上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

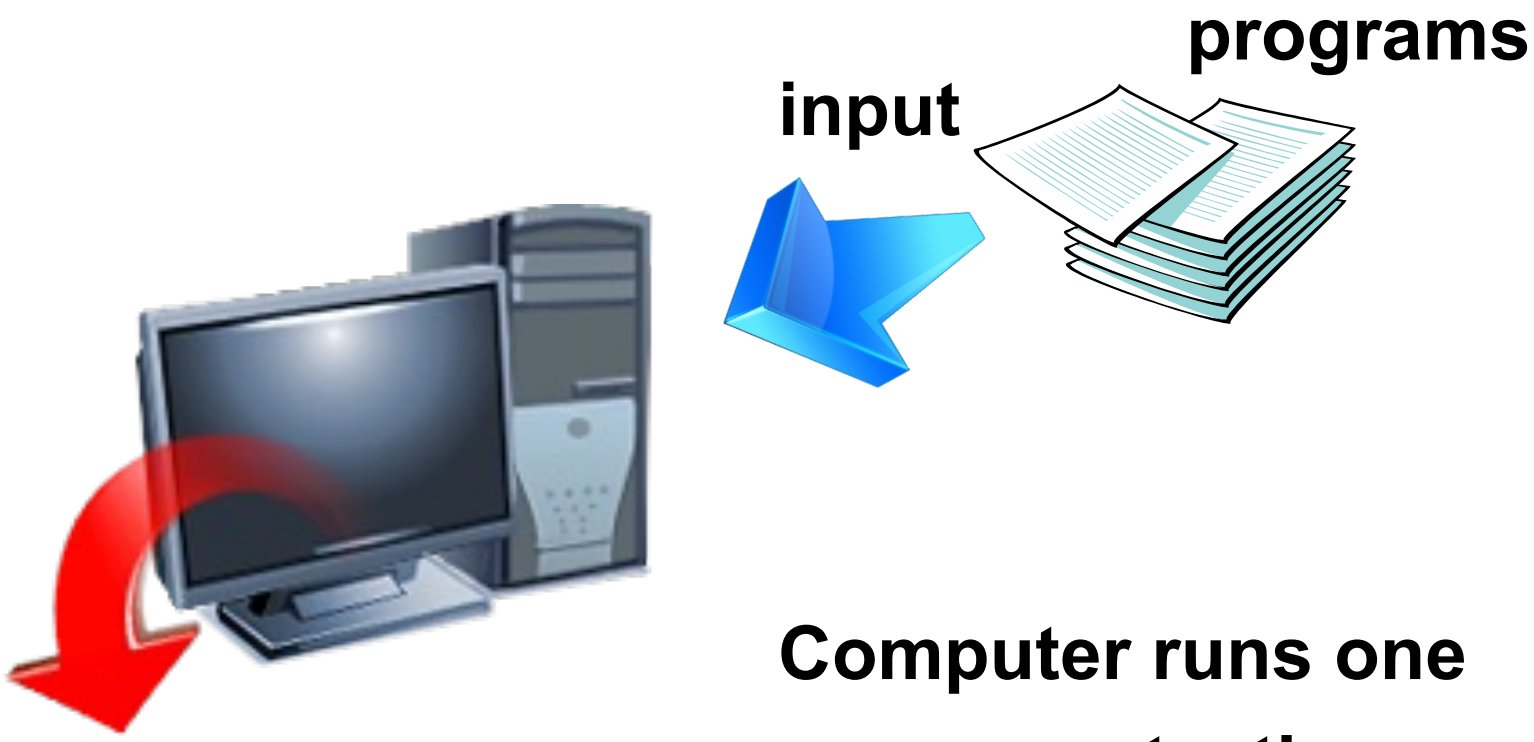
CS4302-01
Parallel and Distributed Computing

Lecture 2 Basic Architecture

Zhuoran Song

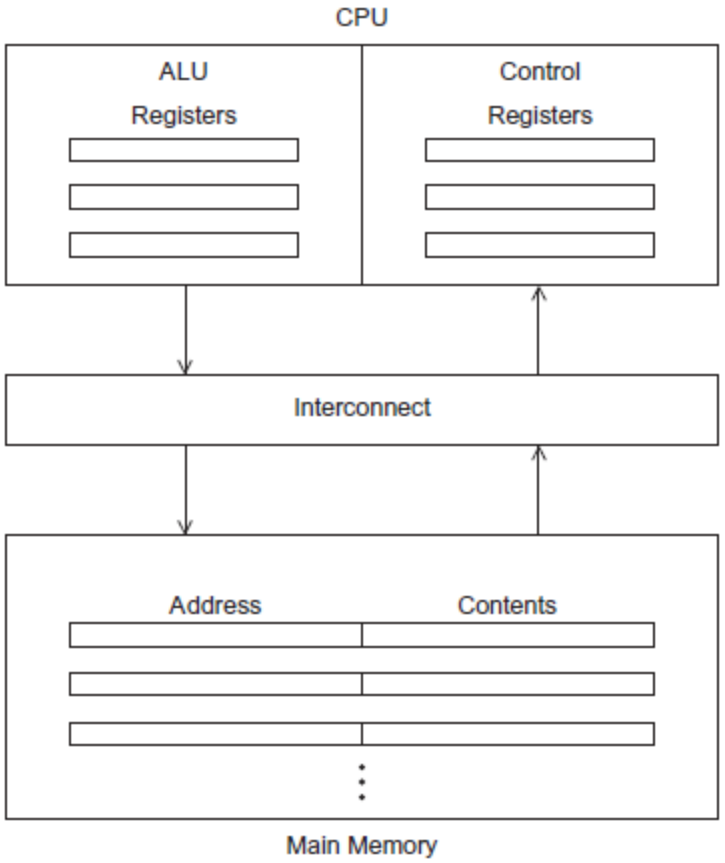
2023/9/15

Basic Computer Model



Computer runs one program at a time.

The Von Neumann Machine



Main Components

- **Main memory**
  - *This is a collection of locations, each of which is capable of storing both instructions and data.*
  - *Every location consists of an address, which is used to access the location, and the contents of the location.*

- **CPU**
  - ***Control unit*** *- responsible for deciding which instruction in a program should be executed.*
  - ***ALU*** *- responsible for executing the actual instructions.*
  - ***Register*** *– very fast storage, part of the CPU.*
  - ***Program counter*** *– stores address of the next instruction to be executed.*

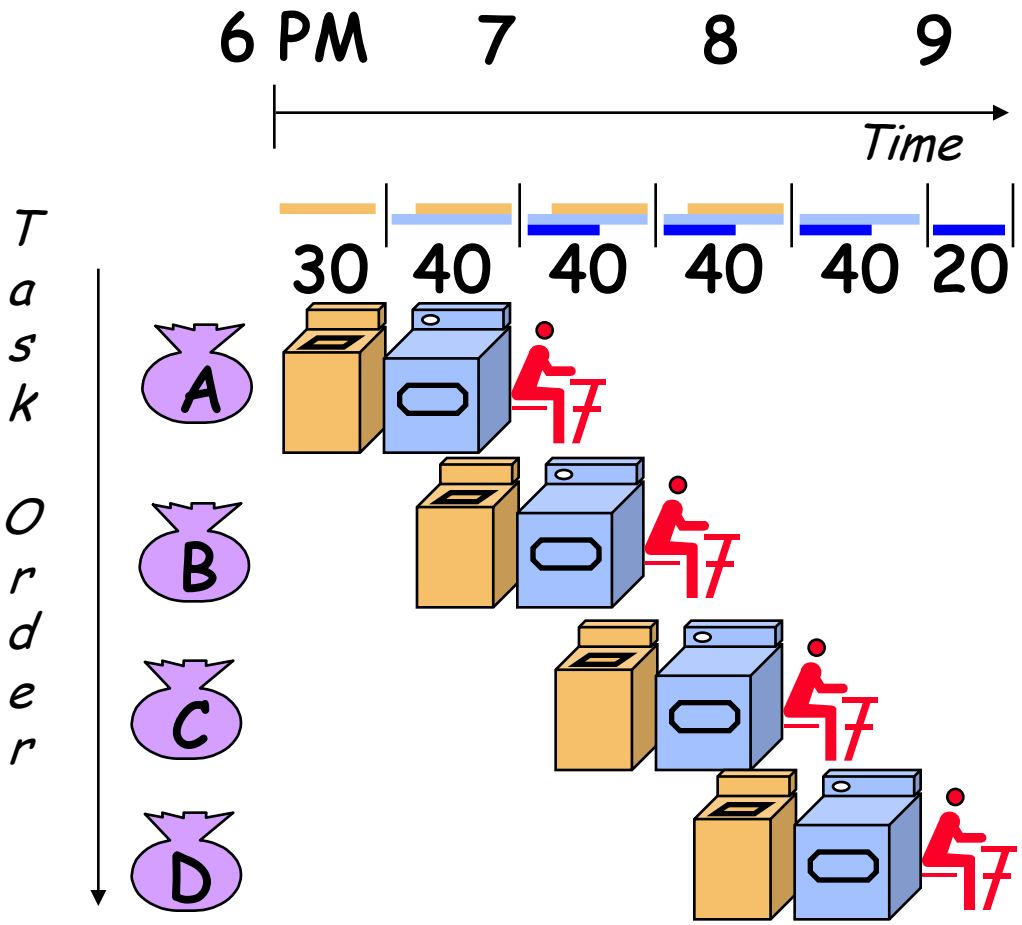
- **Bus**
  - *Wires and hardware that connects the CPU and memory.*

- **Cache**
  - *Fill the gap between CPU and main memory*

Pipeline

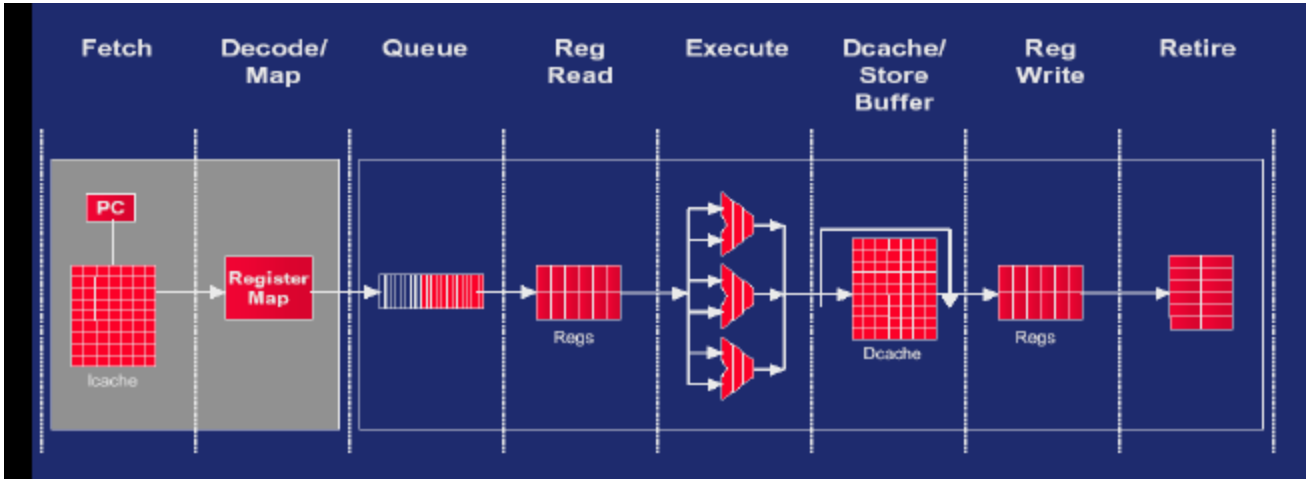
Dave Patterson's Laundry example: 4 people doing laundry

wash (30 min) + dry (40 min) + fold (20 min) = 90 min Latency



- In this example:
  - *Sequential execution takes 4 * 90min = 6 hours*
  - *Pipelined execution takes 30+4*40+20 = 3.5 hours*
- Bandwidth = loads/hour
- BW = 4/6 l/h w/o pipelining
- BW = 4/3.5 l/h w pipelining
- BW <= 1.5 l/h w pipelining, more total loads
- Pipelining helps bandwidth but not latency (90 min)
- Potential speedup = Number pipe stages

Pipeline



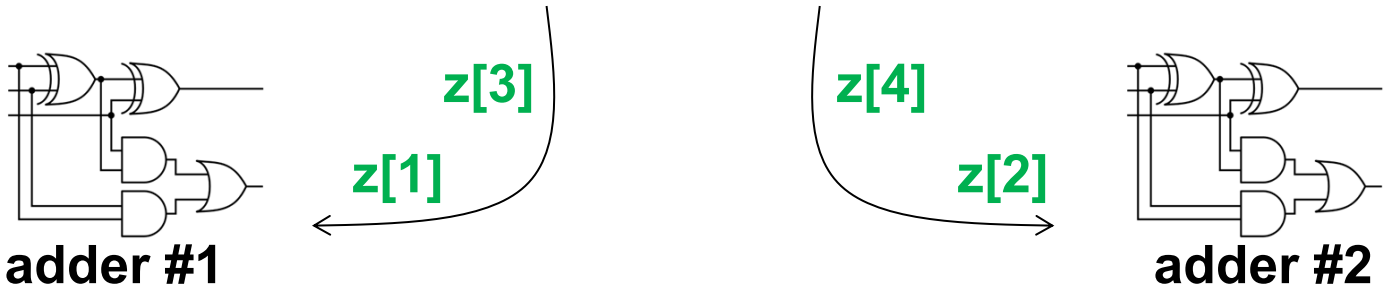

```
float x[1000], y[1000], z[1000];
. . .
for (i = 0; i < 1000; i++)
    z[i] = x[i] + y[i];
```

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Multiple Issue多发射

- Multiple issue processors replicate functional units and try to simultaneously execute different instructions in a program.

```
for (i = 0; i < 1000; i++)
    z[i] = x[i] + y[i];
```



Multiple Issue

- **Static multiple issue**
  - *functional units are scheduled at compile time.*

Type	*PipeStages*				
Int. instructionIF	ID	EX	MEM	WB	
FP instruction	IF	ID	EX	MEM	WB
Int. instruction	IF	ID	EX	MEM	WB
FP instruction		IF	ID	EX	MEM	WB
Int. instruction		IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB

- **Dynamic multiple issue**
  - *functional units are scheduled at run-time.*
  - *Allow instruction behind stalls to proceed*

```
DIVD   F0,F2,F4
ADDD   F10,F0,F8
SUBD   F12,F8,F14
```

  - *Enable out-of-order execution and out-of-order completion*
  - *Hardware limitation, dependency (hazard), if-else branch*

Register Renaming

- **Multiple issue challenge**
  - *WAR, WAW harzard*

I: sub r4,r1,r3
J: add r1,r2,r3

I: sub r1,r4,r3
J: add r1,r2,r3

  - *Register renaming*

1. R1=M[1024]
2. R1=R1+2
3. M[1032]=R1
4. R1=M[2048]
5. R1=R1+4
6. M[2056]=R1

1. R1=M[1024]	4. R2=M[2048]
2. R1=R1+2	5. R2=R2+4
3. M[1032]=R1	6. M[2056]=R2

Loop Unrolling

```
for (i = 1 to bignumber) 【每一条的输入依赖于上一条的输出】
   A(i)
   B(i)
   C(i)
 end
```

After Unrolling:

```
for (i = 1 to bignumber/3)
   A(i)
   A(i+1)
   A(i+2)
   B(i)
   B(i+1)
   B(i+2)
   C(i)
   C(i+1)
   C(i+2)
end
```

Without Unrolling: $A(1)\ \mathcal{R}\ \mathcal{W}\ B(1)\ \mathcal{R}\ \mathcal{W}\ C(1)\ \mathcal{R}\ \mathcal{W}\ldots$

With Unrolling: $A(1)\ A(2)\ A(3)\ B(1)\ B(2)\ B(3)\ C(1)\ C(2)\ C(3)\ \ldots$

Software Pipelining

```
for (i = 1 to bignumber) 【每一条的输入依赖于上一条的输出】
    A(i) ; 3
    B(i) ; 3
    C(i) ; 12(perhaps a floating point operation)
    D(i) ; 3
    E(i) ; 3
    F(i) ; 3
End
```

Software Pipelining:

```
for i = (1 to bignumber - 6)
    A(i+6)
    B(i+5)
    C(i+4)
    D(i+2)
    E(i+1)
    F(i)
end
```

Iteration 1: A(7) B(6) C(5) D(3) E(2) F(1)
Iteration 2: A(8) B(7) C(6) D(4) E(3) F(2)
Iteration 3: A(9) B(8) C(7) D(5) E(4) F(3)
Iteration 4: A(10) B(9) C(8) D(6) E(5) F(4)
Iteration 5: A(11) B(10) C(9) D(7) E(6) F(5)
Iteration 6: A(12) B(11) C(10) D(8) E(7) F(6)
Iteration 7: A(13) B(12) C(11) D(9) E(8) F(7)

Speculation

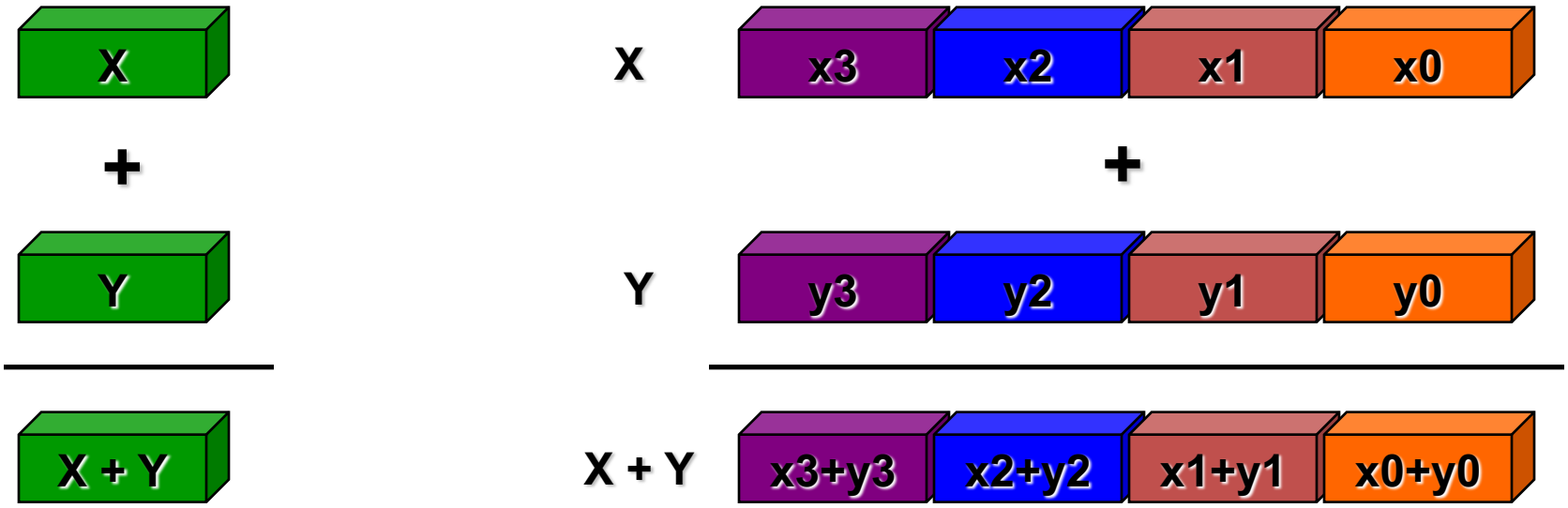
```
for (i = 0; i < 1000; i++)
    z[i] = x[i] + y[i];
```

In speculation, the compiler or the processor makes a guess about an instruction, and then executes the instruction on the basis of the guess

Always speculate the branch is taken, 1/1000 error rate!

Vector Processing

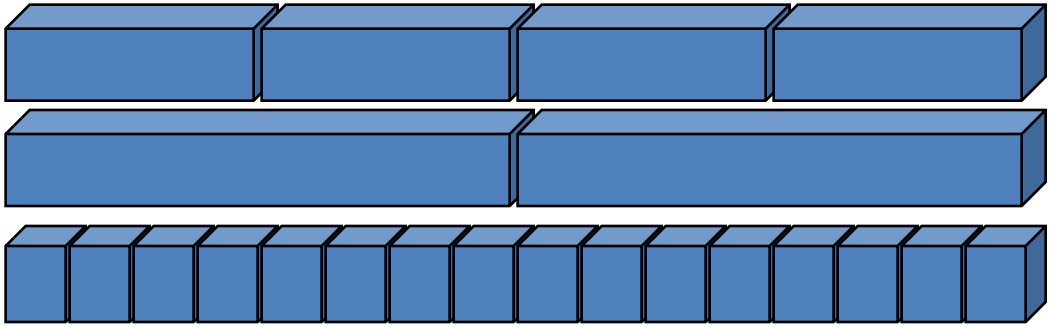
- Scalar processing
  - traditional mode
  - one operation produces one result
- SIMD processing
  - with SSE / SSE2
  - one operation produces multiple results



Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

Vector Processing

- Intel SSE2 data types: anything that fits into 16 bytes



- Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel
- Challenges:
  - *Need to be contiguous in memory and aligned*
  - *Some instructions to move data around from one part of register to another*

Vector Processing

- **Vector Register**
  - Fixed length bank holding a single vector has at least 2 read and 1 write ports typically 8-32 vector registers, each holding 64-128 64-bit elements
- **Vector Functional Units (FUs)**
  - Fully pipelined, start new operation every clock typically 4 to 8 FUs: FP add, FP mult, FP reciprocal (1/X), integer add, logical, shift; may have multiple of same unit
- **Vector Load-Store Units (LSUs)**
  - fully pipelined unit to load or store a vector; may have multiple LSUs
- **Scalar registers**
  - Single element for FP scalar or address
- **Cross-bar to connect FUs , LSUs, registers**

Vector Processing

- **Easy to get high performance**
  - *N operations:*
  - *are independent*
  - *use same functional unit*
  - *access disjoint registers*
  - *access registers in same order as previous instructions*
  - *access contiguous memory words or known pattern*
  - *can exploit large memory bandwidth*
  - *hide memory latency (and any other latency)*
- **Scalable**
  - *get higher performance as more HW resources available*
- **Compact**
  - *Describe N operations with 1 short instruction (v. VLIW)*
- **Predictable**
  - *(real-time) performance vs. statistical performance (cache)*
- **Multimedia ready**
  - *choose N * 64b, 2N * 32b, 4N * 16b, 8N * 8b*

Multithreading

- **Hardware multithreading provides a means for systems to continue doing useful work when the task being currently executed has stalled.**
  - *Ex., the current task has to wait for data to be loaded from memory.*

- **Coarse-grained - only switches threads that are stalled waiting for a time-consuming operation to complete.**
  - *Pros: switching threads doesn't need to be nearly instantaneous.*
  - *Cons: the processor can be idled on shorter stalls, and thread switching will also cause delays.*

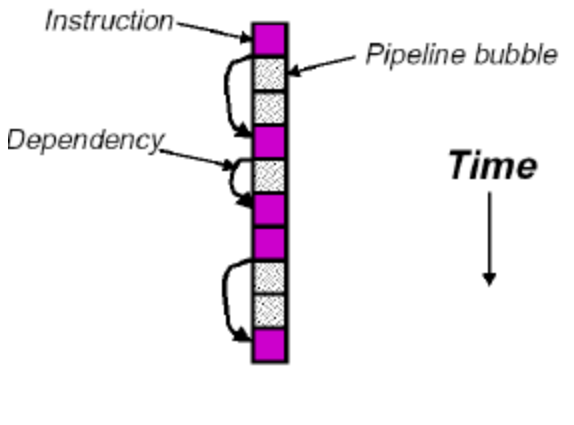
- **Fine-grained - the processor switches between threads after each instruction, skipping threads that are stalled.**
  - *Pros: potential to avoid wasted machine time due to stalls.*
  - *Cons: a thread that's ready to execute a long sequence of instructions may have to wait to execute every instruction.*

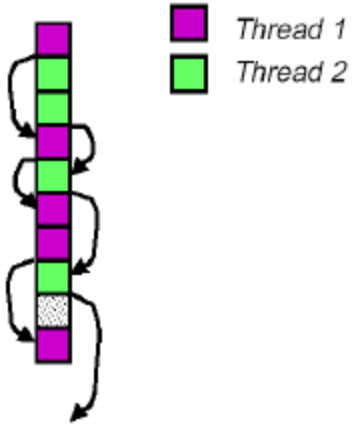
Multithreading

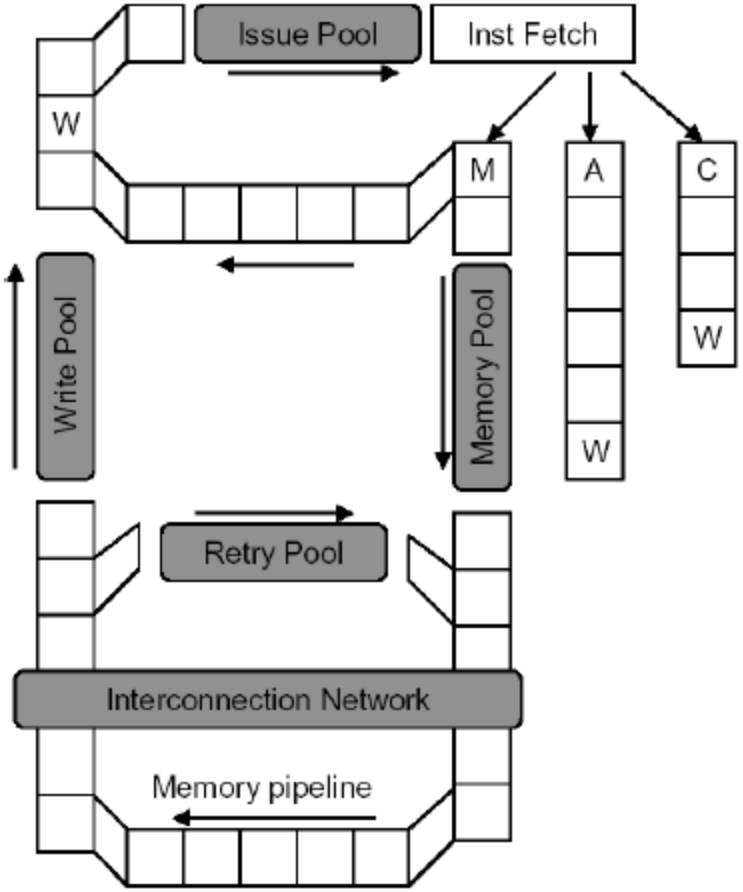
- Simultaneous Multithreading (SMT)

Cray/Tera MTA

- http://www.cray.com/products/systems/mta/, http://www.utc.edu/~jdumas/cs460/paperfa01/craymta/

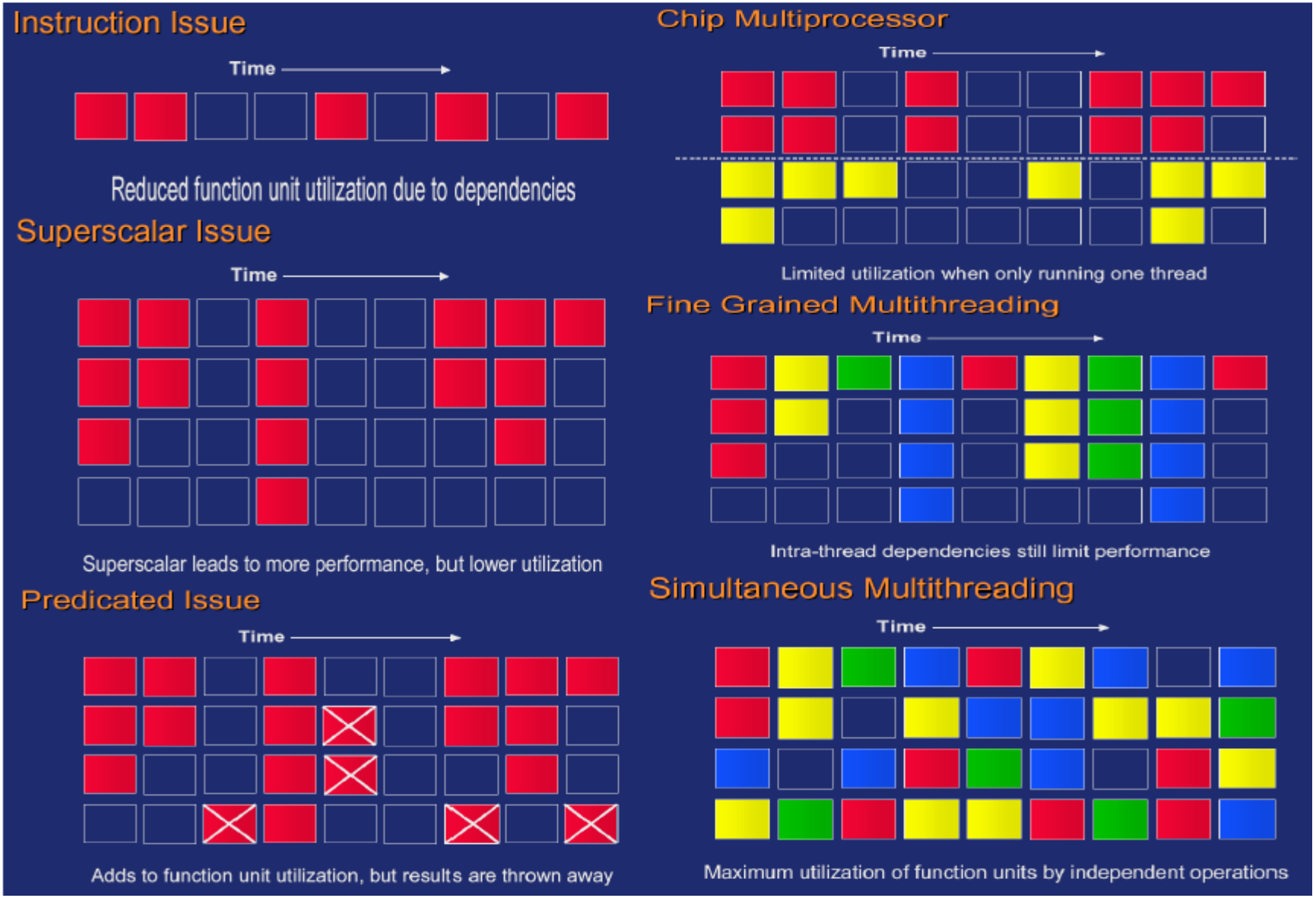






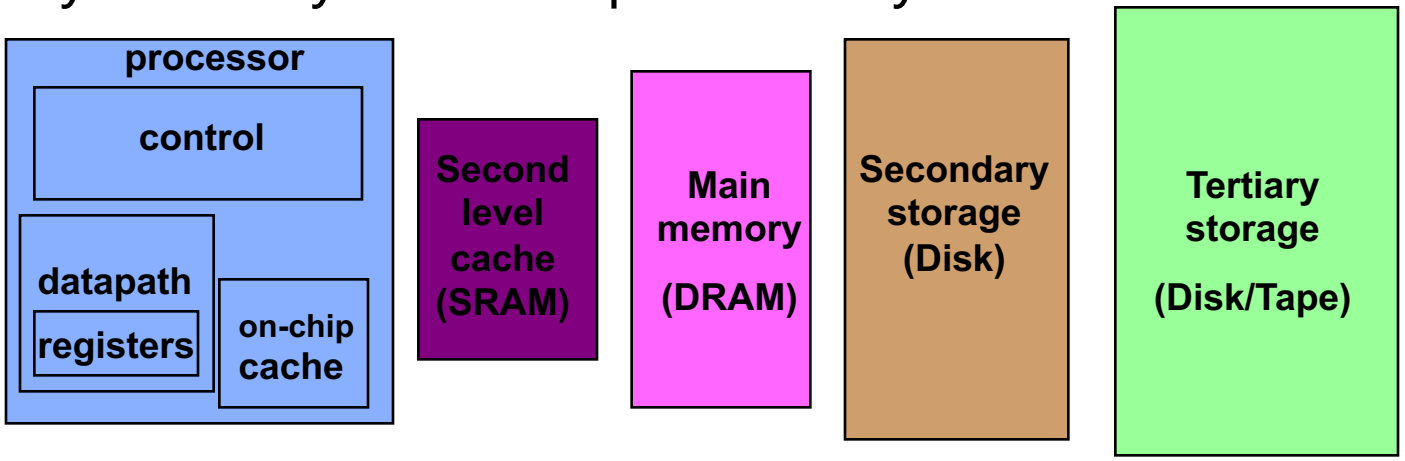
- **Problem: Dependencies limit sustainable throughput of single instruction stream**
- **Solution: Interleave execution of two or more instruction streams on same hardware to increase utilization**

Parallel Processors



Memory Hierarchy

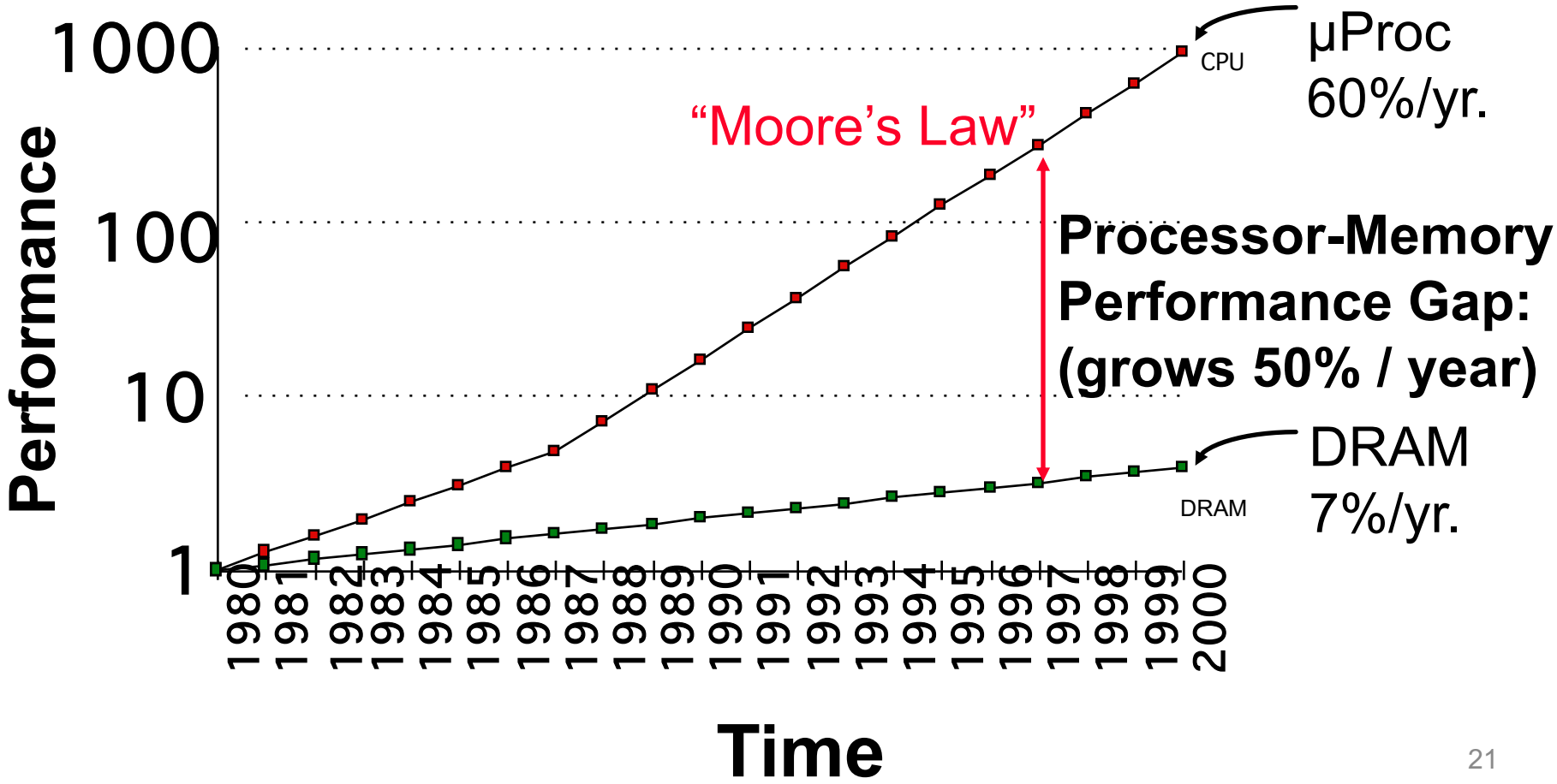
- Most programs have a high degree of locality in their accesses
  - *spatial locality: accessing things nearby previous accesses*
  - *temporal locality: reusing an item that was previously accessed*
- Memory hierarchy tries to exploit locality



Speed	1ns	10ns	100ns	10ms	10sec
Size	B-KB	KB-MB	GB	TB	PB

Processor–DRAM Gap

- Memory hierarchies are getting deeper
  - *Processors get faster more quickly than memory*



Cache Mappings

- **Full associative**
  - *a new line can be placed at any location in the cache.*
- **Direct mapped**
  - *each cache line has a unique location in the cache to which it will be assigned.*
- **n-way set associative**
  - *each cache line can be place in one of n different locations in the cache.*

Cache Write

- **Cache write**
  - *the value in cache may be inconsistent with the value in main memory.*
- **Write-through caches**
  - *handle this by updating the data in main memory at the time it is written to cache.*
- **Write-back caches**
  - *mark data in the cache as dirty. When the cache line is replaced by a new cache line from memory, the dirty line is written to memory.*

Reduce Misses

Classifying Misses: 3 Cs

- **Compulsory**
  - *The first access to a block is not in the cache, so the block must be brought into the cache. Also called cold start missesor first reference misses.(Misses in even an Infinite Cache)*
- **Capacity**
  - *If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved.(Misses in Fully Associative Size X Cache)*
- **Conflict**
  - *If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called collision missesor interference misses.(Misses in N-way Associative, Size X Cache)*
- **Maybe four: 4th "C"**
  - *Coherence-Misses caused by cache coherence: data may have been invalidated by another processor or I/O device*

Cache and Program

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
    for (j = 0; j < MAX; j++)
        y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
    for (i = 0; i < MAX; i++)
        y[i] += A[i][j]*x[j];
```

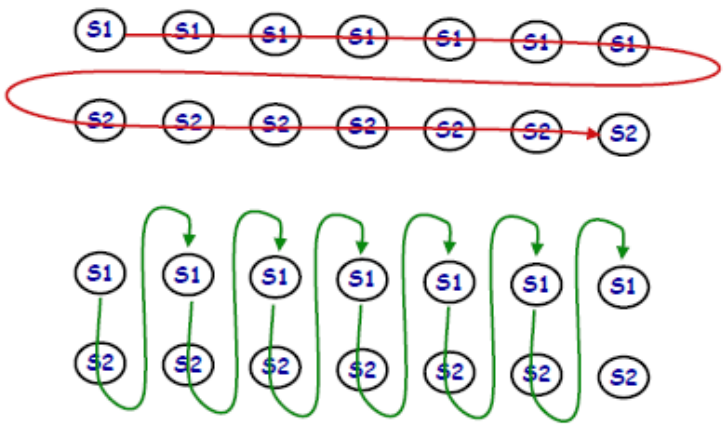
Assuming only two cache lines

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

Cache and Program

Loop Fusion: example

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  S1: a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  S2: d[i][j] = a[i][j] + c[i][j];
/* After fusion */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  {S1: a[i][j] = 1/b[i][j] * c[i][j];
   S2: d[i][j] = a[i][j] + c[i][j];}
```




```
/* After array contraction */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  {    c = c[i][j];
   S1: a = 1/b[i][j] * c;
   S2: d[i][j] = a + c;}
```

2 misses per access to a & c vs. one miss per access; improve spatial locality

The real payoff comes if fusion enables Array Contraction: values transferred in scalar instead of via array

Common Techniques

- **Merging Arrays**
  - *improve spatial locality by single array of compound elements vs. 2 arrays*
- **Permuting a multidimensional array**
  - *improve spatial locality by matching array layout to traversal order*
- **Loop Interchange**
  - *change nesting of loops to access data in order stored in memory*
- **Loop Fusion**
  - *Combine 2 independent loops that have same looping and some variables overlap*
- **Blocking**
  - *Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows*

Main Memory

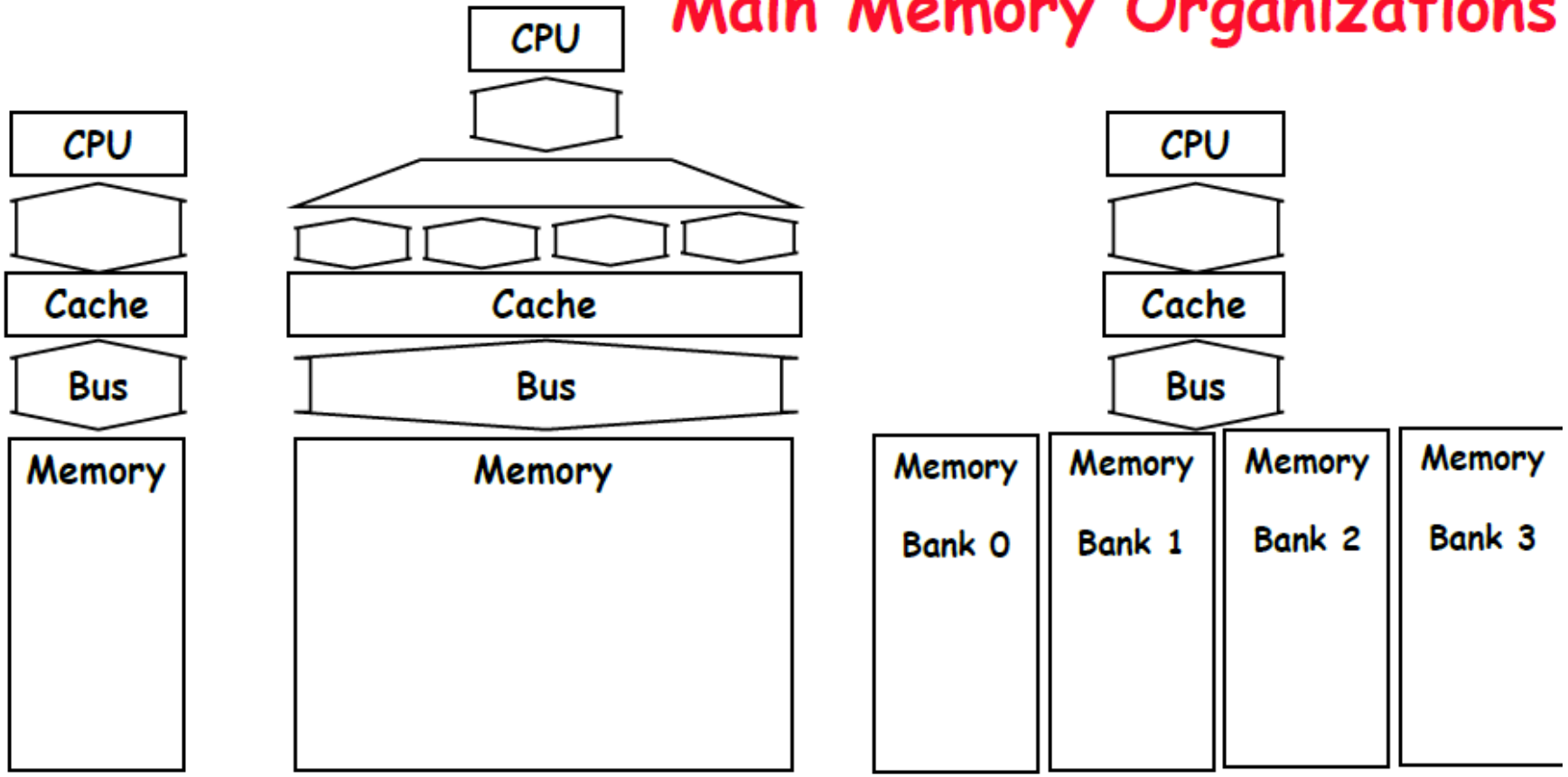
- **Performance of Main Memory**
  - *Latency: Cache Miss Penalty*
  - *Access Time: time between request and word arrives*
  - *Cycle Time: time between requests*
  - *Turn around time (write-read)*
  - *Burst read/write mode*
  - *Bandwidth: I/O & Large Block Miss Penalty (L2)*

- **Main Memory is DRAM: Dynamic Random Access Memory**
  - *Dynamic since needs to be refreshed periodically (8 ms, 1% time)*
  - *Addresses divided into 2 halves (Memory as a 2D matrix):RAS or Row Access Strobe*
  - *CAS or Column Access Strobe*

- **Cache uses SRAM: Static Random Access Memory**
  - *No refresh (6 transistors/bit vs. 1 transistor)*
  - *Size: DRAM/SRAM 4-8,*
  - *Cost/Cycle time: SRAM/DRAM 8-16*

Memory Organization

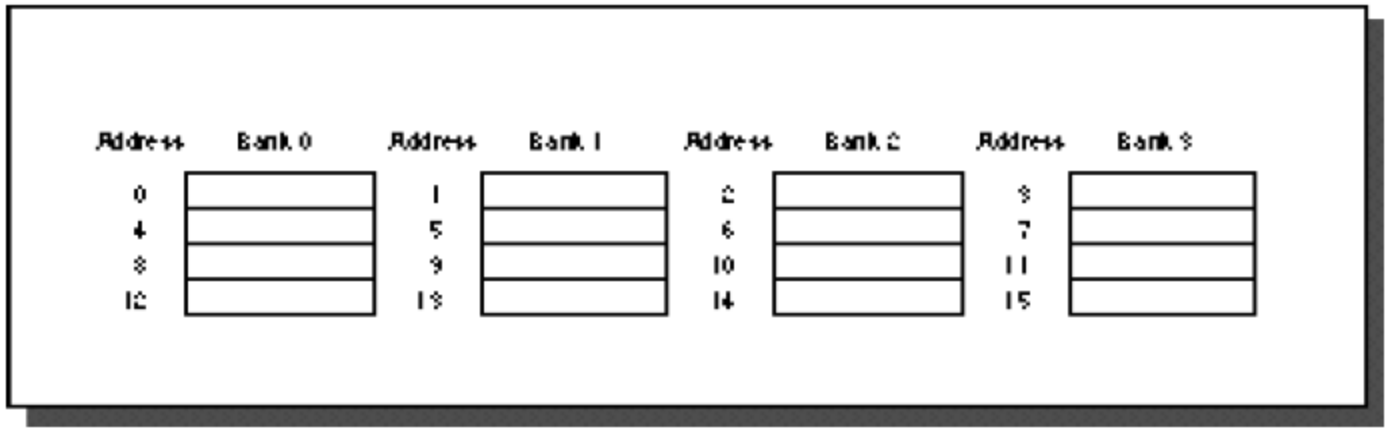
Main Memory Organizations



- *Simple*:
  - CPU, Cache, Bus, Memory same width (32 or 64 bits)
- *Wide*:
  - CPU/Mux 1 word; Mux/Cache, Bus, Memory N words (Alpha: 64 bits & 256 bits; UtraSPARC 512)
- *Interleaved*:
  - CPU, Cache, Bus 1 word: Memory N Modules (4 Modules); example is *word interleaved*

Main Memory Performance

- Timing model (word size is 32 bits)
  - 1 to send address,
  - 6 access time, 1 to send data
  - Cache Block is 4 words
- *Simple M.P.* = 4 x (1+6+1) = 32
- *Wide M.P.* = 1 + 6 + 1 = 8
- *Interleaved M.P.* = 1 + 6 + 4x1 = 11



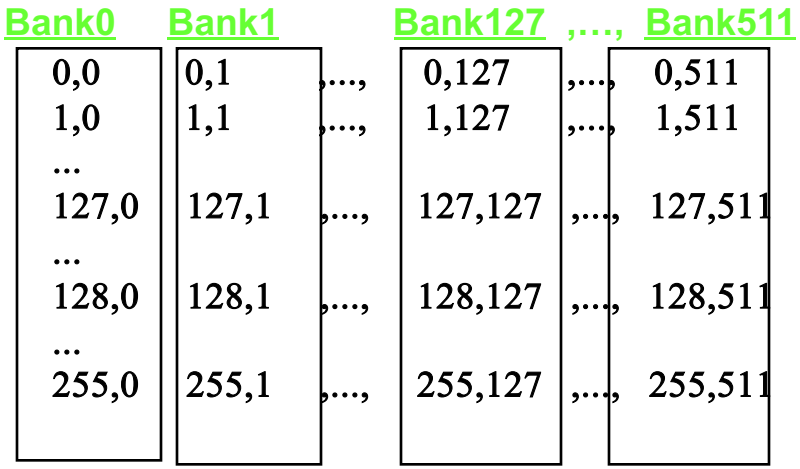
Avoid Bank Conflict

- **Even a lot of banks, still bank conflict in certain regular accesses e.g. Storing 256x512 array in 512 banks and column processing (512 is an even multiple of 128)**

```
int x[256][512];
        for (j = 0; j < 512; j = j+1)
                for (i = 0; i < 256; i = i+1)
                        x[i][j] = 2 * x[i][j];
```

Column processing

Inner Loop is a column processing which causes bank conflicts

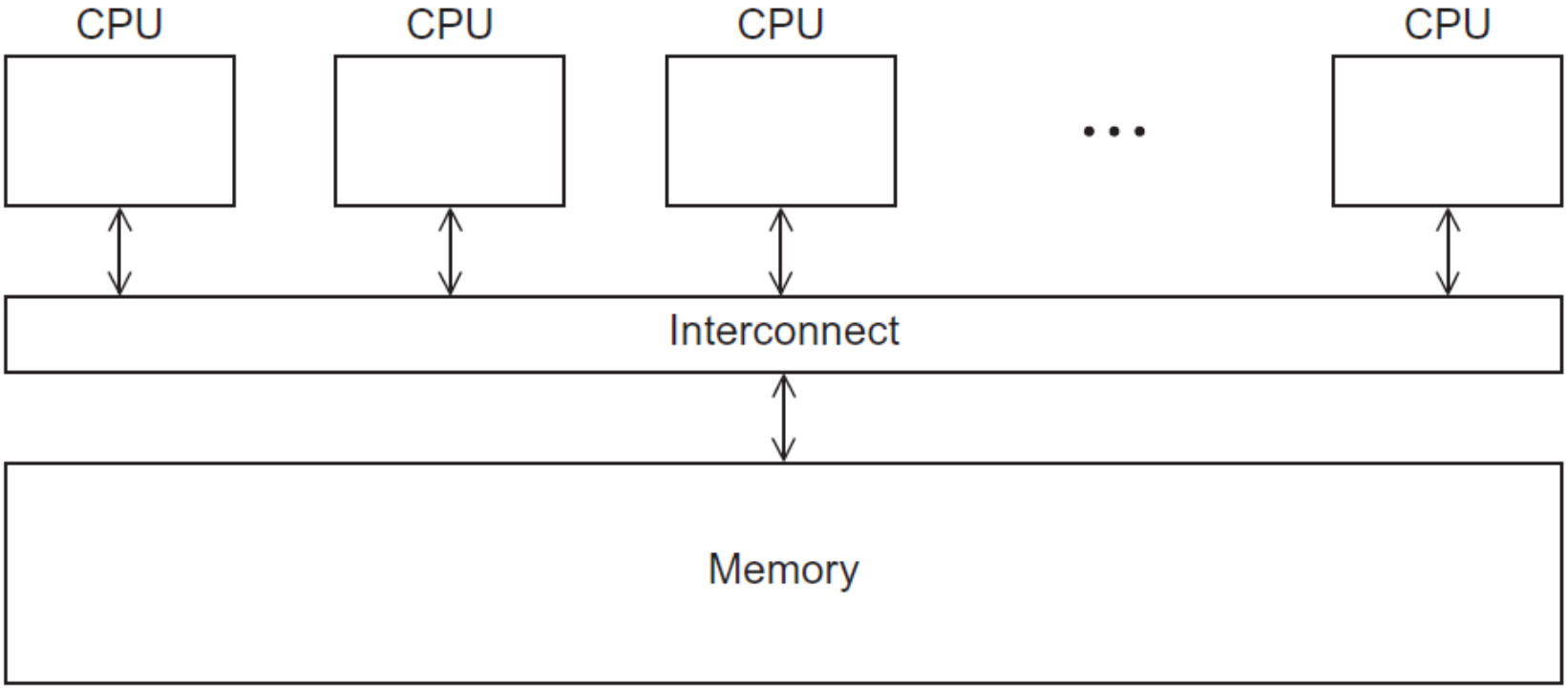


Column elements are in the same bank

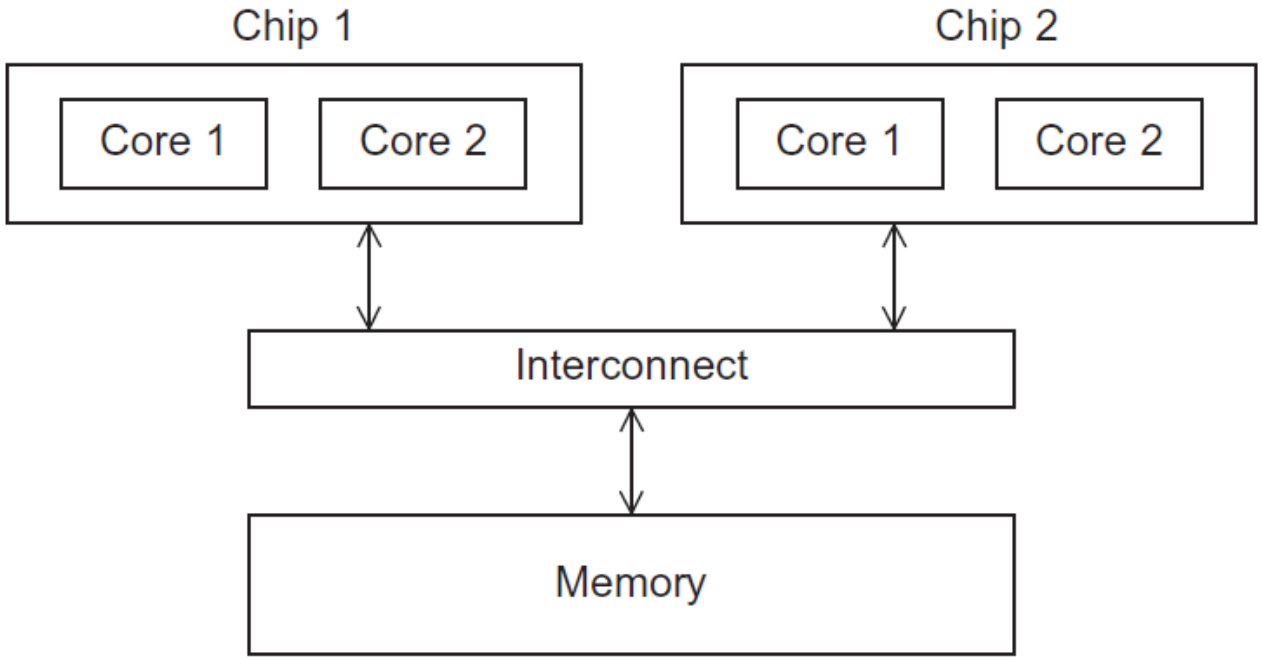
Two Major Classes

- **Shared memory multiprocessor architectures**
  - *A collection of autonomous processors connected to a memory system.*
  - *Supports a global address space where each processor can access each memory location.*
- **Distributed memory architectures**
  - *A collection of autonomous systems connected by an interconnect.*
  - *Each system has its own distinct address space, and processors must explicitly communicate to share data.*
  - *Clusters of PCs connected by commodity interconnect is the most common example.*

Shared Memory System



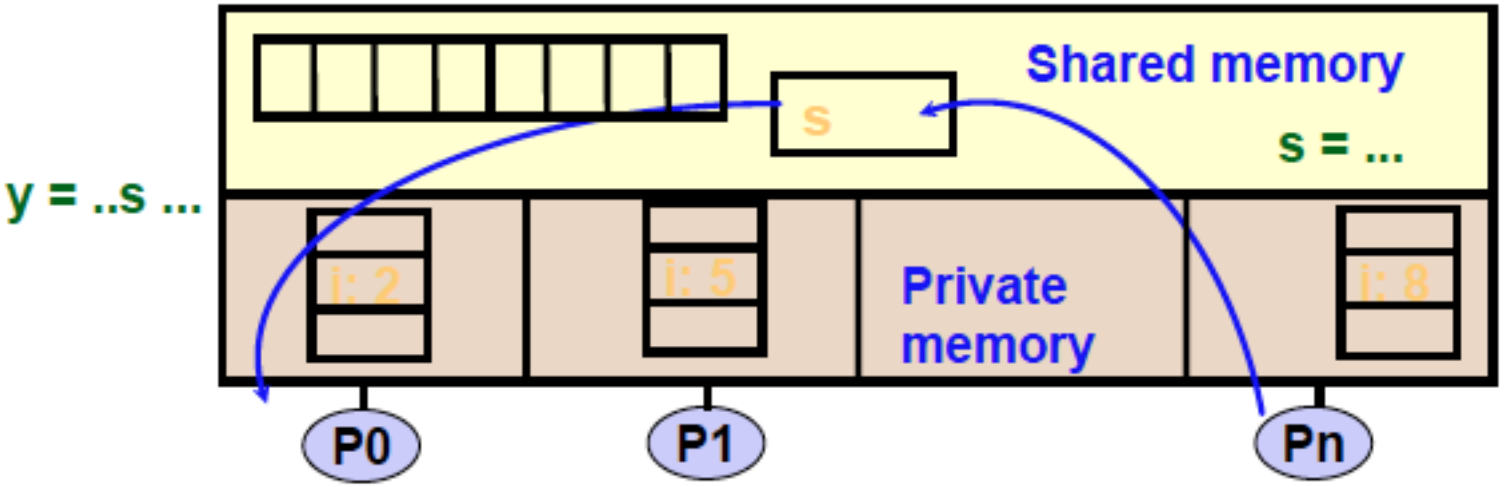
UMA Multicore System



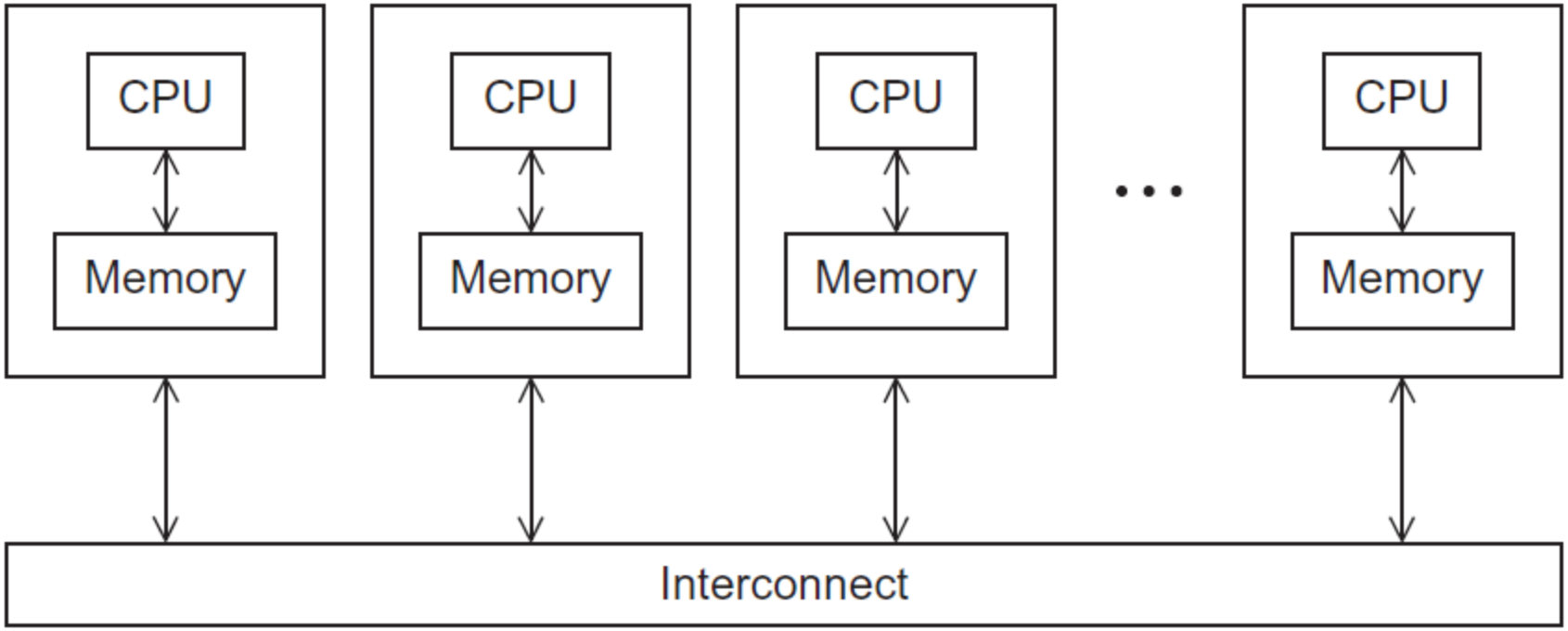
Uniform Memory Access: Time to access all the memory locations will be the same for all the cores.

Programming Abstraction

- A shared-memory program is a collection of threads of control.
- Each thread has private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
- Threads communicate implicitly by writing and reading shared variables.
- Threads coordinate through locks and barriers implemented using shared variables.

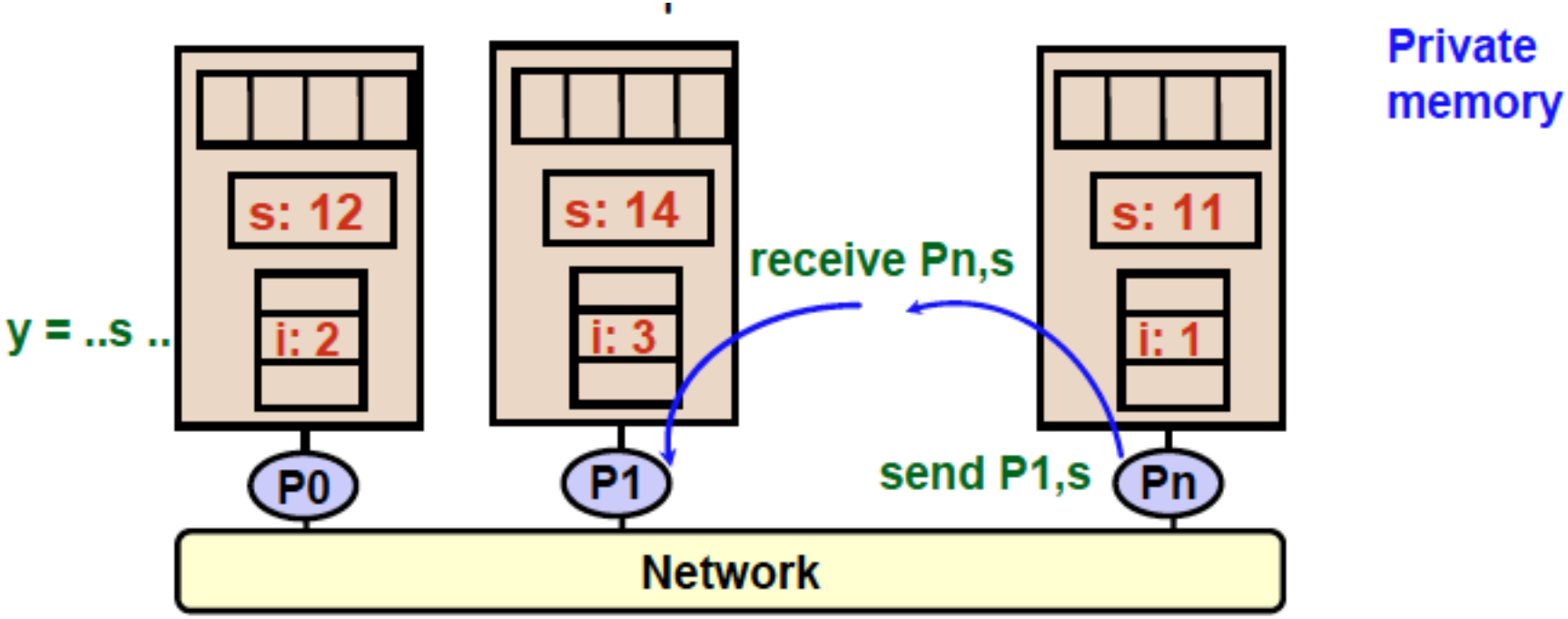


Distributed Memory System



Programming Abstraction

- Distributed-memory program consists of named processes.
- Process is a thread of control plus local address space -- NO shared data.
- Logically shared data is partitioned over local processes.
- Processes communicate by explicit send/receive pairs
- Coordination is implicit in every communication event.



Interconnects

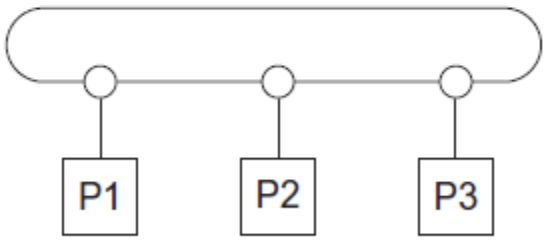
- Affects performance of both distributed and shared memory systems.

- Two categories
  - *Shared memory interconnects*
  - *Distributed memory interconnects*

Shared Memory Interconnects

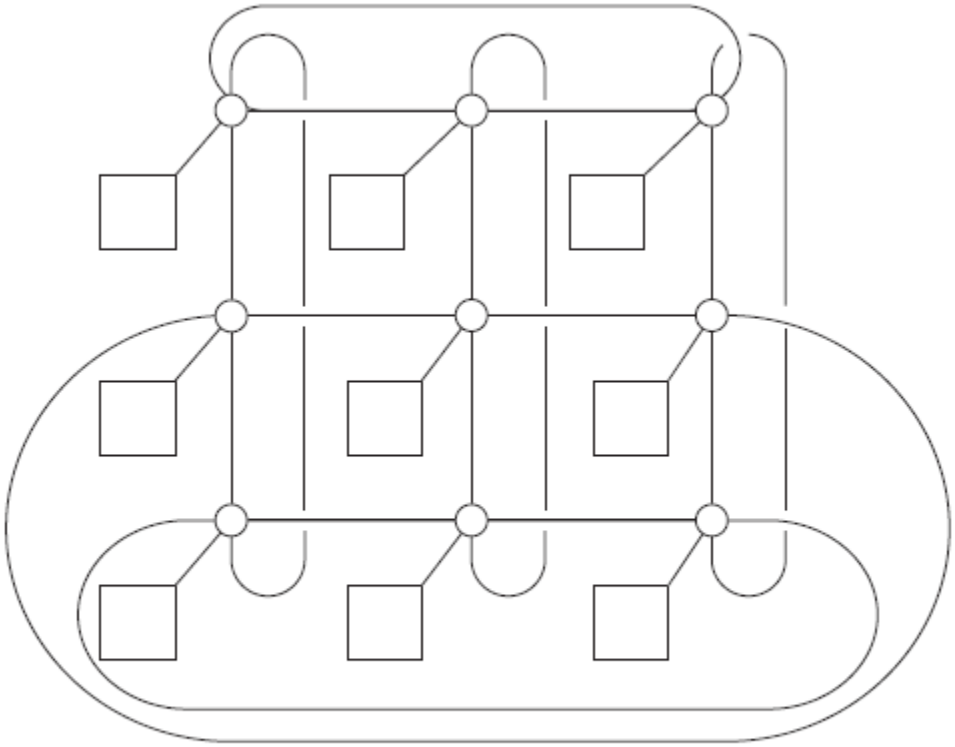
- Bus interconnect
  - *A collection of parallel communication wires together with some hardware that controls access to the bus.*
  - *Communication wires are shared by the devices that are connected to it.*
  - *As the number of devices connected to the bus increases, contention for use of the bus increases, and performance decreases.*
- Switched interconnect
  - *Uses switches to control the routing of data among the connected devices.*

Distributed Memory Interconnects



(a)

ring



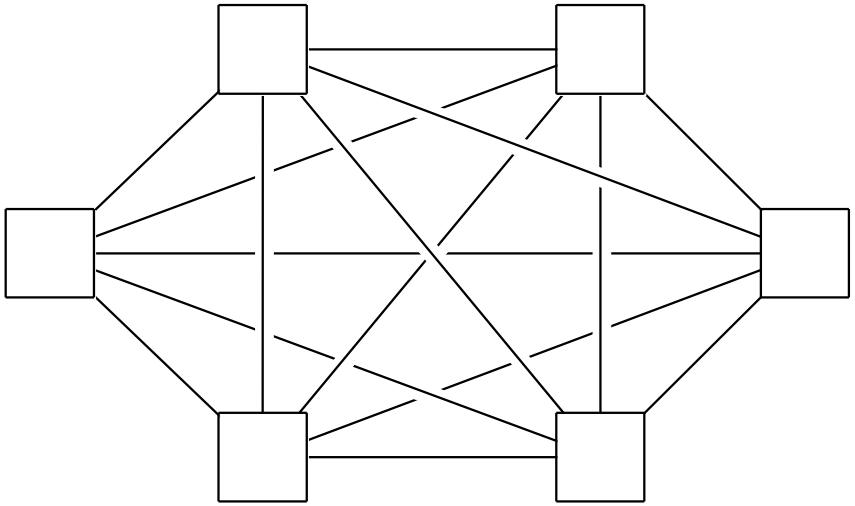
(b)

toroidal mesh

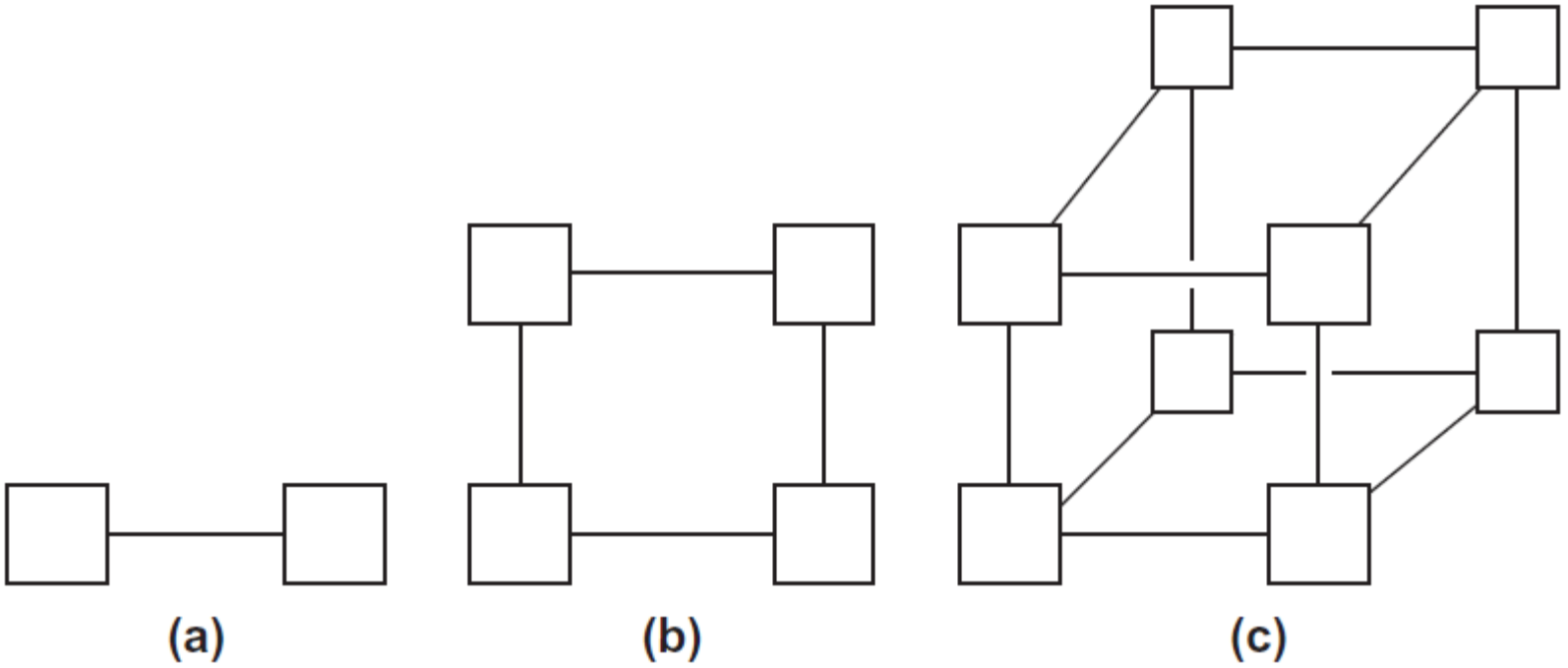
Fully Connected Interconnects

- Each switch is directly connected to every other switch.

impractical



Hypercubes



one- two- three-dimensional

Network Parameters

Any time data is transmitted, we're interested in how long it will take for the data to finish transmission.

- Latency
  - *The time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte.*
- Bandwidth
  - *The rate at which the destination receives data after it has started to receive the first byte.*

Data Transmission Time

Message transmission time = l + n / b