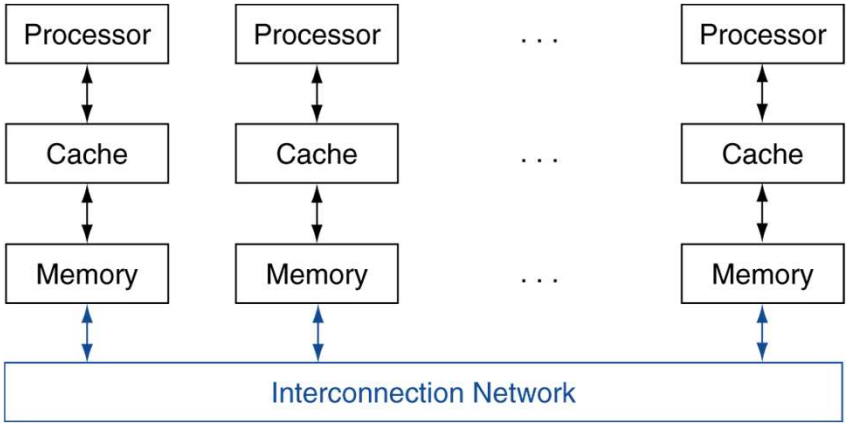
Shared Memory Multiprocessor (SMP)

- Single address space shared by all processors
- Processors coordinate/communicate through shared variables in memory (via loads and stores)
  - Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one processor at a time
- They come in two styles
  - Uniform memory access (UMA) multiprocessors
  - Nonuniform memory access (NUMA) multiprocessors

Shared Memory Multiprocessor (SMP)

- Uniform memory access (UMA) multiprocessors





- Nonuniform memory access (NUMA) multiprocessors

- Programming NUMAs is harder
- But NUMAs can scale to larger sizes and have lower latency to local memory

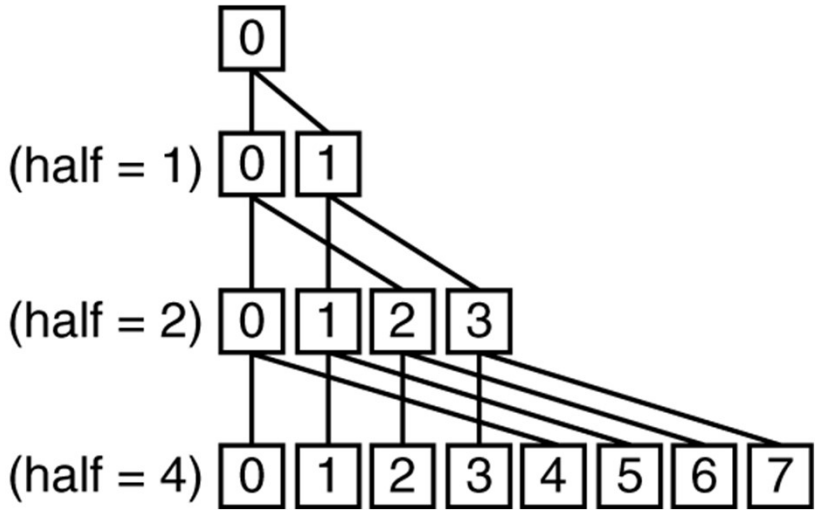
Example: Sum Reduction

- 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 partial sums
  - Reduction: divide and conquer
  - Half the processors add pairs, then quarter, …
  - Need to synchronize between reduction steps

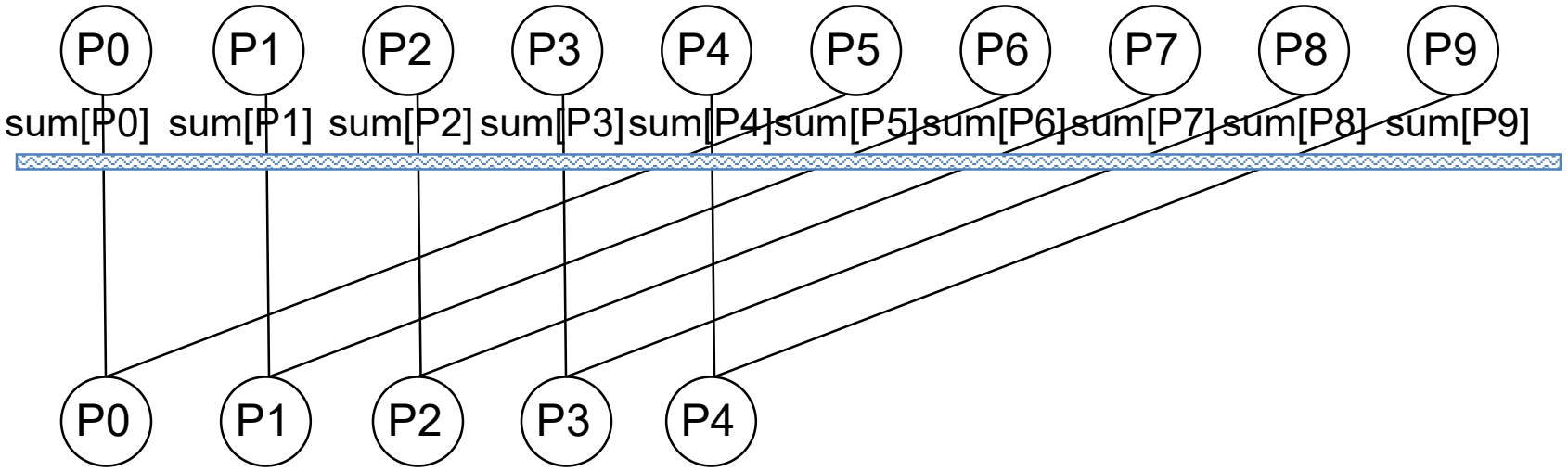
Example: Sum Reduction




```
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 missing element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```

An Example with 10 Processors

- `synch()`: Processors must synchronize before the "consumer" processor tries to read the results from the memory location written by the "producer" processor
  - Barrier synchronization: a synchronization scheme where processors wait at the barrier, not proceeding until every processor has reached it

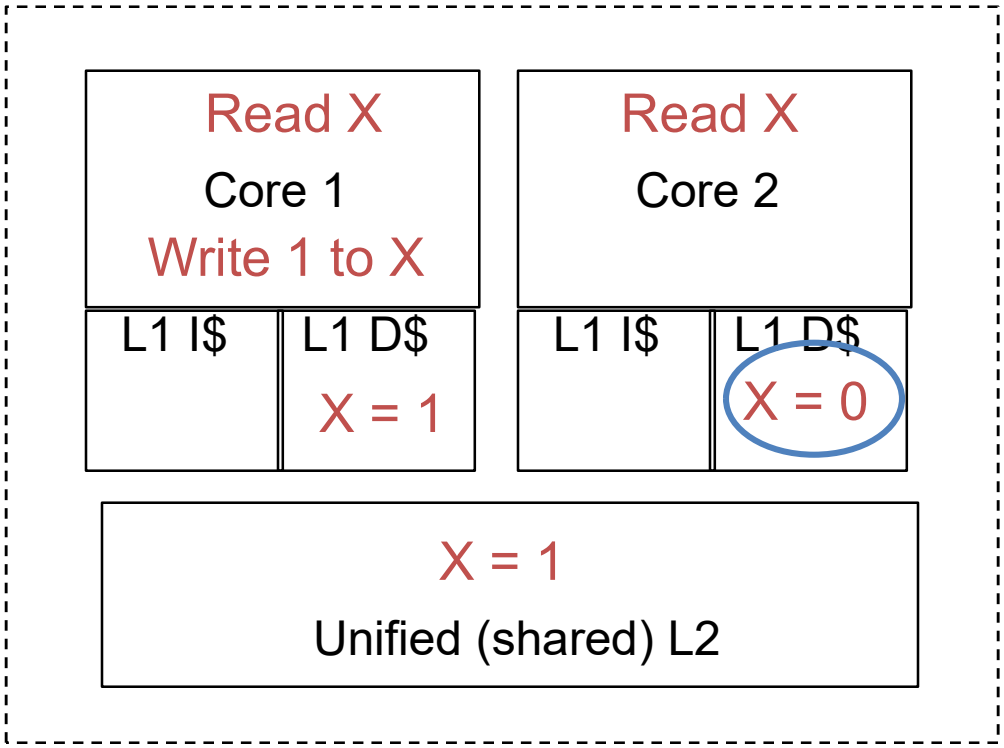


Thread Synchronization

- Need to coordinate threads working on a common task
- Lock variables (semaphores) are used to coordinate or synchronize threads
- Need an architecture-supported arbitration mechanism to decide which thread gets access to the lock variable
  - Single bus provides arbitration mechanism, since the bus is the only path to memory - the processor that gets the bus wins
- Need an architecture-supported operation that locks the variable
  - Locking can be done via an atomic swap operation (on the MIPS we have `ll` and `sc` one example of where a processor can both read a location and set it to the locked state – test-and-set – in the same bus operation)

Cache Coherence in SMPs

- Use of caches using a common physical address space creates a memory coherence problem



A Coherent Memory System

- Any read of a data item should return the most recently written value of the data item
  - Coherence: defines what values can be returned by a read
    - Writes to the same location are serialized (two writes to the same location must be seen in the same order by all cores)
  - Consistency: determines when a written value will be returned by a read
- Extra complexity is acceptable for parallel systems
  - Replication of shared data items in multiple cores' caches
    - Replication reduces both latency and contention for a read shared data item
  - Migration of shared data items to a core's local cache
    - Migration reduced the latency of the accessed data and the bandwidth demand on the shared memory (L2 in our example)

Cache Coherence Protocols

- Need a hardware protocol to ensure cache coherence the most popular of which is snooping
  - Cache controllers monitor (snoop) on the system bus to determine if their cache has a copy of a block that is requested

- Exclusive access ensure that no other readable or writable copies of an item exists
  - If two processors attempt to write the same data at the same time, one of them wins the race, thus enforcing write serialization

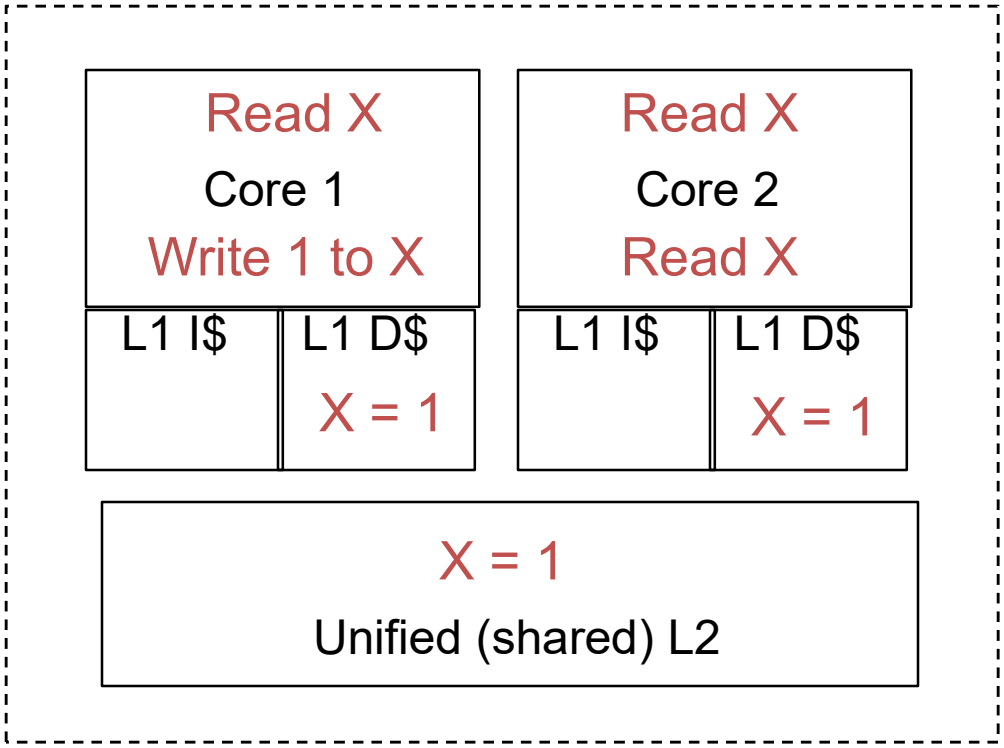
Cache Coherence Protocols

- Ensuring that all other processors sharing data are informed of writes can be handled two ways:

  1. Write-update (write-broadcast): writing processor broadcasts new data over the bus, all copies are updated
     - All writes go to the bus → higher bus traffic
     - Since new values appear in caches sooner, can reduce latency

  2. Write-invalidate: writing processor issues invalidation signal on bus, cache snoops check to see if they have a copy of the data, if so they invalidate their cache block containing the word (this allows multiple readers but only one writer)
     - Uses the bus only on the first write → lower bus traffic, so better use of bus bandwidth

Example of Snooping Invalidation



When the second miss by Core 2 occurs, Core 1 responds with the value canceling the response from the L2 cache (and also updating the L2 copy)

Multithreading

- Find a way to “hide” true data dependency stalls, cache miss stalls, and branch stalls by finding instructions (from 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
  - Caches and TLBs can be shared (although the miss rates may increase if they are not sized accordingly)
  - Memory can be shared through virtual memory mechanisms
  - Hardware must support *efficient* thread context switching

Multithreading

Thread 1

```
addi $s0, $s0, 8     F D X M W
lw   $t1, 0($s0)       F D X M W
add  $s1, $s1, $t1       F - D X M W
```

Thread 2

```
bne $t2, $t3, L8     F D X M W
sw $t0, 0($s0)         F - - D X M W
addi $s0, $s0, 8             F D X M W
```

```
addi $s0, $s0, 8       F D X M W
bne $t2, $t3, L8         F D X M W
lw $t1, 0($s0)             F D X M W
sw $t0, 0($s0)               F - D X M W
addi $s1, $s1, $t1             F D X M W
addi $s0, $s0, 8                 F D X M W
```

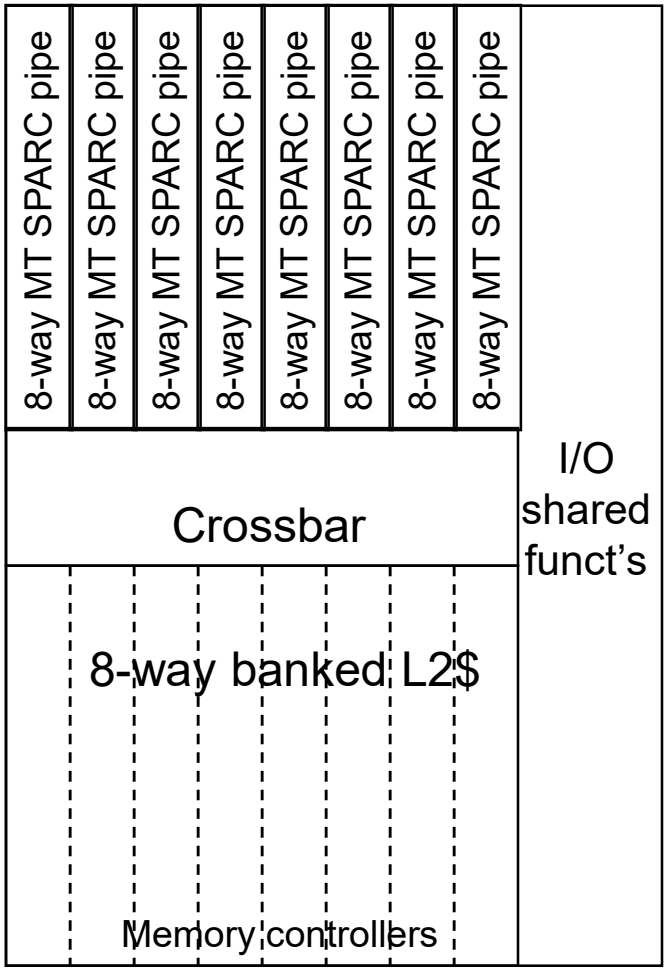
Types of Multithreading

- Fine-grain: switch threads on every instruction issue
  - Round-robin thread interleaving (skipping stalled threads)
  - Processor must be able to switch threads on every clock cycle
  - Advantage: can hide throughput losses that come from both short and long stalls
  - Disadvantage: slows down the execution of an individual thread since a thread that is ready to execute without stalls is delayed by instructions from other threads
- Coarse-grain: switches threads only on costly stalls (e.g., L2 cache misses)
  - Advantages: thread switching doesn't have to be essentially free and much less likely to slow down the execution of an individual thread
  - Disadvantage: limited, due to pipeline start-up costs, in its ability to overcome throughput loss
    - Pipeline must be flushed and refilled on thread switches

Multithreaded Example: Sun's Niagara (UltraSparc T2)

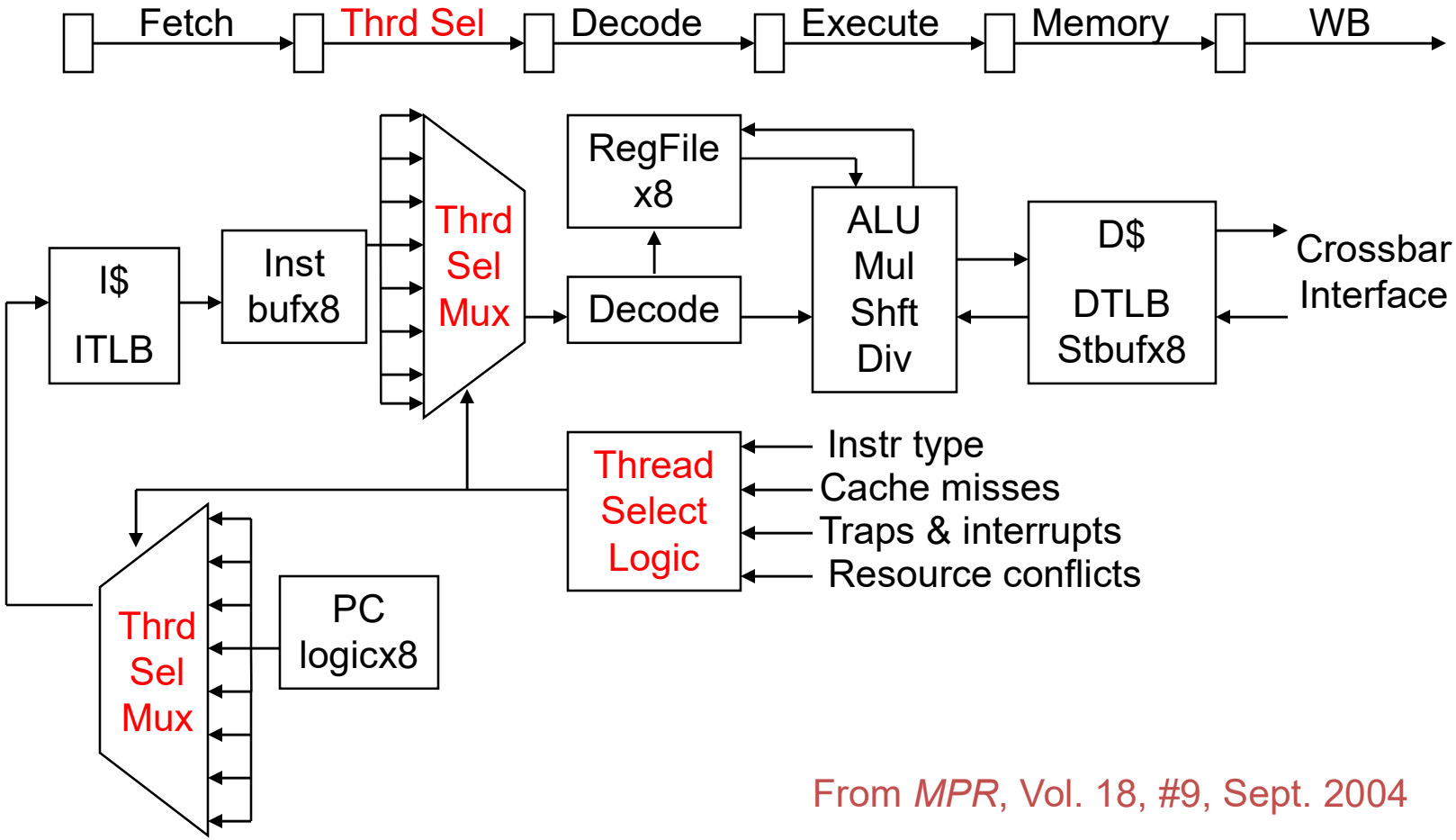
- Eight fine grain multithreaded single-issue, in-order cores (no speculation, no dynamic branch prediction)

	Niagara 2
Data width	64-b
Clock rate	1.4 GHz
Cache (I/D/L2)	16K/8K/4M
Issue rate	1 issue
Pipe stages	6 stages
BHT entries	None
TLB entries	64I/64D
Memory BW	60+ GB/s
Transistors	??? million
Power (max)	<95 W



Niagara Integer Pipeline

- Cores are simple (single-issue, 6 stage, no branch prediction), small, and power-efficient

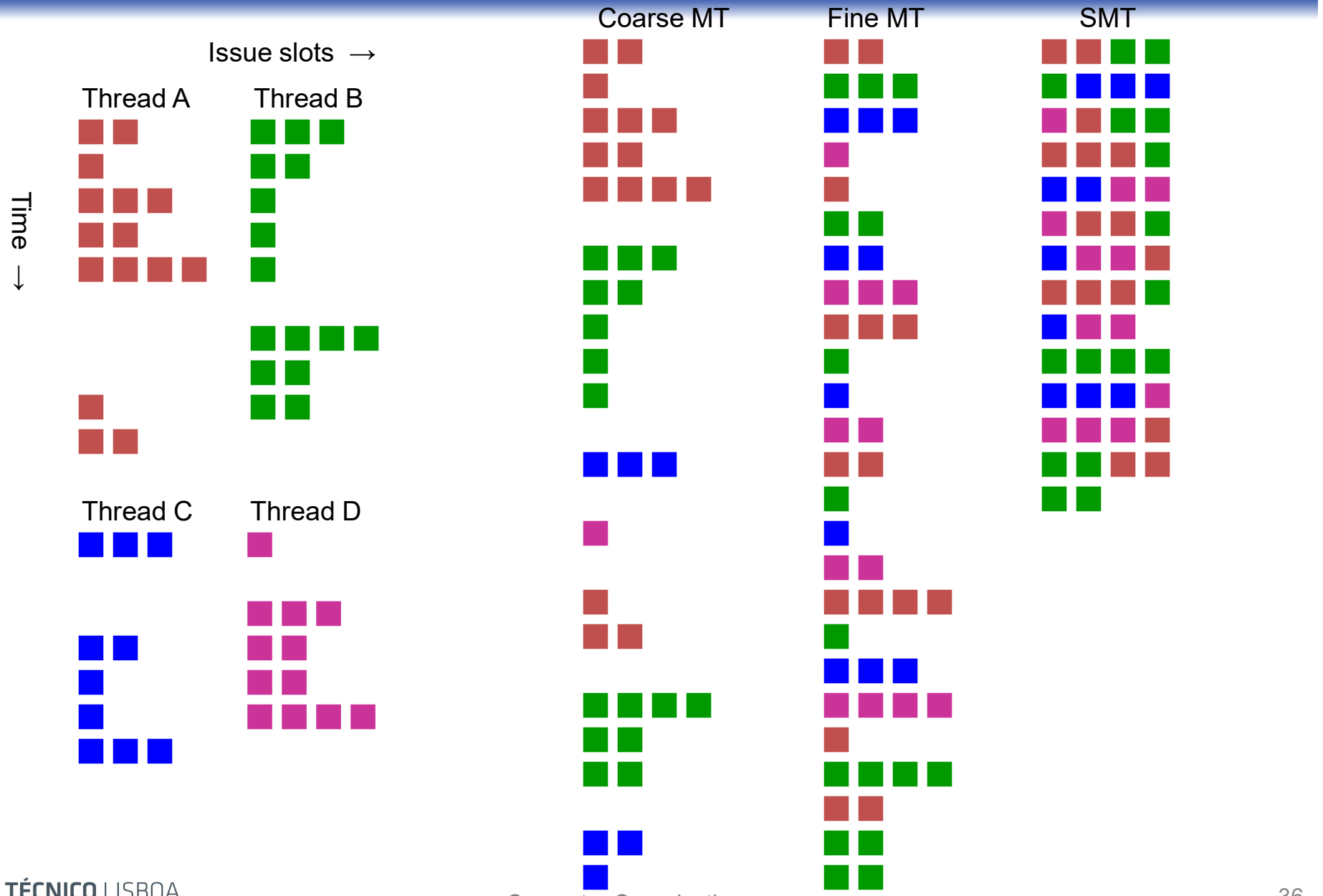


From *MPR*, Vol. 18, #9, Sept. 2004

Simultaneous Multithreading (SMT)

- Variation on multithreading that uses the resources of a multiple-issue, dynamically scheduled processor (superscalar) to exploit both program ILP and thread-level parallelism (TLP)
  - Most SS processors have more machine level parallelism than most programs can effectively use (i.e., than have ILP)
  - With register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to dependencies among them
    - Need separate rename tables for each thread or need to be able to indicate which thread the entry belongs to
    - Need the capability to commit from multiple threads in one cycle
- Intel's SMT is called hyperthreading
  - Supports just two threads (doubles the architecture state)

Threading on a 4-way SS Processor Example

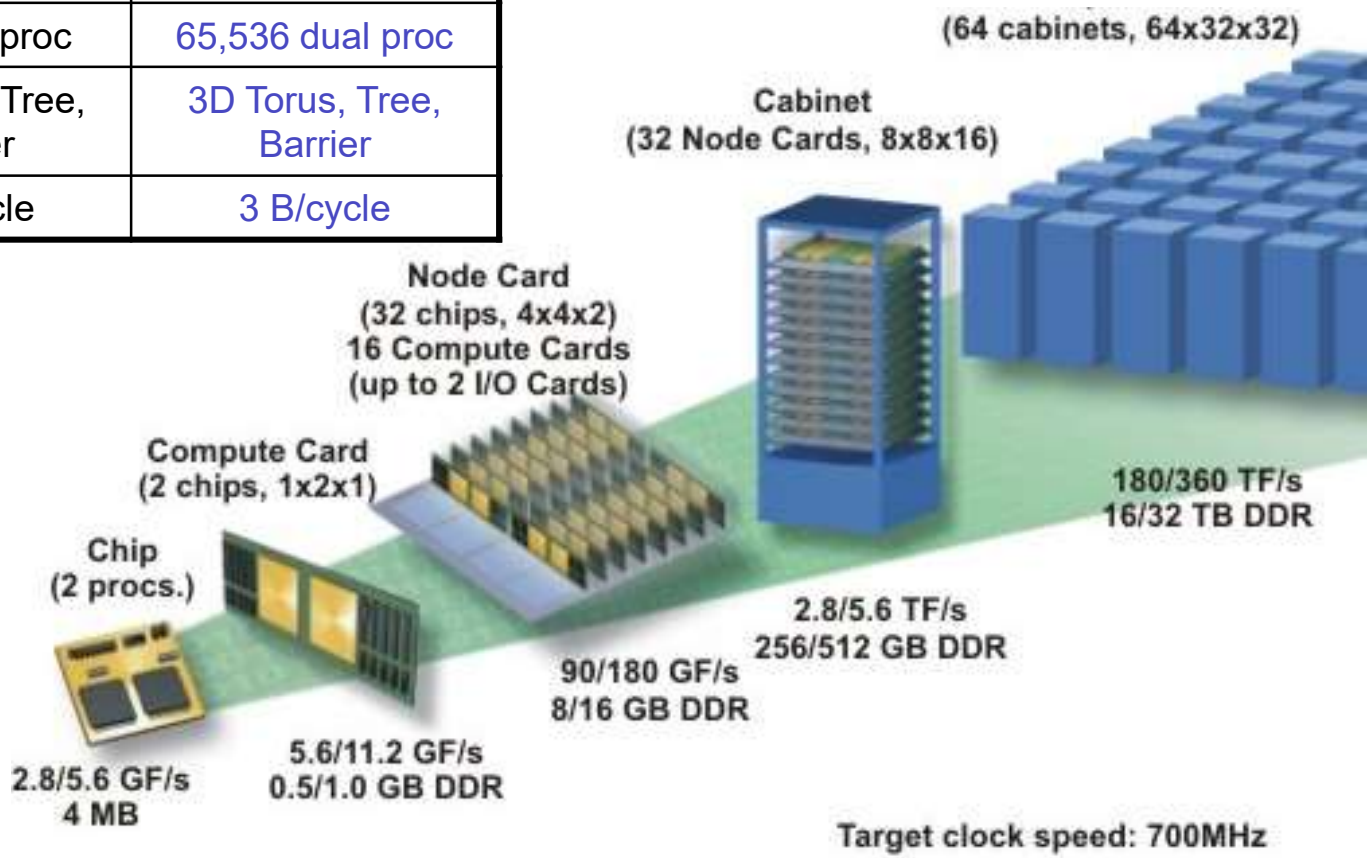


Network Connected Multiprocessors

	Proc	Proc Speed	# Proc	IN Topology	BW/link (MB/sec)
SGI Origin	R16000		128	fat tree	800
Cray 3TE	Alpha 21164	300MHz	2,048	3D torus	600
Intel ASCI Red	Intel	333MHz	9,632	mesh	800
IBM ASCI White	Power3	375MHz	8,192	multistage Omega	500
NEC ES	SX-5	500MHz	640 x 8	640-xbar	16,000
NASA Columbia	Intel Itanium2	1.5GHz	512 x 20	fat tree, Infiniband	
IBM BG/L	Power PC 440	0.7GHz	65,536 x 2	3D torus, fat tree, barrier	

IBM BlueGene

	512-node proto	BlueGene/L
Peak Perf	1.0 / 2.0 TFlops/s	180 / 360 TFlops/s
Memory Size	128 GByte	16 / 32 TByte
Foot Print	9 sq feet	2500 sq feet
Total Power	9 KW	1.5 MW
# Processors	512 dual proc	65,536 dual proc
Networks	3D Torus, Tree, Barrier	3D Torus, Tree, Barrier
Torus BW	3 B/cycle	3 B/cycle



Next Class

- Vector Processors
  - SIMD
  - GPUs

Thread-Level Parallelism

Computer Organization

Wednesday, 09 October 2024

Many slides adapted from:
Computer Organization and Design,
Patterson & Hennessy
5th Edition, © 2014, MK
and from Prof. Mary Jane Irwin, PSU

TÉCNICO LISBOA