Chapter Six (1/2)

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)

Multiprocessor motivation, part 1

- Many scientific applications take too long to run on a single processor machine
  - Modeling of weather patterns, astrophysics, chemical reactions, ocean currents, etc.
- Many of these are *parallel applications* which largely consist of loops which operate on independent data
- Such applications can make efficient use of a multiprocessor machine with each loop iteration running on a different processor and operating on independent data

Multiprocessor motivation, part 2

- Many multi-user environments require more compute power than available from a single processor machine
  - Airline reservation system, department store chain inventory system, file server for a large department, web server for a major corporation, etc.
- These consist of largely *parallel transactions* which operate on independent data
- Such applications can make efficient usage of a multiprocessor machine with each transaction running on a different processor and operating on independent data

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

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

Amdahl's Law

- Sequential part can limit speedup
- Example: 100 processors, 90× speedup?
  - $T_{new} = T_{parallelizable}/100 + T_{sequential}$

$$Speedup = \frac{1}{(1 - F_{parallelizable}) + F_{parallelizable}/100} = 90$$

  - 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 = $(10 + 100) \times t_{add}$
- 10 processors
  - Time = $10 \times t_{add} + 100/10 \times t_{add} = 20 \times t_{add}$
  - Speedup = 110/20 = 5.5 (5.5/10 = 55% of potential)
- 100 processors
  - Time = $10 \times t_{add} + 100/100 \times t_{add} = 11 \times t_{add}$
  - 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 (9.9/10 = 99% of potential)
- 100 processors
  - Time = $10 \times t_{add} + 10000/100 \times t_{add} = 110 \times t_{add}$
  - Speedup = 10010/110 = 91 (91/100 = 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 = $10 \times t_{add} + 100/10 \times t_{add} = 20 \times t_{add}$
  - 100 processors, 32 × 32 matrix
    - Time = $10 \times t_{add} + 1024/100 \times t_{add} \approx 20 \times t_{add}$
  - Constant performance in this example

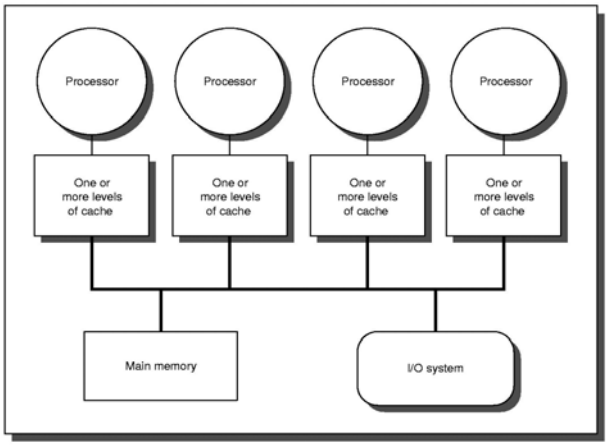
Multiprocessor

- **Shared Memory Multiprocessors**
  - **Communication through shared memory**
    - Load/Store
  - **Shared address space**
  - **Variants**
    - UMA (Uniform Memory Access)
    - NUMA (Non-Uniform Memory Access)
    - COMA (Cache-Only Memory Architecture)

- **Message Passing Multiprocessors**
  - **Communication through message passing**
    - Send/Receive
  - **Independent address space**

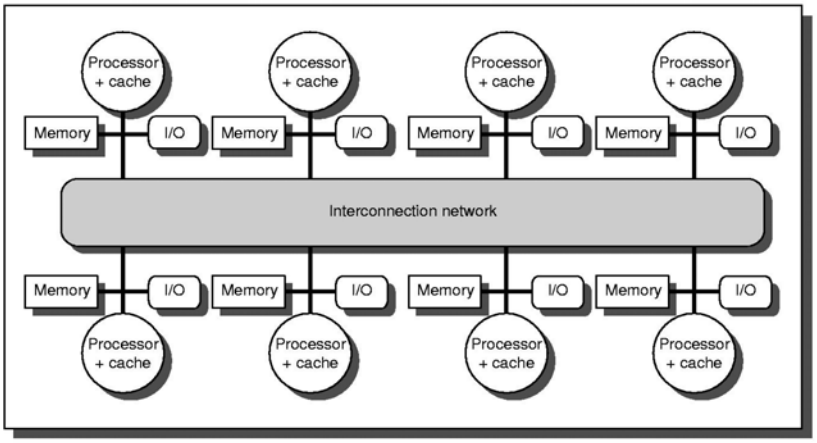
Multiprocessor organizations

- **Uniform memory access (UMA) shared memory multiprocessors**
  - **All processors share the same memory address space**
  - **Single copy of the OS (although some parts may be parallel)**
  - **Relatively easy to program and port sequential code to**
  - **Hardware-intensive to scale to large numbers of processors**



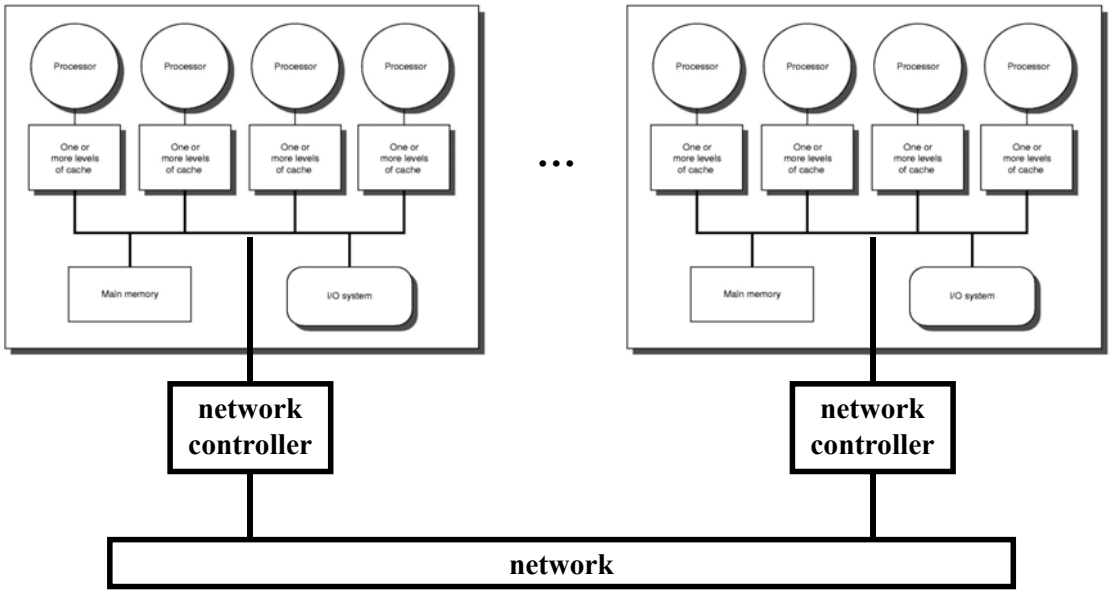
Multiprocessor organizations

- **Non-uniform memory access (NUMA) shared memory multiprocessors**
  - **All memory can be addressed by all processors, but access to a processor's own *local memory* is faster than access to another processor's *remote memory***
  - **Looks like a distributed machine, but interconnection network is usually custom-designed switches and/or buses**



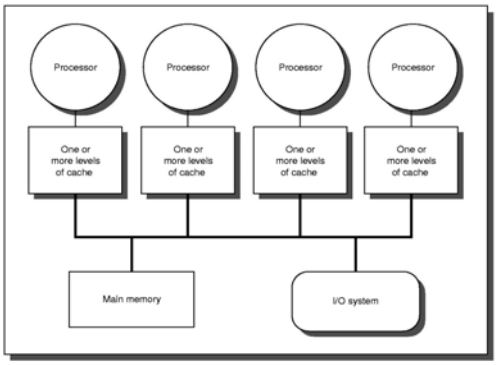
Multiprocessor variants

- **Shared memory machines connected together over a network (operating as a distributed memory or DSM machine)**



Shared Memory

- SMP: shared memory multiprocessor
  - Hardware provides single physical address space for all processors
  - Synchronize shared variables using locks

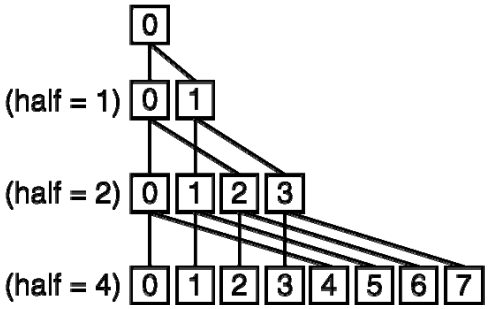


- Major design issues
  - Cache coherence: ensuring that stores to cached data are seen by other processors
  - Synchronization: the coordination among processors accessing shared data
  - Memory consistency: definition of when a processor must observe a write from another processor

Example: Sum Reduction

- Sum 100,000 numbers on 100 processor UMA
  - Each processor has ID: $0 \le P_n \le 99$
  - 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
    ```

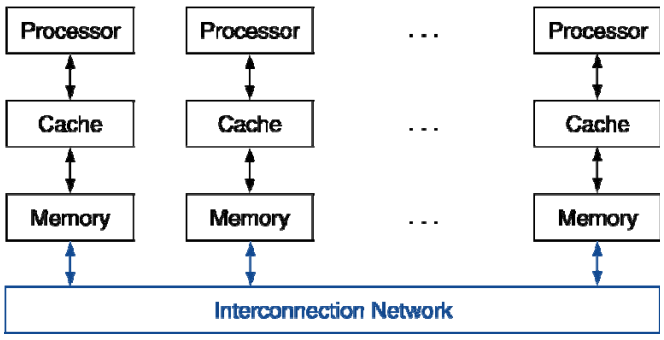
Example: Sum Reduction




```
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);
```


Message Passing

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



Network Characteristics

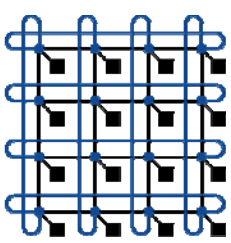
- **Performance**
  - **Latency per message (unloaded network)**
  - **Throughput**
    - Link bandwidth
    - Total network bandwidth
    - Bisection bandwidth
  - **Congestion delays (depending on traffic)**
- **Cost**
- **Power**
- **Routability in silicon**

Interconnection Networks

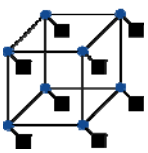
- **Network topologies**
  - **Arrangements of processors, switches, and links**

Bus

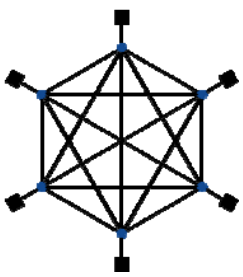
Ring



2D Mesh

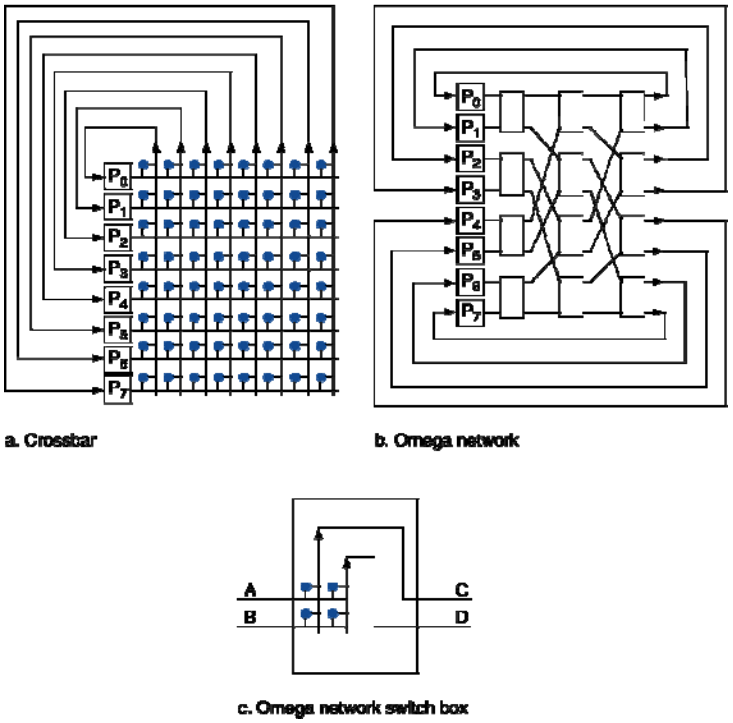


N-cube (N = 3)



Fully connected

Multistage Networks



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 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

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

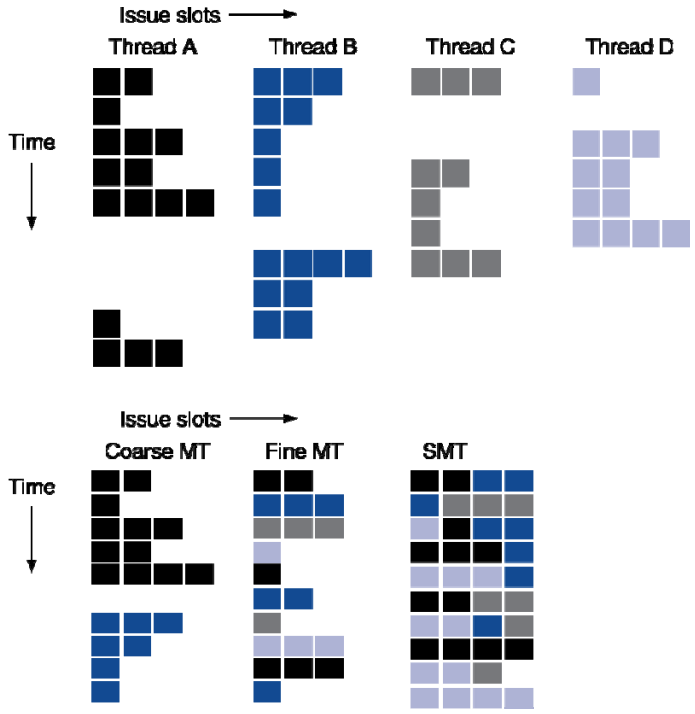
Multithreading

- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads
- 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

- 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

Multithreading Example



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 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**

SIMD

- **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.**
- **Simplifies synchronization**
- **Reduced instruction control hardware**
- **Works best for highly data-parallel applications**

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