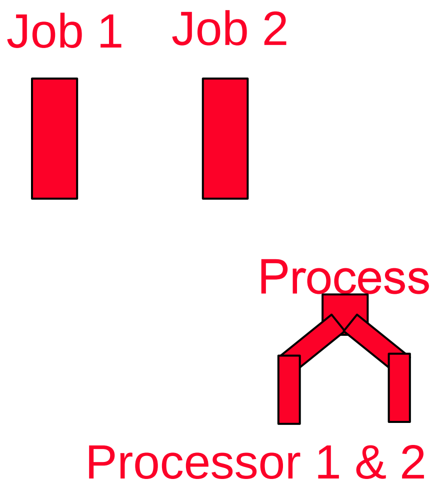
IT4272E-COMPUTER SYSTEMS

Chapter 7: Multicores, Multiprocessors, and Clusters

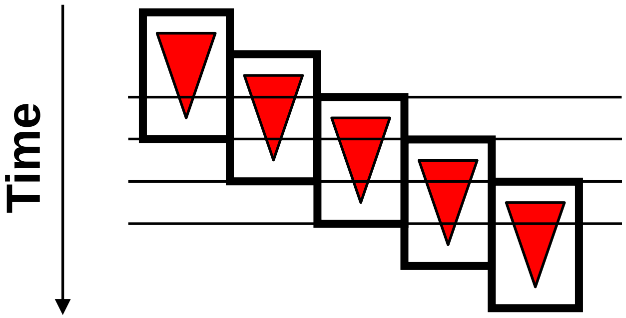
[with materials from *Computer Organization and Design, 4th Edition*, Patterson & Hennessy, © 2008, MK]

7.1. Introduction

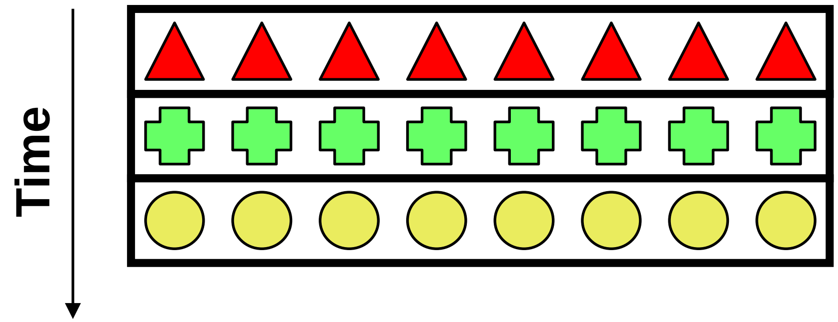
- Goal: connecting multiple computers to get higher performance
  - Multiprocessors
  - Scalability, availability, power efficiency
- Job-level (process-level) parallelism
  - High throughput for independent jobs
- Parallel processing program
  - Single program run on multiple processors
- Multicore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



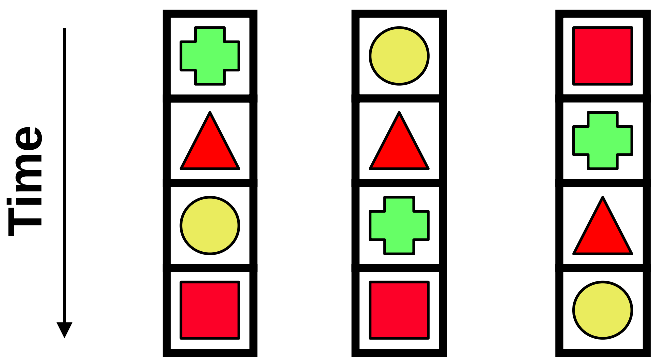
Types of Parallelism



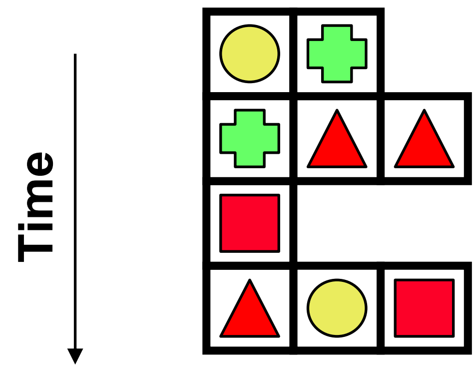
Pipelining



Data-Level Parallelism (DLP)



Thread-Level Parallelism (TLP)



Instruction-Level Parallelism (ILP)

Hardware and Software

Hardware

- Serial: e.g., Pentium 4

- Parallel: e.g., quad-core Xeon e5345

Software

- Sequential: e.g., matrix multiplication
- Concurrent: e.g., operating system

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

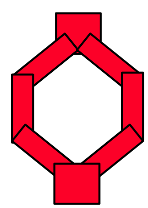
Quard Core

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

7.2. The Difficulty of Creating Parallel Processing Programs

- 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



Amdahl's Law

- Sequential part can limit speedup
- Example: 100 processors, 90× speedup?
  - $T_{old} = T_{parallelizable} + T_{sequential}$
  - $T_{new} = T_{parallelizable}/100 + T_{sequential}$
  - $$\text{Speedup} = \frac{1}{(1 - F_{parallelizable}) + F_{parallelizable}/100} = 90$$
  - 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 (5.5/10 = 55% of potential)
- **100 processors**
  - Time = 10 × t_{add} + 100/100 × t_{add} = 11 × t_{add}
  - Speedup = 110/11 = 10 (10/100 = 10% of potential)
- Assumes load can be balanced across processors

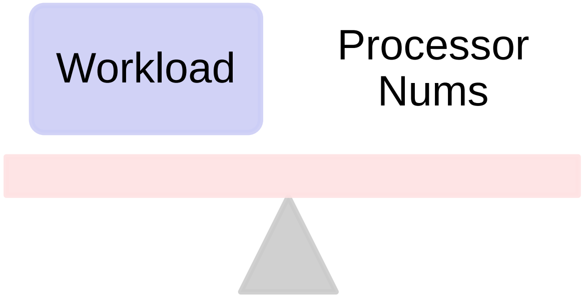
Scaling Example (cont)

- 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

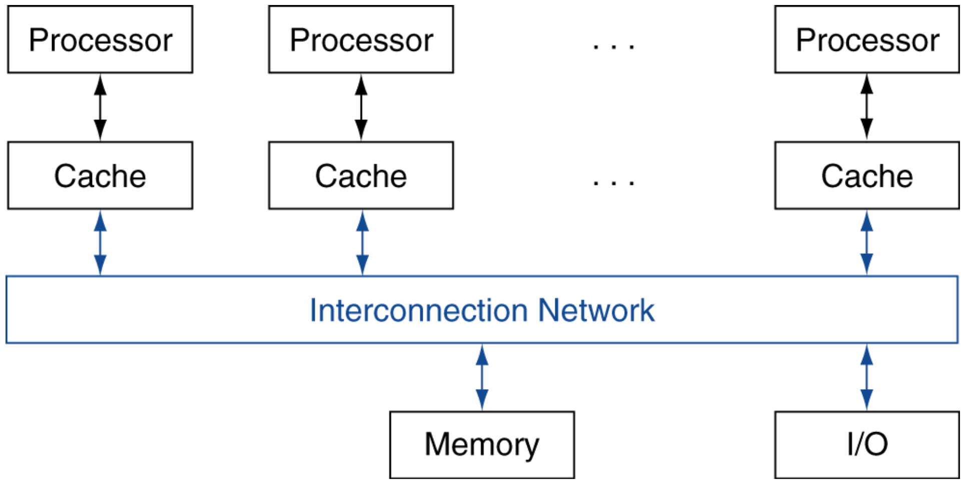
- Strong scaling: problem size fixed
  - As in example

- Weak scaling: problem size proportional to number of processors
  - 10 processors, 10 × 10 matrix
    - Time = $20 \times t_{add}$
  - 100 processors ↑, 32 × 32 matrix ↑
    - Time = $10 \times t_{add} + 1000/100 \times t_{add} = 20 \times t_{add}$
  - Constant performance in this example



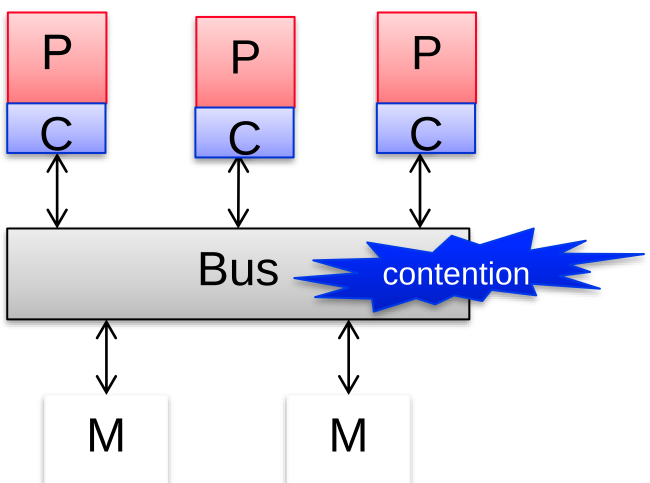
7.3. Shared Memory Multiprocessors

- SMP: shared memory multiprocessor
  - Hardware provides single physical address space for all processors
  - Synchronize shared variables using locks
  - Usually adapted in general purpose CPU's in laptops and desktops
- Memory access time: UMA vs NUMA

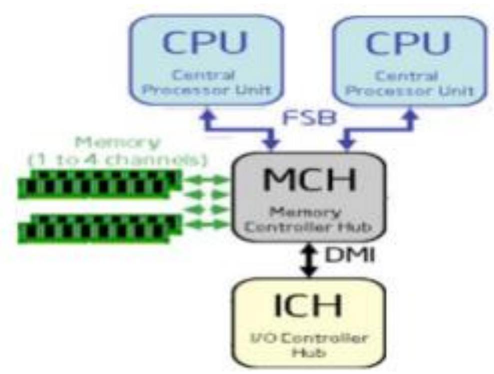


Shared Memory Arch: UMA

- access time to a memory location is independent of which processor makes the request, or which memory chip contains the transferred data.
- Used for a few processors.

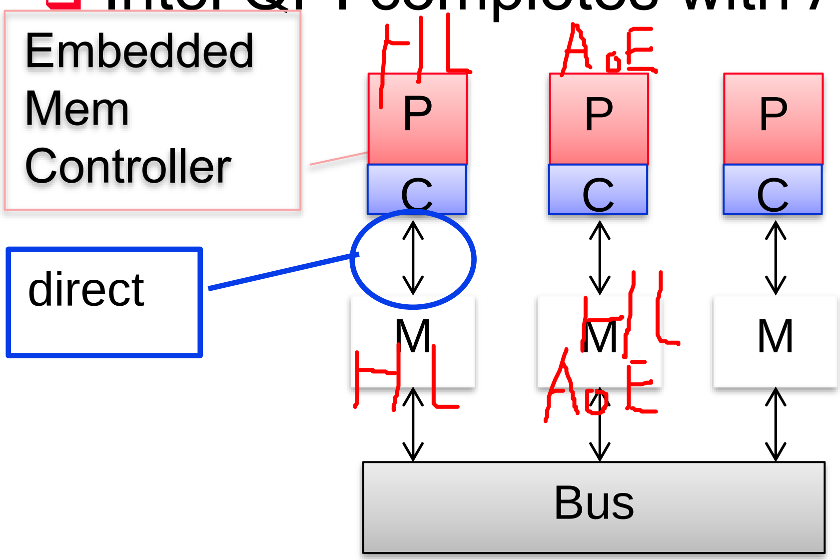


Intel's FSB based UMA Arch

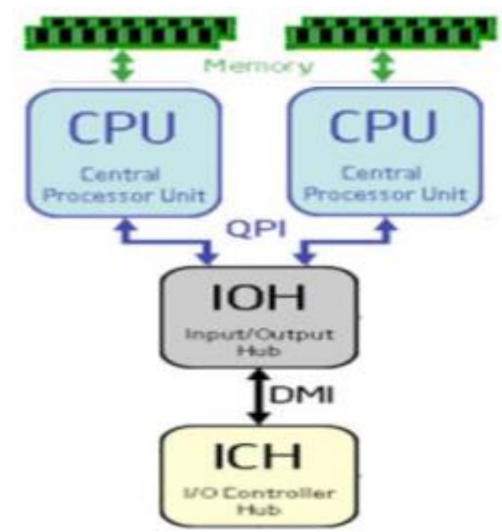


Shared Memory Arch: NUMA

- access time depends on the memory location relative to a processor.
- Used for dozens, hundreds of processors
- Processors use the same memory address space. Distributed Shared Memory, DSM
- Intel QPI completes with AMD HyperTransport, not bus.

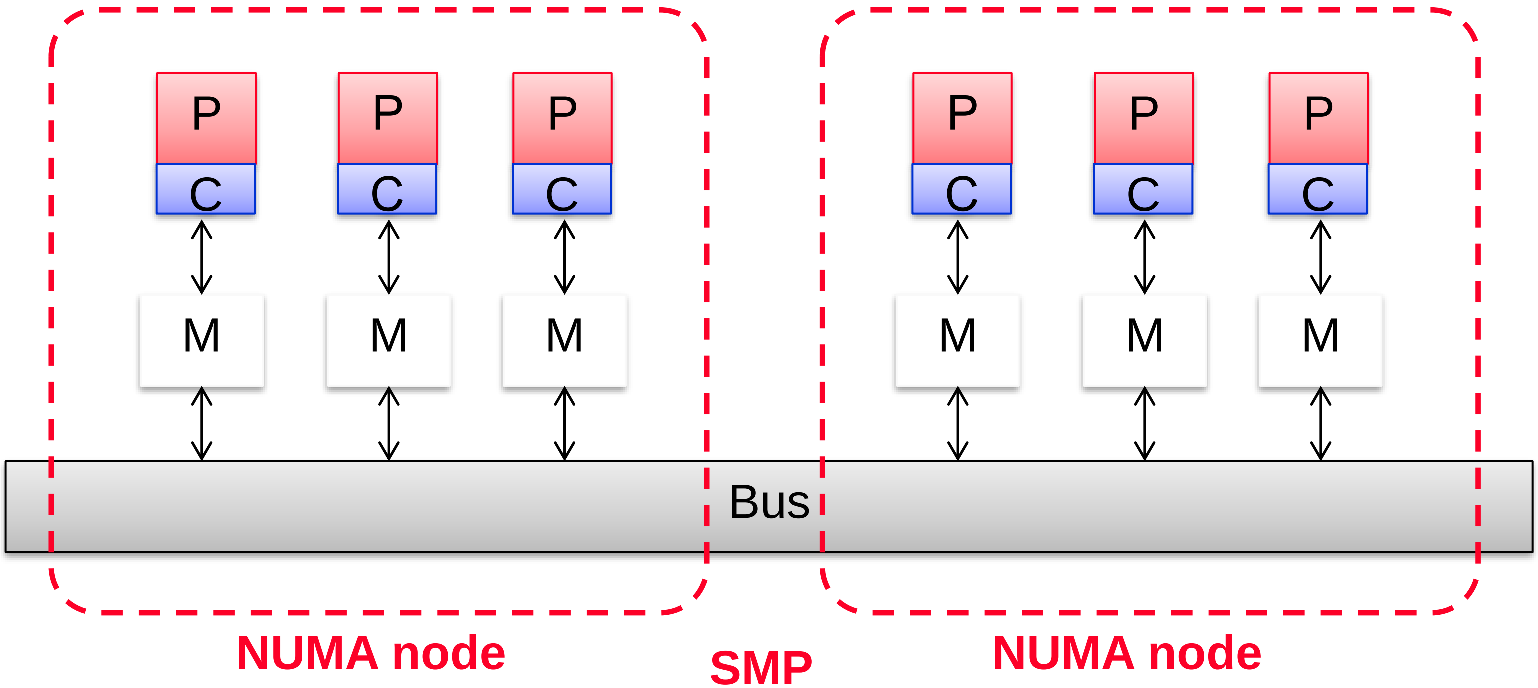


Quick Path Interconnect Arch

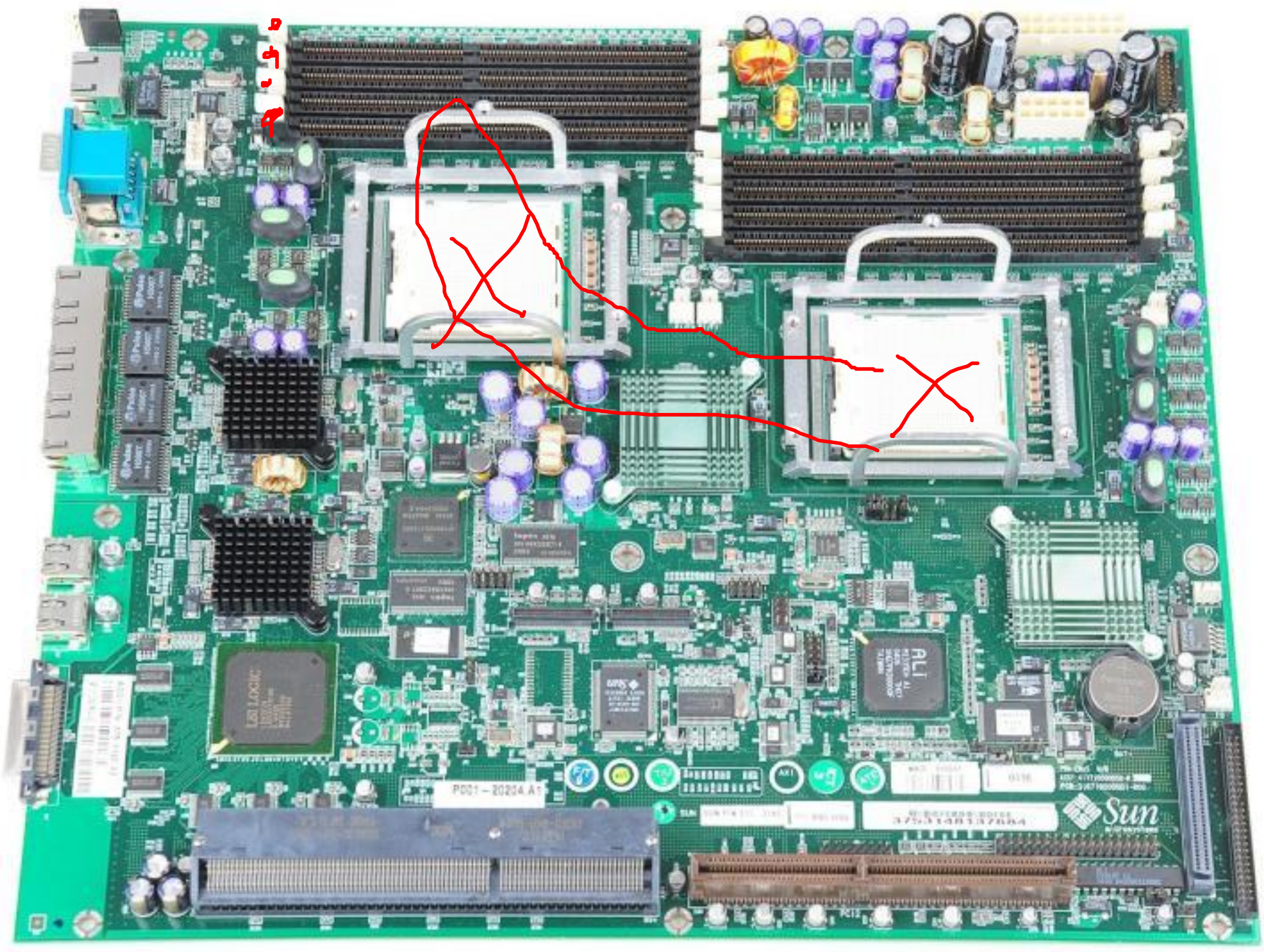


Shared Memory Arch: NUMA

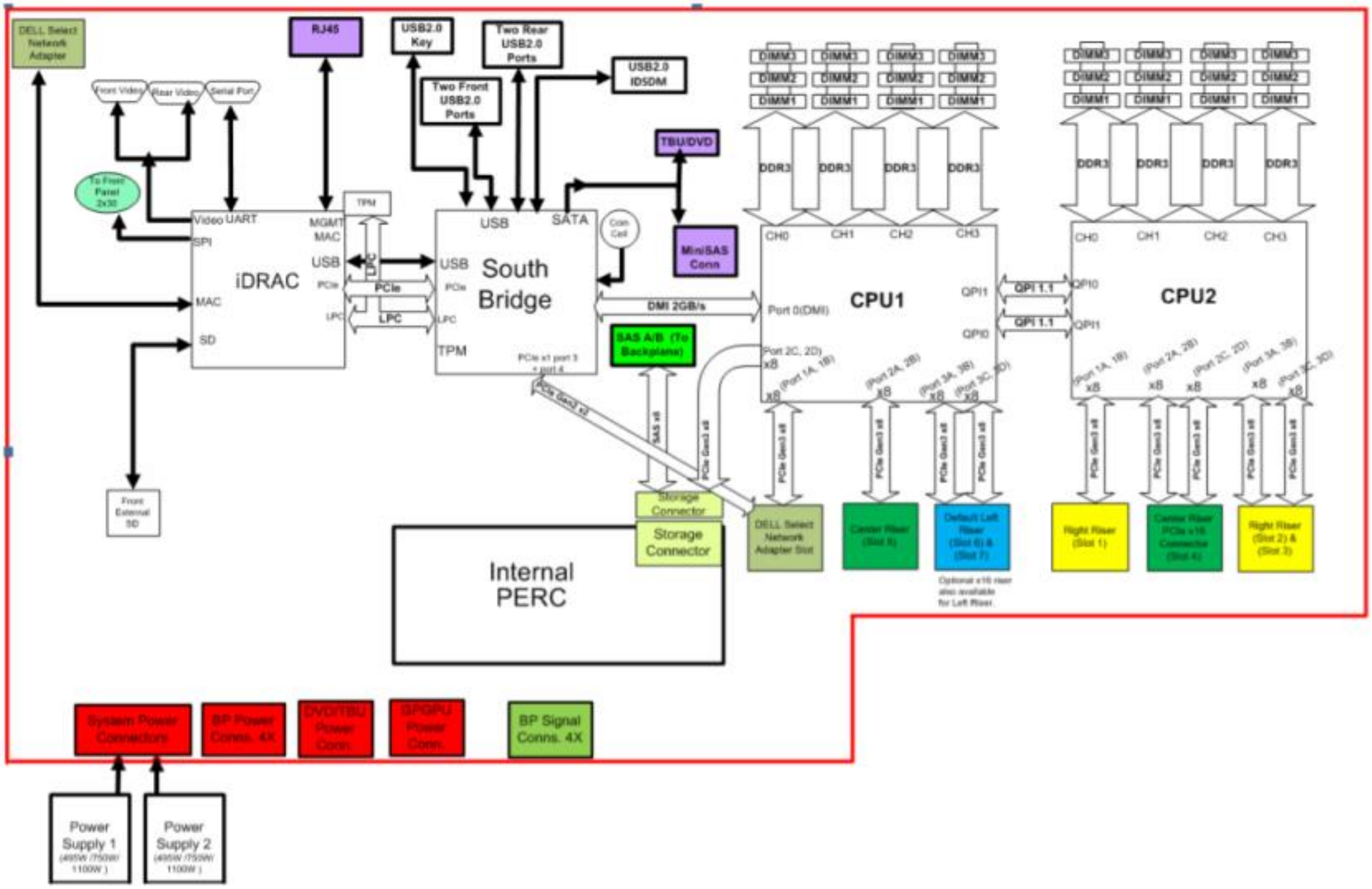
- Eg. the memory manager of programming languages also need to be NUMA aware. Java is NUMA aware.
- Eg. Oracle 11g explicitly enabled for NUMA support
- Eg. Windows XP SP2, Server 2003, Vista supported NUMA



Example: Sun Fire V210 / V240 Mainboard

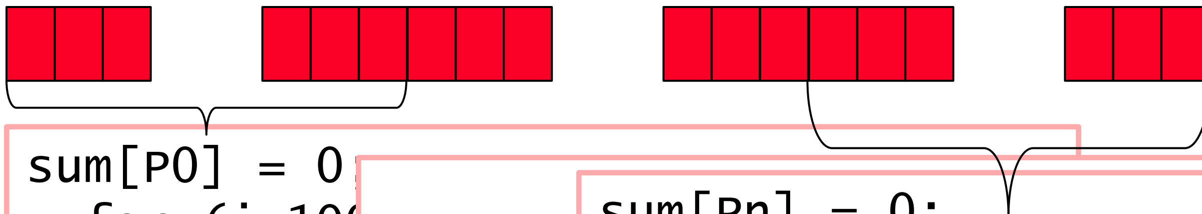


Example: Dell PowerEdge R720



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



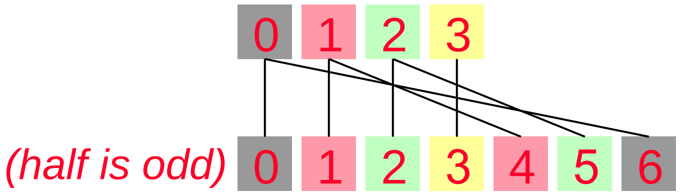
100,000 numbers

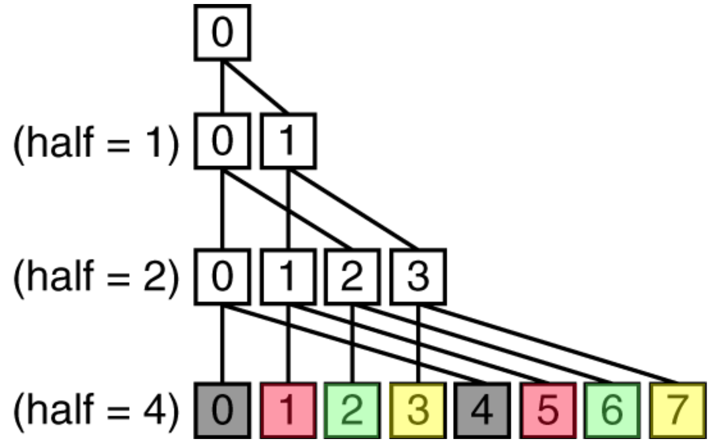
```
sum[P0] = 0;
  for (i=100
    sum[P0]
```

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

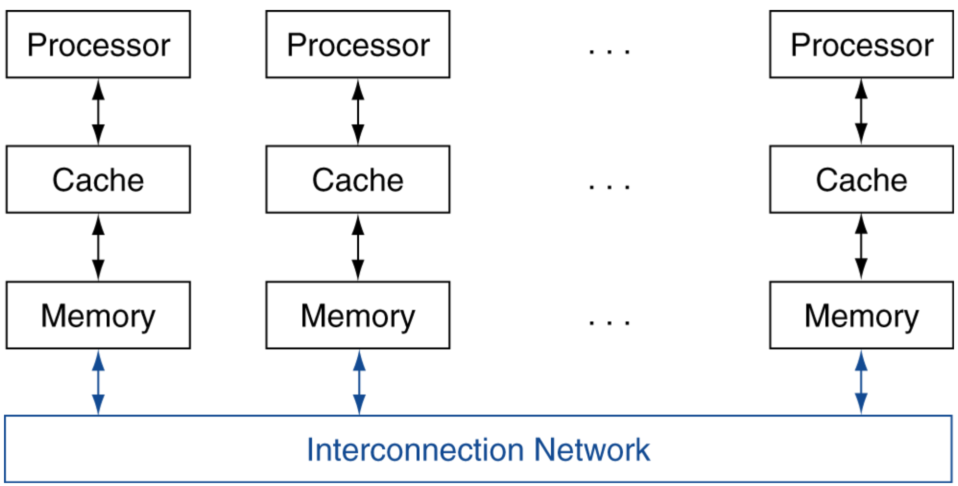


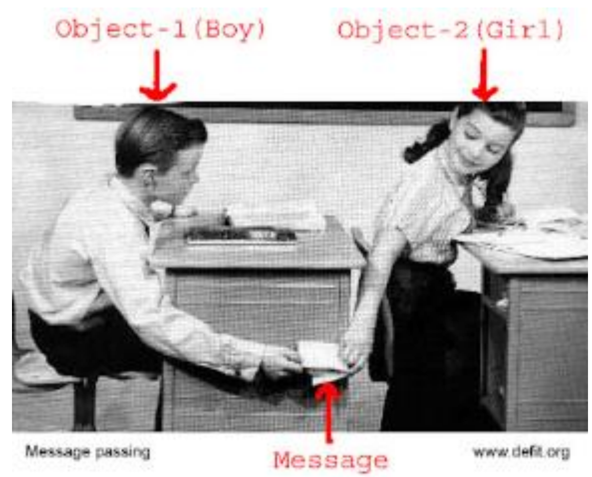



```
half = 100;
repeat
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets missing element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
```

7.4. Clusters and Other Message-Passing Multiprocessors

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





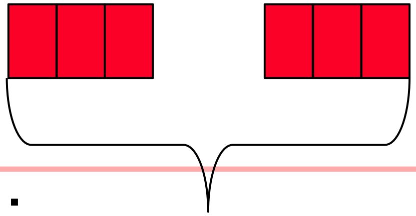
Loosely Coupled Clusters

- 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 (prefer virtual machines)
  - Low interconnect bandwidth
    - c.f. processor/memory bandwidth on an SMP

Sum Reduction (Again)

- Sum 100,000 on 100 processors
- First distribute 1000 numbers to each
  - The do partial sums



1,000 numbers

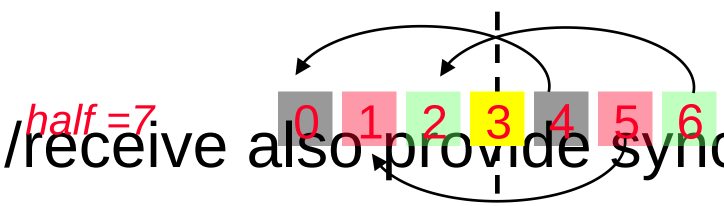
```
sum = 0;
  for (i=0;i<1000;i++)
    sum = sum + AN[i];
```

- Reduction
  - Half the processors s
  - 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

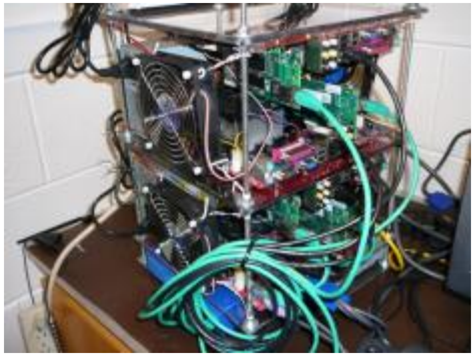
Message Passing Systems

- ONC RPC, CORBA, Java RMI, DCOM, SOAP, .NET Remoting, CTOS, QNX Neutrino RTOS, OpenBinder, D-Bus, Unison RTOS

- Message passing systems have been called "shared nothing" systems (each participant is a black box).

- Message passing is a type of communication between processes or objects in computer science

- Opensource: Beowulf, Microwulf



Microwulf

Beowulf

A sending-message cluster

7.5. Hardware Multithreading

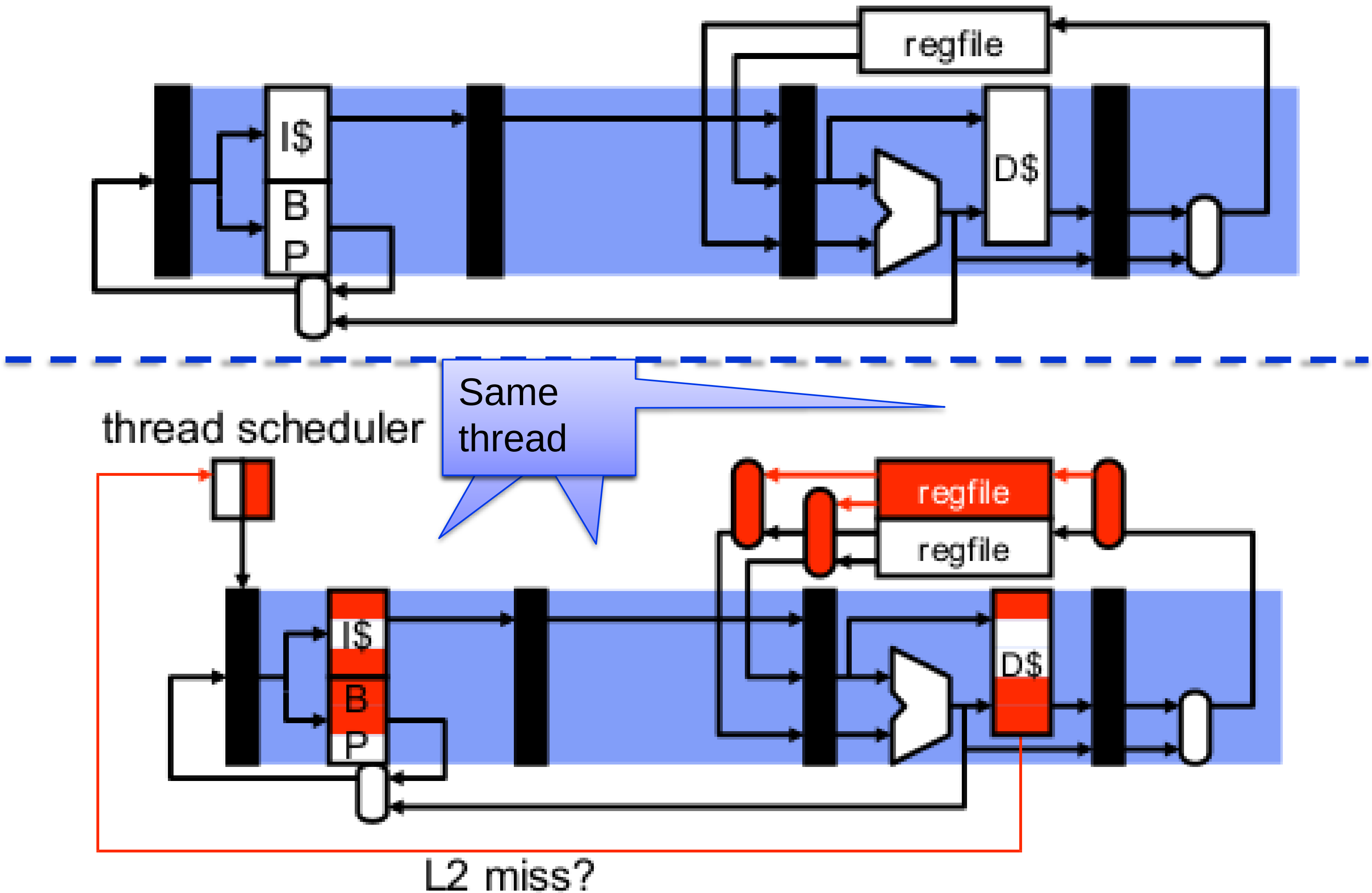
- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads
- Three designs:
  - Coarse-grain multithread, CMT
  - Fine-grain multithread, FMT
  - Simultaneous multithread, SMT

Coarse: Thiết kế tồi, kém, đơn giản
Fine: thiết kế tốt
Simultaneous: Đồng thời

Coarse-grain multithreading

- Only switch on long stall (e.g., L2-cache miss)
- Simplifies hardware, but doesn’t hide short stalls (eg, data hazards)
- Thread scheduling policy
  - Designate a “preferred” thread (e.g., thread A)
  - Switch to thread B on thread A L2 miss
  - Switch back to A when A L2 miss returns
- Sacrifices very little single thread performance (of one thread)
- Example: IBM Northstar/Pulsar

Coarse-grain multithreading

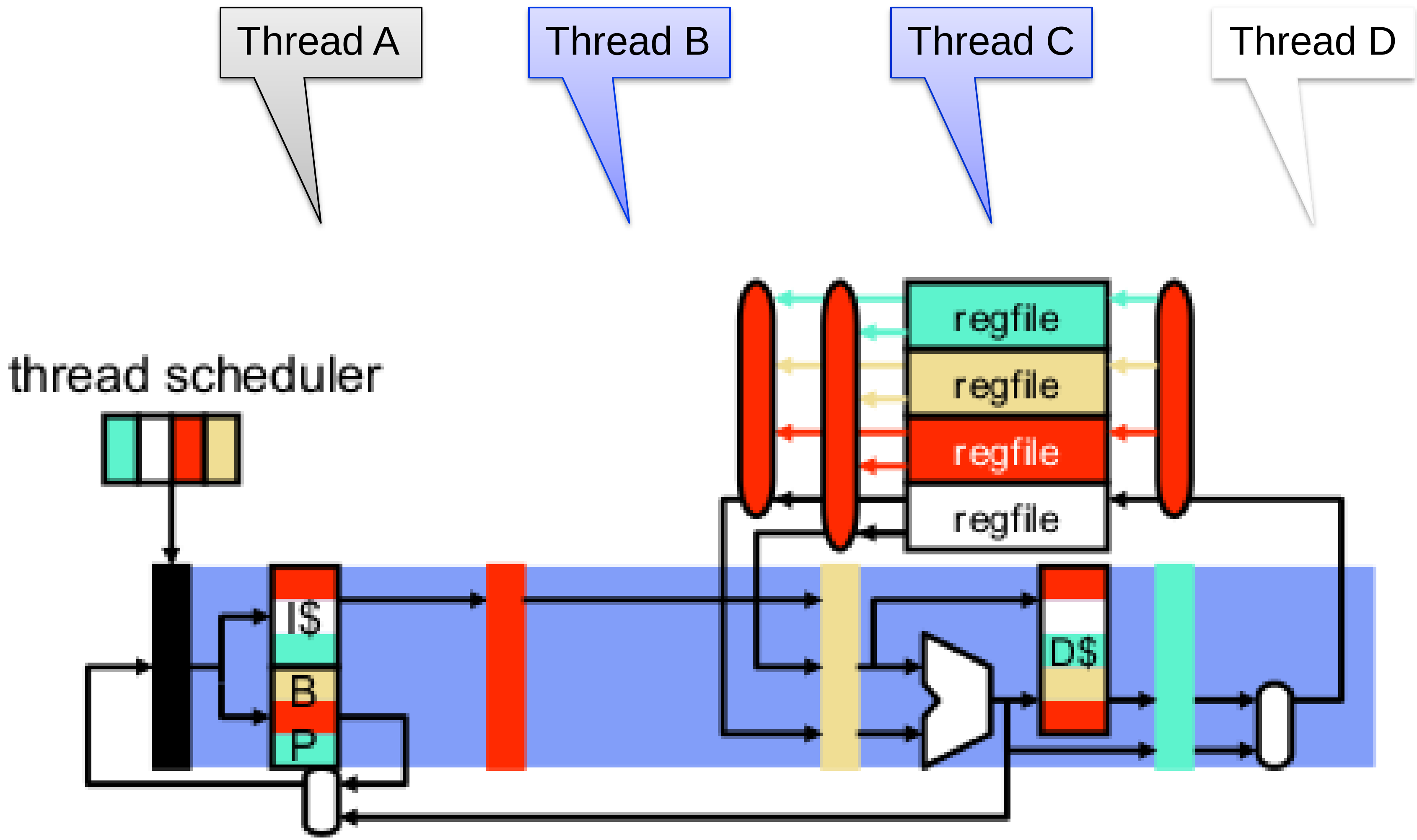


Fine-grain multithread

- Switch threads after each cycle (round-robin), L2 miss or no.
- Interleave instruction execution
- If one thread stalls, others are executed
- Sacrifices significant single thread performance
- Need a lot of threads

- Not popular today
  - Many threads ! many register files
- Extreme example: Denelcor HEP
  - So many threads (100+), it didn't even need caches
  - Failed commercially

Fine-grain multithread



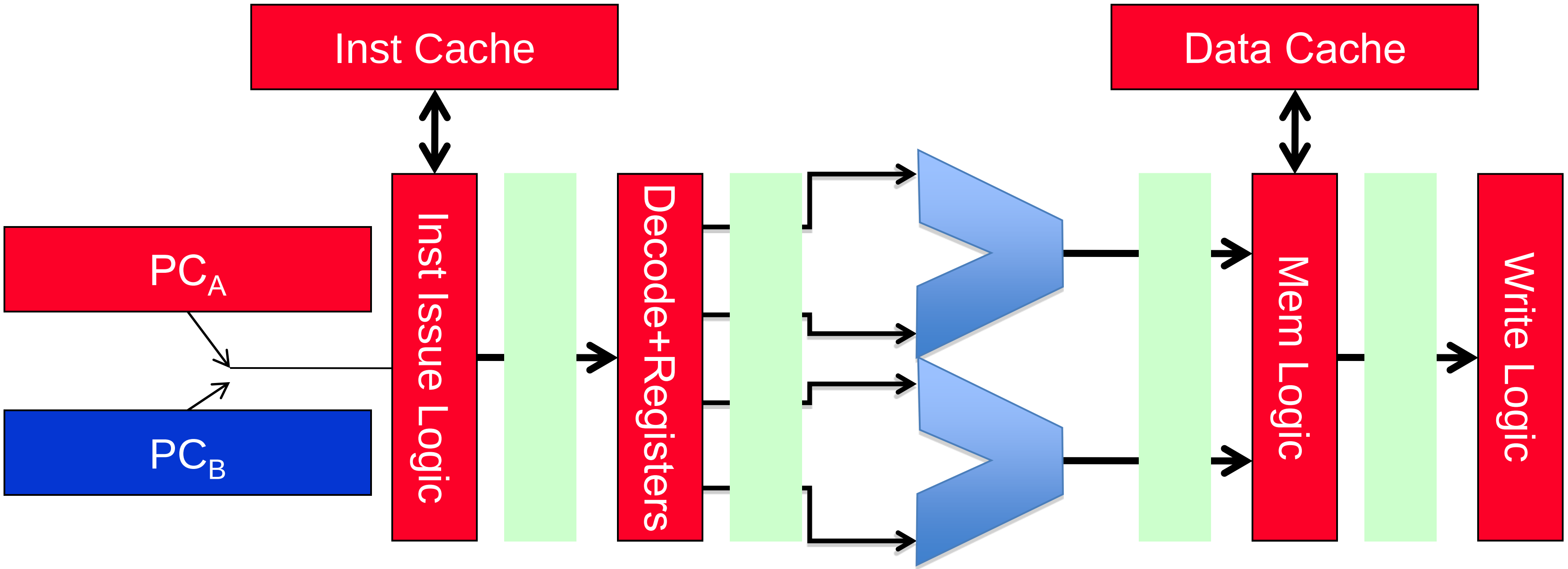
Simultaneous Multithreading

- 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

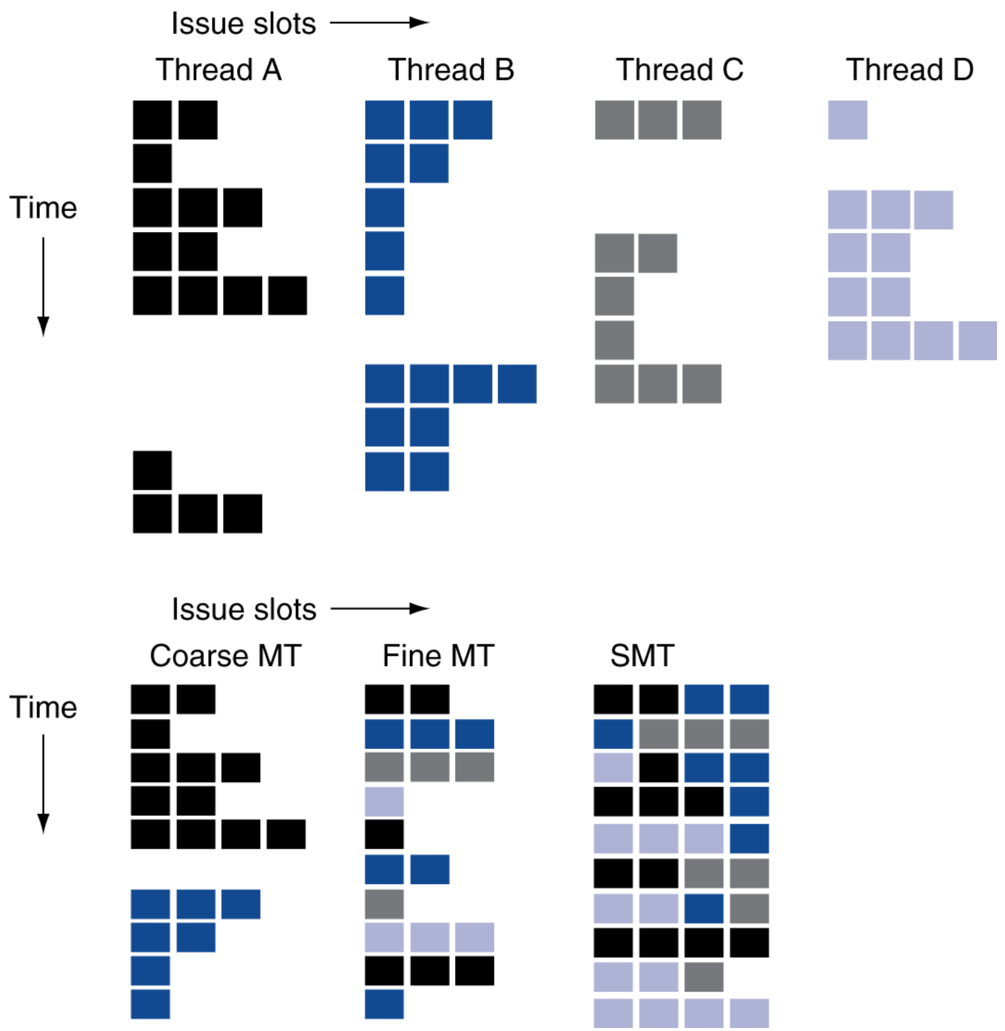
Simultaneous Multi-Threading

"permit different threads to occupy the same pipeline stage at the same time"

- This makes most sense with superscalar issue



Multithreading Example



Future of Multithreading

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

7.6. Instruction and Data Streams

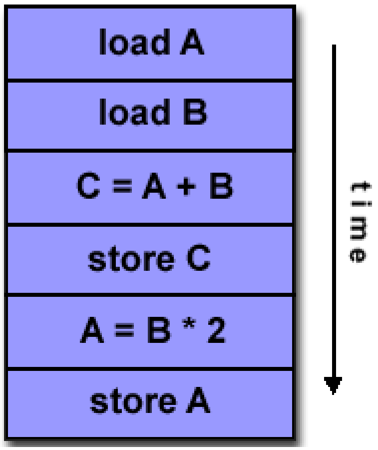
- An alternate classification

		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 parallel program on a MIMD computer
  - Conditional code for different processors

Single Instruction, Single Data

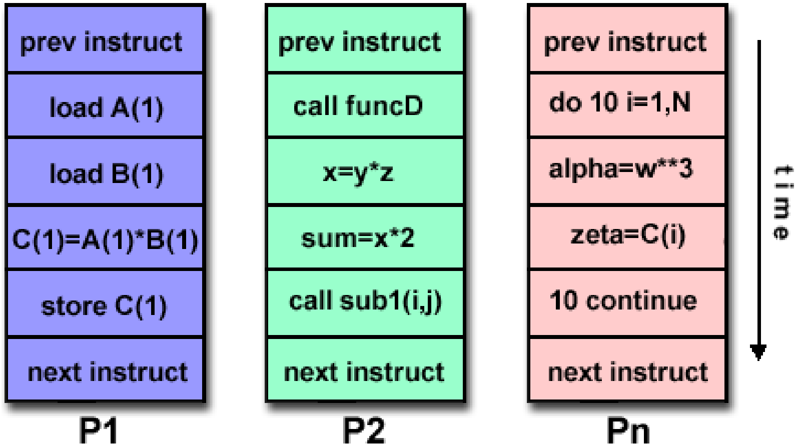
- **Single Instruction:** Only one instruction stream is being acted on by the CPU during any one clock cycle
- **Single Data:** Only one data stream is being used as input during any one clock cycle
- Deterministic execution
- Examples: older generation mainframes, minicomputers and workstations; most modern day PCs.



Single Instruction, Single Data

Multi Instruction, Multi Data

- **Multiple Instruction**: Every processor may be executing a different instruction stream
- **Multiple Data**: Every processor may be working with a different data stream



- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Currently, the most common type of parallel computer - most modern supercomputers fall into this category.
- Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.
- Note: many MIMD architectures also include SIMD execution sub-components

Multi Instruction, Multi Data


IBM POWER5


HP/Compaq Alphaserver


Intel IA32

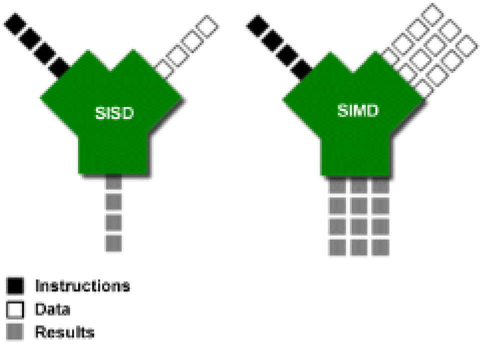

AMD Opteron


Cray XT3


IBM BG/L

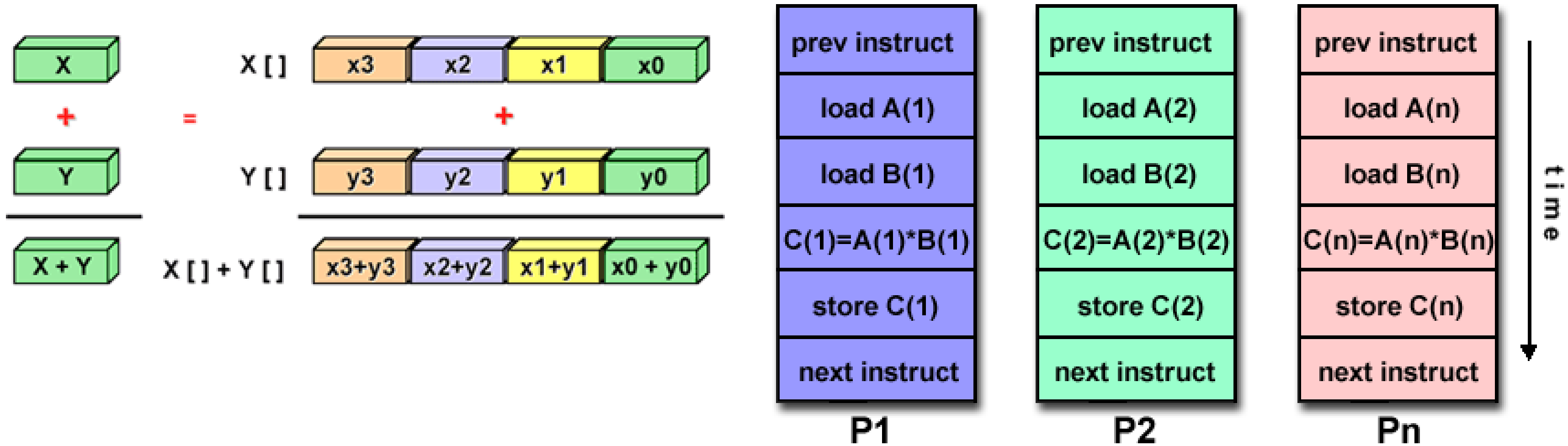
Single Instruction, Multiple Data

- Operate elementwise 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.



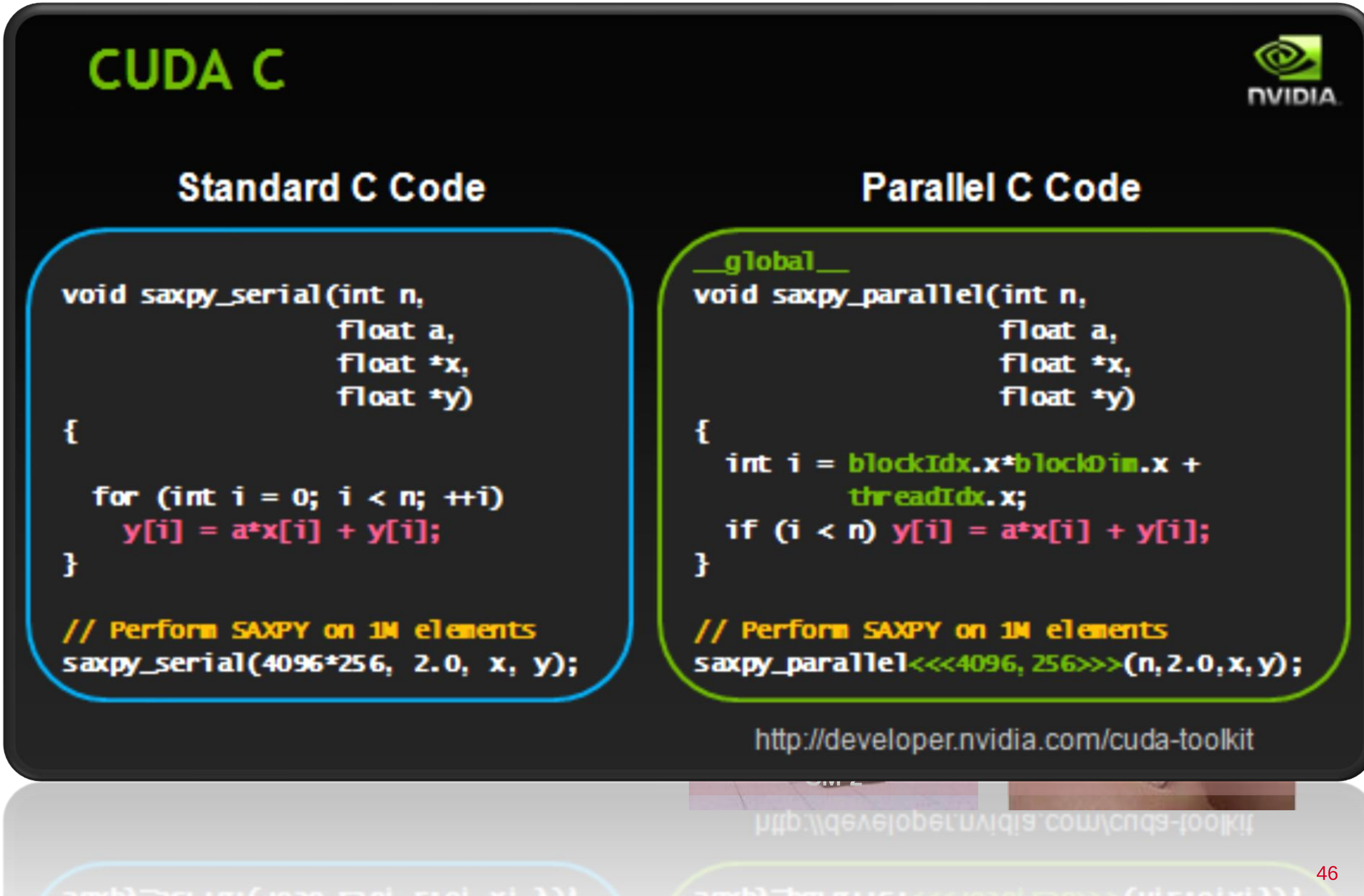
- Reduced instruction control hardware
- Works best for highly data-parallel applications, high degree of regularity, such as graphics/image processing

Single Instruction, Multiple Data



- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines
- Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.

Single Instruction, Multiple Data



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
```

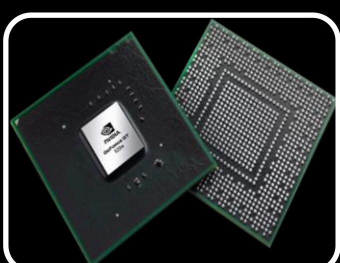
Vector vs. Scalar

- 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

7.7. History of GPUs

3D graphics processing

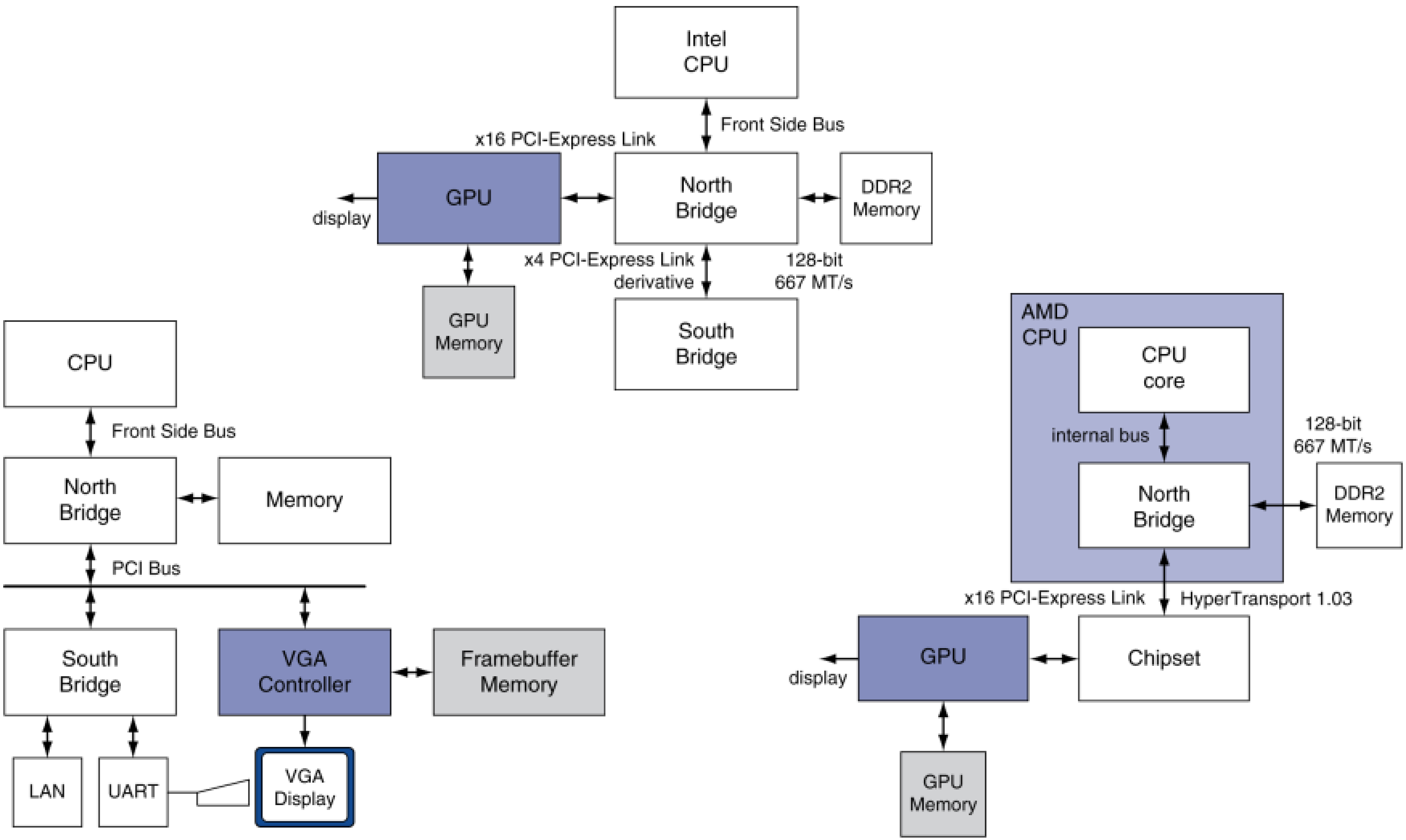
- Originally high-end computers (e.g., SGI)
- Moore's Law ⇒ lower cost, higher density
- 3D graphics cards for PCs and game consoles



Graphics Processing Units

- Processors oriented to 3D graphics tasks
- Vertex/pixel processing, shading, texture mapping, rasterization

Graphics in the System

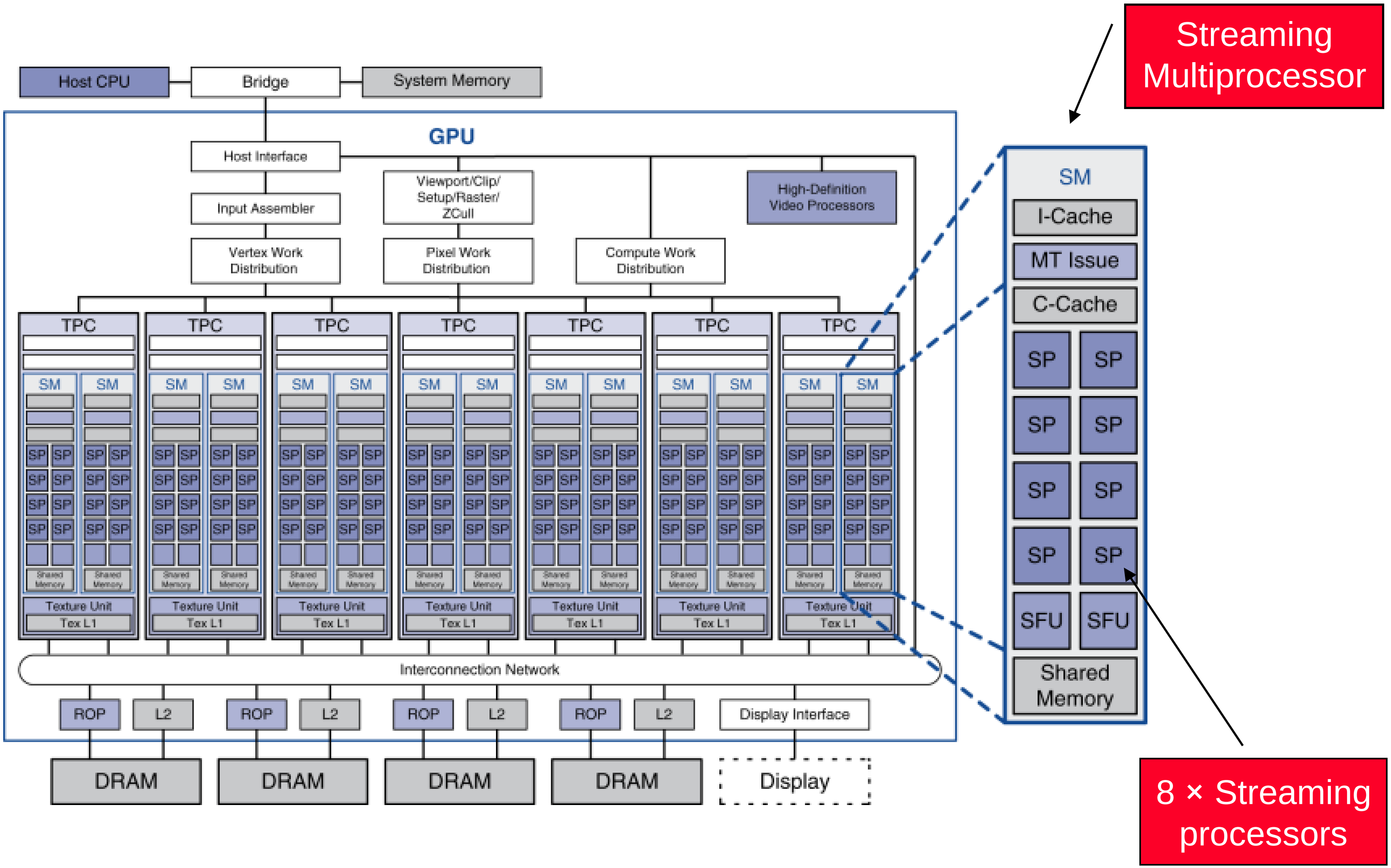


GPU Architectures

- 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)
  - C[illegible]ecture (CUDA)

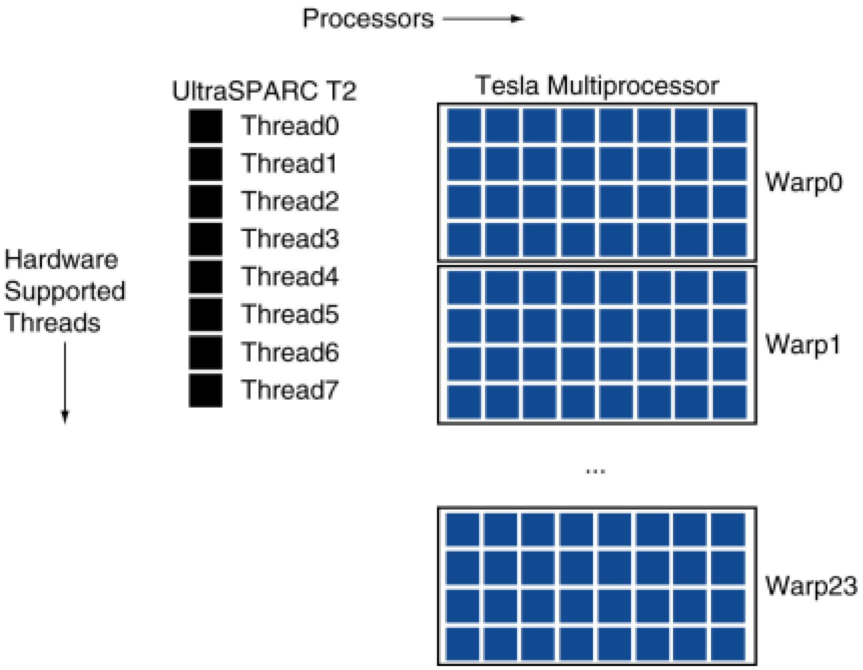
Heterogeneous: không đồng nhất

Example: NVIDIA Tesla



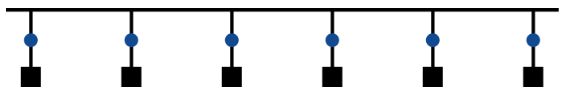
Example: NVIDIA Tesla

- Streaming Processors
  - Single-precision FP and integer units
  - Each SP is fine-grained multithreaded
- Warp: group of 32 threads
  - Executed in parallel, SIMD style
    - 8 SPs × 4 clock cycles
  - Hardware contexts for 24 warps
    - Registers, PCs, …

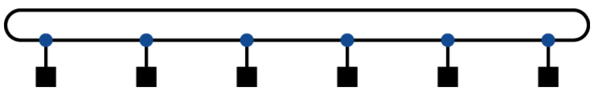


7.8. Interconnection Networks

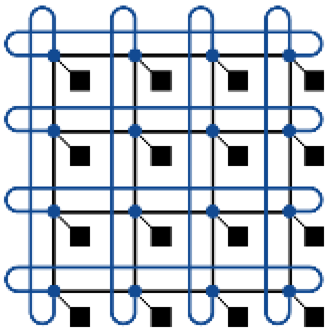
- Network topologies
  - Arrangements of processors, switches, and links



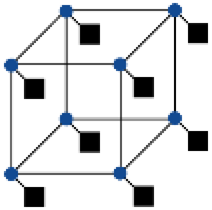
Bus



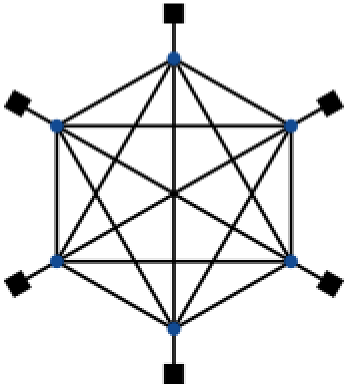
Ring



2D Mesh



N-cube (N = 3)



Fully connected

Network Characteristics

- Performance
  - Latency per message (unloaded network)
  - Throughput
    - Link bandwidth
    - Total network bandwidth
    - Bisection bandwidth
  - Congestion delays (depending on traffic)
- Cost
- Power
- Routability in silicon

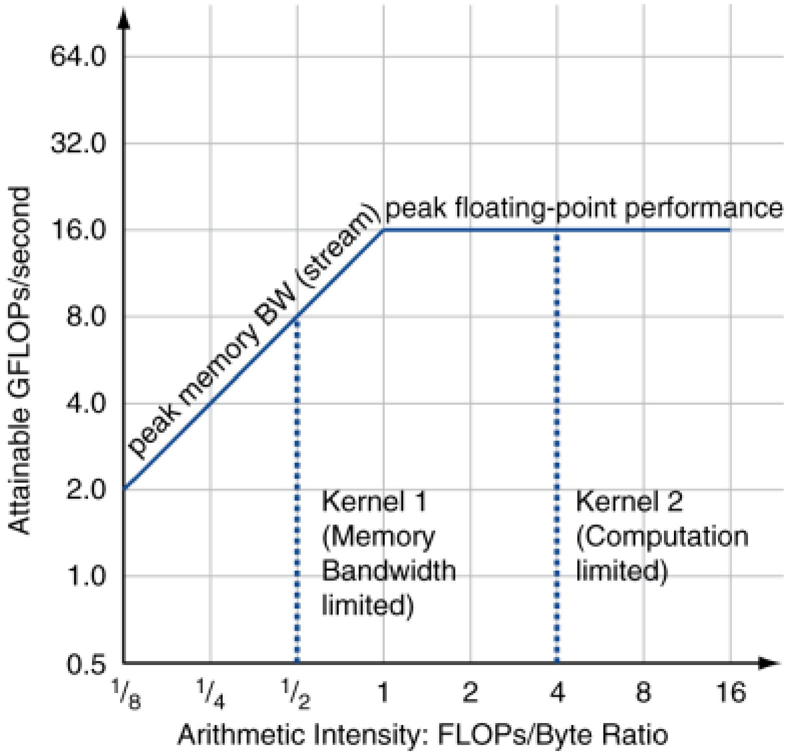
7.9. Parallel Benchmarks

- 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

7.10. Modeling Performance

- Assume performance metric of interest is achievable GFLOPs/sec
  - Measured using computational kernels from Berkeley Design Patterns
- Arithmetic intensity of a kernel
  - FLOPs per byte of memory accessed
- For a given computer, determine
  - Peak GFLOPS (from data sheet)
  - Peak memory bytes/sec (using Stream benchmark)

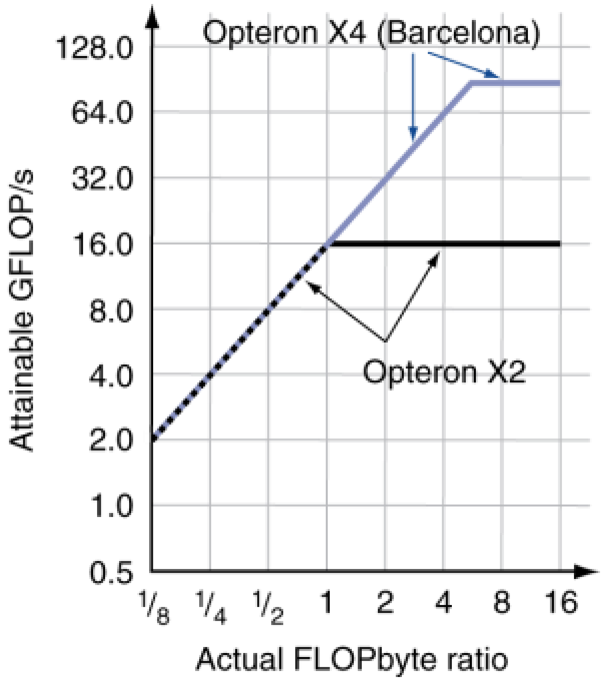
Roofline Diagram



Attainable GPLOPs/sec
= Max (Peak Memory BW × Arithmetic Intensity, Peak FP Performance)

Comparing Systems

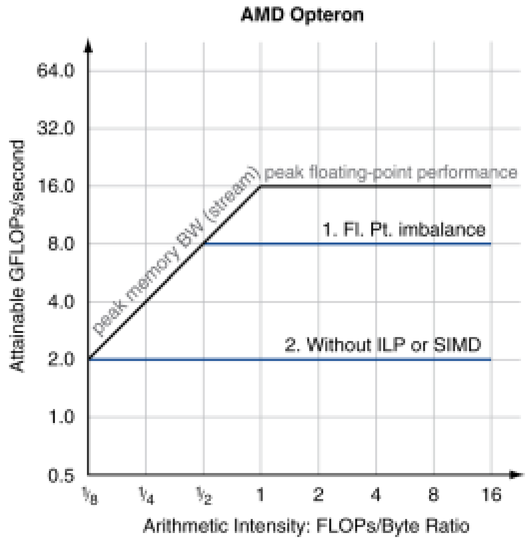
- Example: Opteron X2 vs. Opteron X4
  - 2-core vs. 4-core, 2× FP performance/core, 2.2GHz vs. 2.3GHz
  - Same memory system

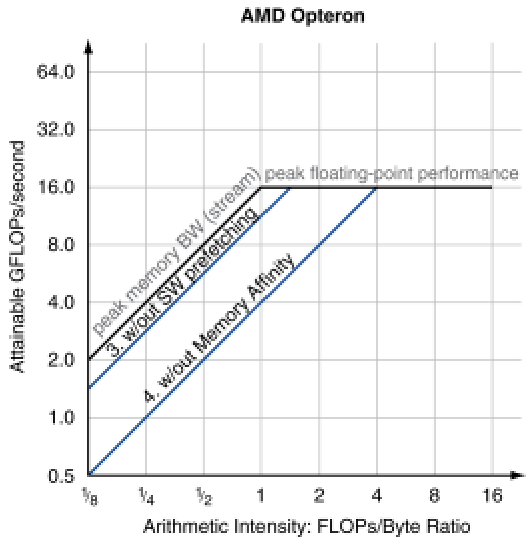


- To get higher performance on X4 than X2
  - Need high arithmetic intensity
  - Or working set must fit in X4's 2MB L-3 cache

Optimizing Performance

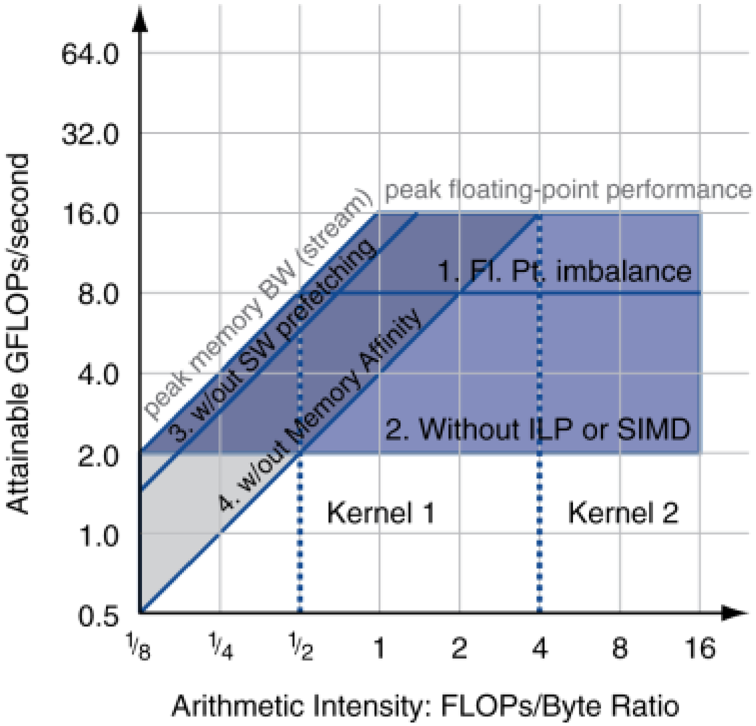
- Optimize FP performance
  - Balance adds & multiplies
  - Improve superscalar ILP and use of SIMD instructions
- Optimize memory usage
  - Software prefetch
    - Avoid load stalls
  - Memory affinity
    - Avoid non-local data accesses





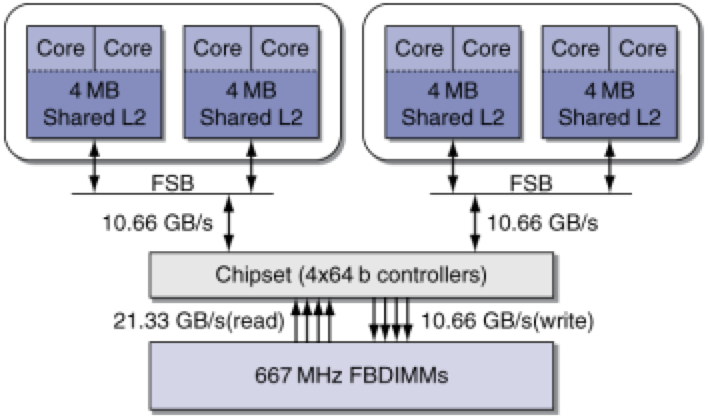
Optimizing Performance

- Choice of optimization depends on arithmetic intensity of code

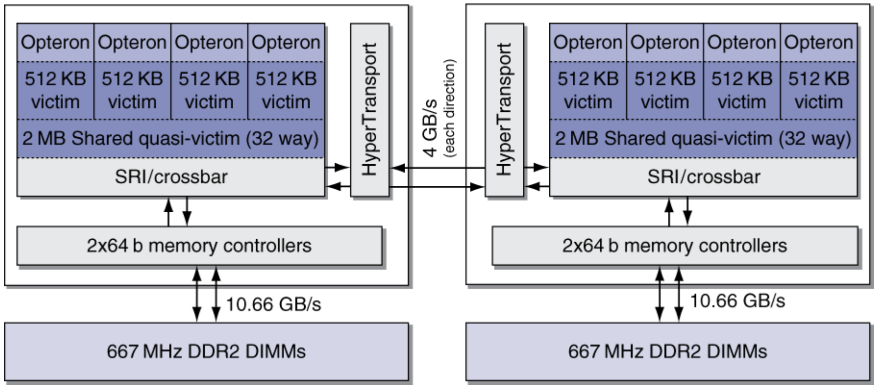


- Arithmetic intensity is not always fixed
  - May scale with problem size
  - Caching reduces memory accesses
    - Increases arithmetic intensity

7.11. Four Example Systems

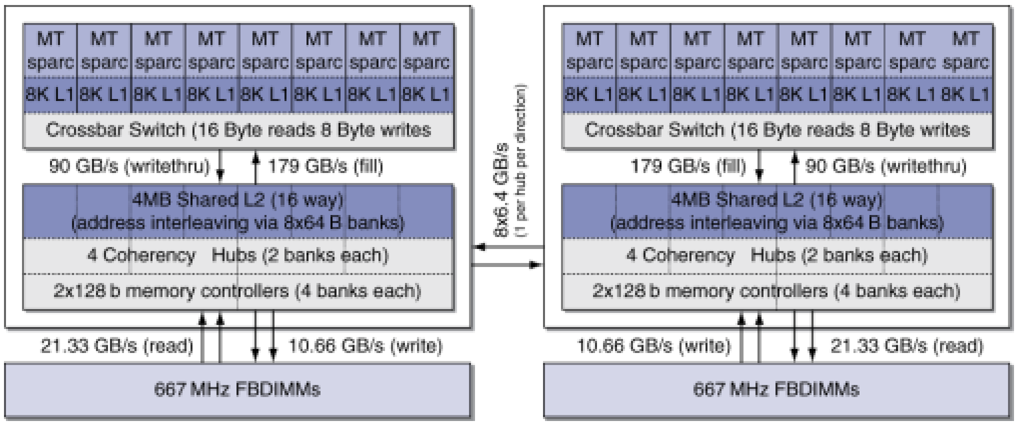


2 × quad-core
Intel Xeon e5345
(Clovertown)

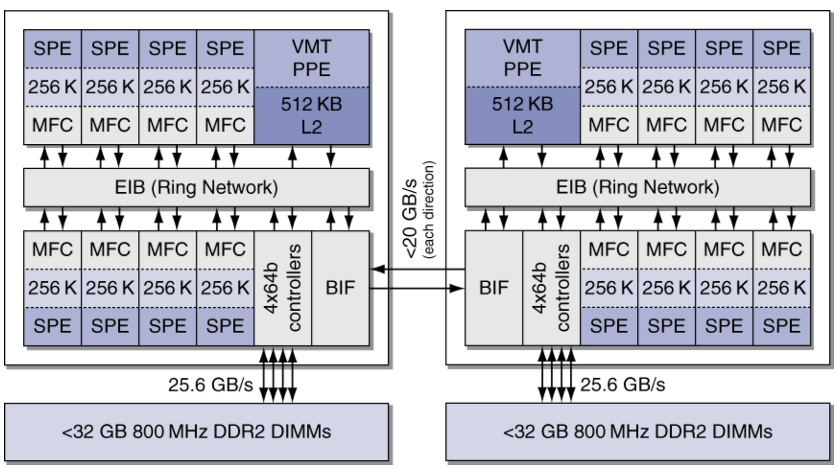


2 × quad-core
AMD Opteron X4 2356
(Barcelona)

Four Example Systems



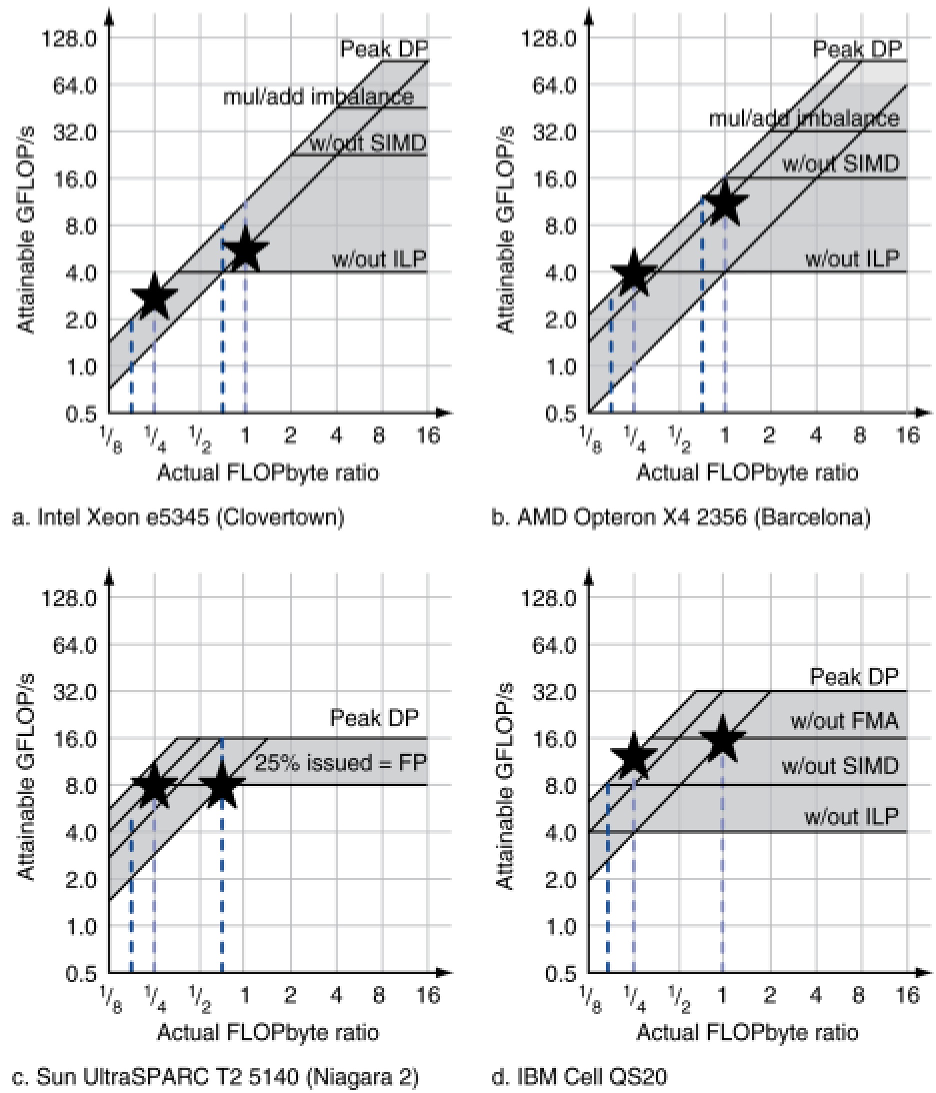
2 × oct-core
Sun UltraSPARC
T2 5140 (Niagara 2)



2 × oct-core
IBM Cell QS20

And Their Rooflines

- Kernels
  - SpMV (left)
  - LBHMD (right)
- Some optimizations change arithmetic intensity
- x86 systems have higher peak GFLOPs
  - But harder to achieve, given memory bandwidth



a. Intel Xeon e5345 (Clovertown)

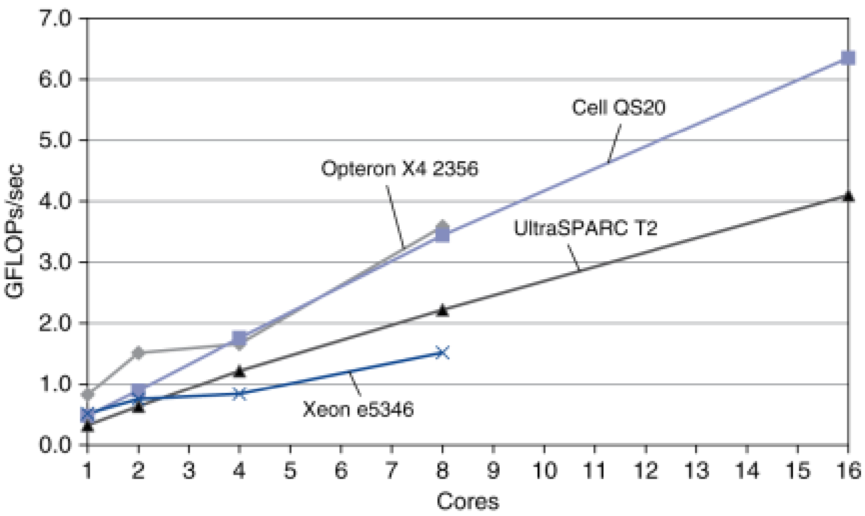
b. AMD Opteron X4 2356 (Barcelona)

c. Sun UltraSPARC T2 5140 (Niagara 2)

d. IBM Cell QS20

Performance on SpMV

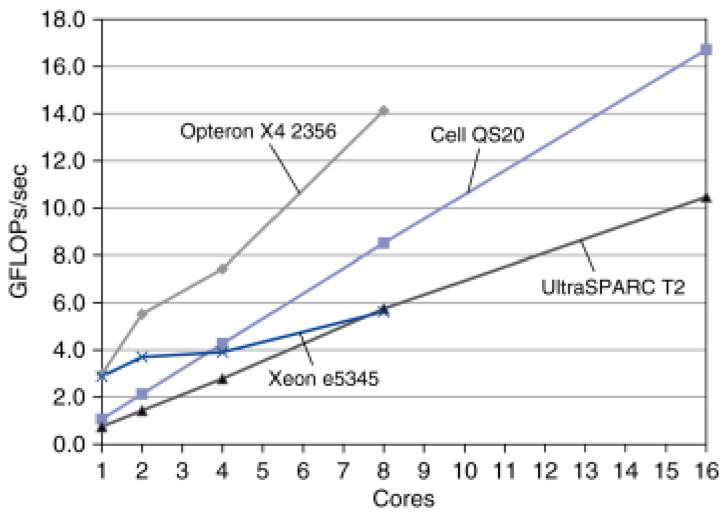
- Sparse matrix/vector multiply
  - Irregular memory accesses, memory bound
- Arithmetic intensity
  - 0.166 before memory optimization, 0.25 after



- Xeon vs. Opteron
  - Similar peak FLOPS
  - Xeon limited by shared FSBs and chipset
- UltraSPARC/Cell vs. x86
  - 20 – 30 vs. 75 peak GFLOPs
  - More cores and memory bandwidth

Performance on LBMHD

- Fluid dynamics: structured grid over time steps
  - Each point: 75 FP read/write, 1300 FP ops
- Arithmetic intensity
  - 0.70 before optimization, 1.07 after



- Opteron vs. UltraSPARC
  - More powerful cores, not limited by memory bandwidth
- Xeon vs. others
  - Still suffers from memory bottlenecks

Achieving Performance

- Compare naïve vs. optimized code
  - If naïve code performs well, it's easier to write high performance code for the system

System	Kernel	Naïve GFLOPs/sec	Optimized GFLOPs/sec	Naïve as % of optimized
Intel Xeon	SpMV	1.0	1.5	64%
	LBMHD	4.6	5.6	82%
AMD Opteron X4	SpMV	1.4	3.6	38%
	LBMHD	7.1	14.1	50%
Sun UltraSPARC T2	SpMV	3.5	4.1	86%
	LBMHD	9.7	10.5	93%
IBM Cell QS20	SpMV	Naïve code not feasible	6.4	0%
	LBMHD		16.7	0%

7.12. 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
  - Example: using a single lock for a shared composite resource
    - Serializes accesses, even if they could be done in parallel
    - Use finer-granularity locking

7.13. 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!