Chapter 6

Parallel Processing

Courtesy： Rami Melhem

Evolution of parallel hardware

- I/O channels and DMA
- Pipelined functional units
- Vector processors (ILLIAV IV was built in 1974)

- Multiprocessors (cm* and c.mmp were built in the 70's)
- Instruction pipelining and superscalers

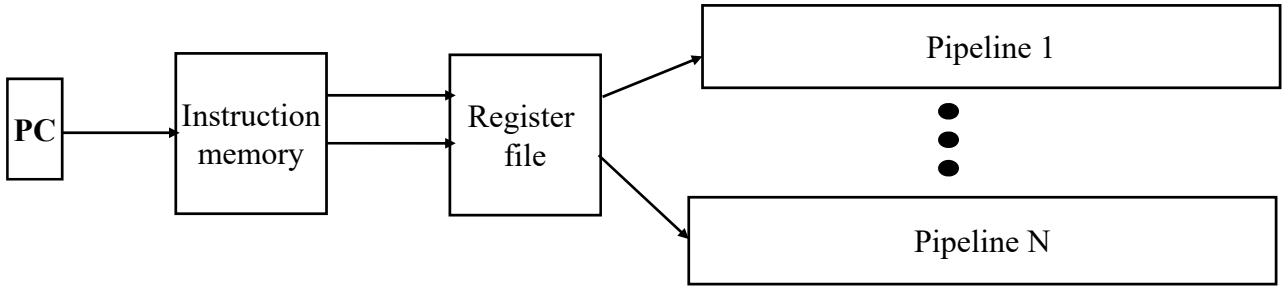
- Supercomputers - Massively Parallel Processors (Connection machine, T3E, Blue Gene, …)
- Symmetric Multiprocessors (SMPs)

- Distributed computing (Clusters, server farms, grids, clouds)

- Multi-core processors and Chip Multiprocessors
- Graphics Processor Units (GPU) as accelerators

Pipelining and Instruction Level Parallelism

- Pipelining overlaps various stages of instruction execution
- May use multiple pipelines → VLIW and Superscalers

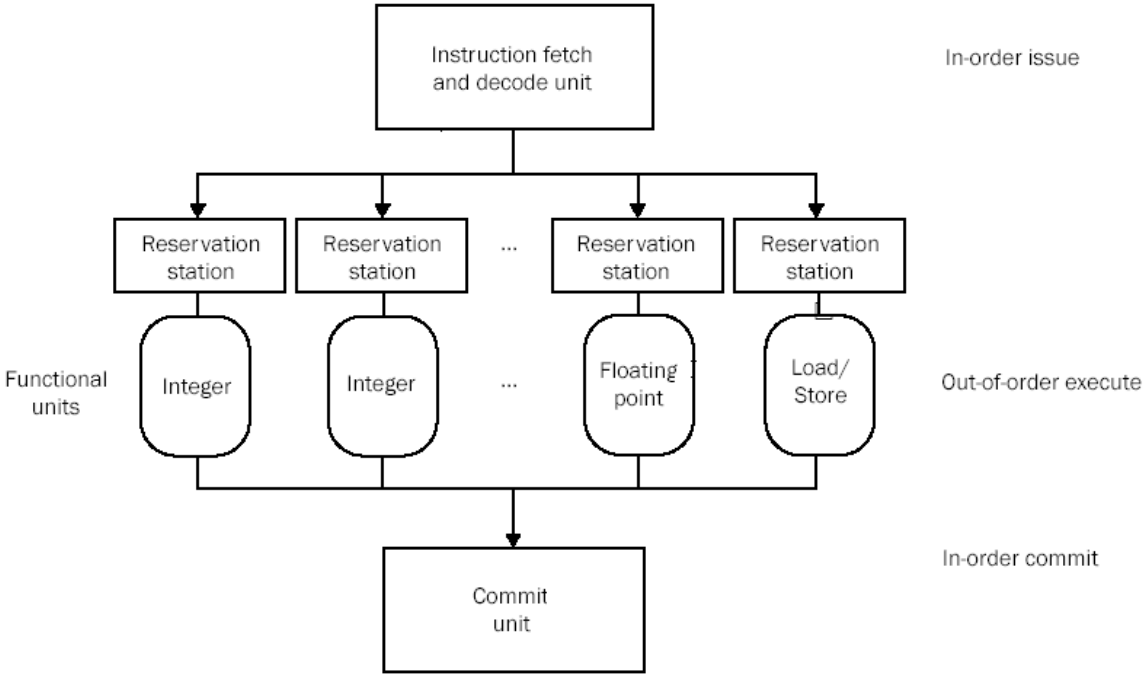


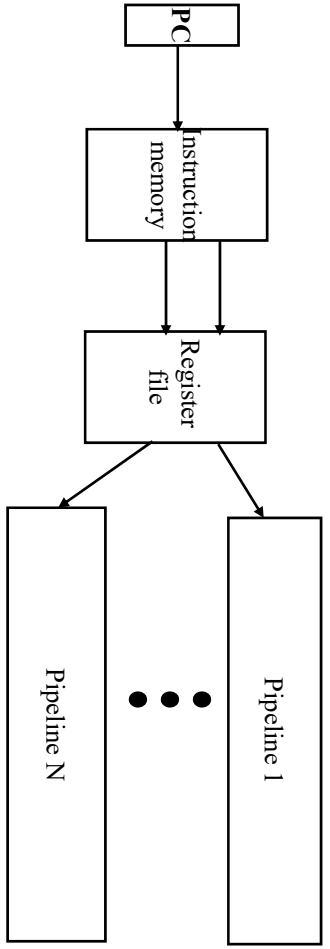
- Pipelining, however, has several limitations.
  - The speed of a pipeline is limited by the slowest stage.
  - Data and structural dependencies
  - Control dependencies

- **In-order issue/execution**: If an instruction cannot be issued because of potential hazard, the following instruction(s) cannot be issued.

Superscalar Execution

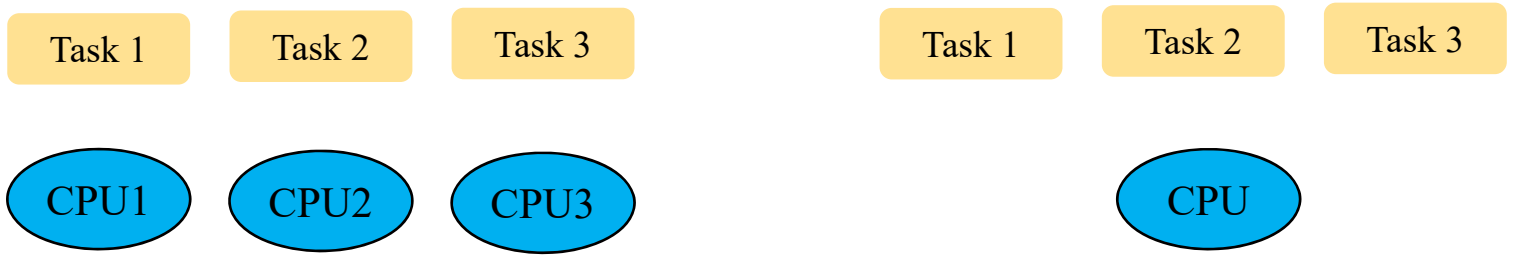
- **Out-of-order execution**: a more aggressive model where instructions can be issued to the pipeline(s) out of order. In this case, if an instruction cannot be issued because a potential hazard, the following instruction(s) can be issued (sometimes called dynamic issued).
- Usually, cannot keep all pipelines busy all the time



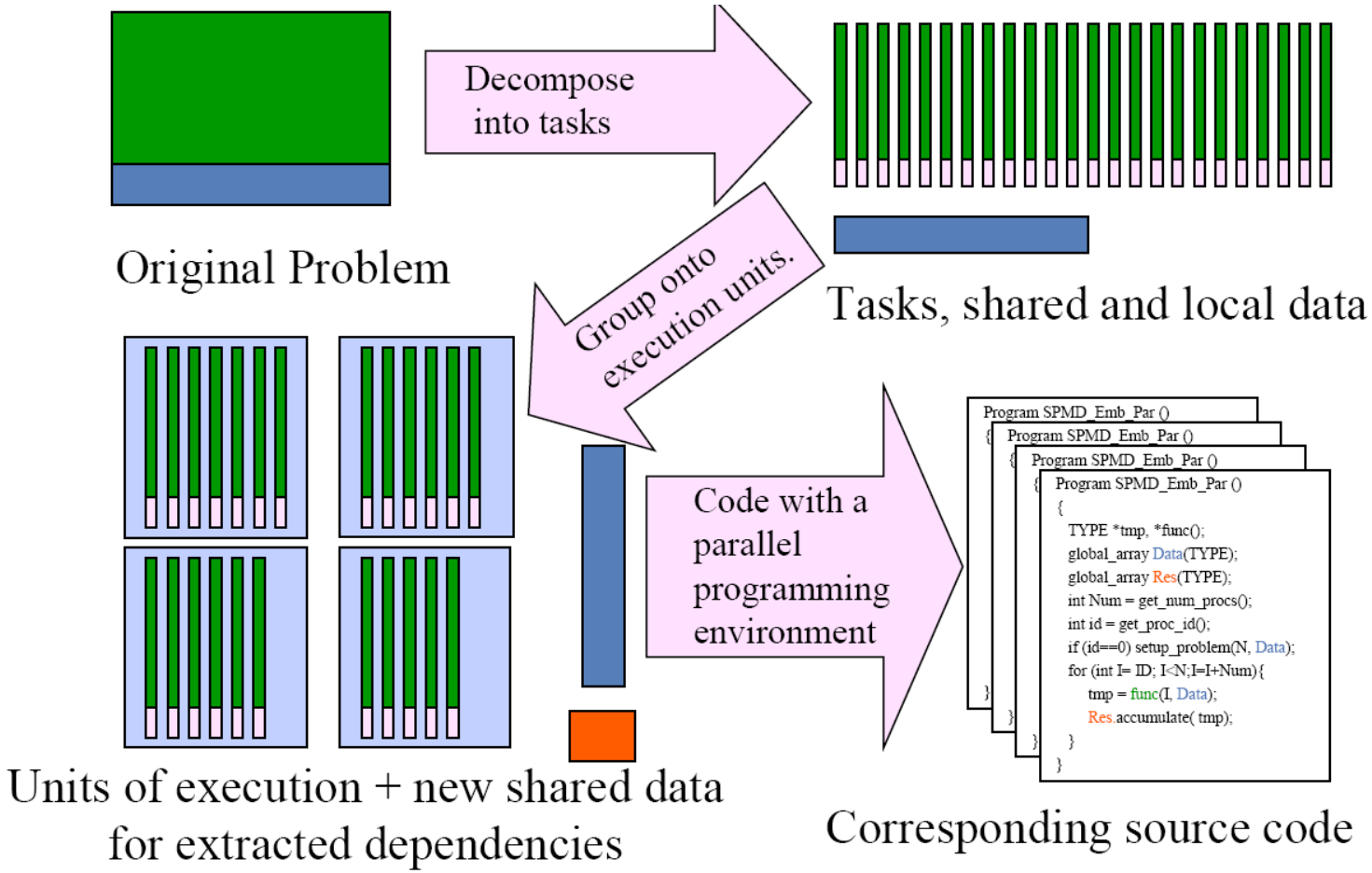


Exploring System Level Parallelism

- **Why ?**
  - ILP (Instruction Level Parallelism) is limited
  - Power consumption limits the increase in clock frequency
- **Multi-tasking:**
  - Divide your task into multiple sub-tasks to run on multiple CPUs.
  - Multi-threading is a form of multi-tasking (threads are light weight tasks).
- The number of tasks (threads) does not have to be equal to the number of CPU's – can multiplex tasks (threads) on a CPU.



How to create parallel applications

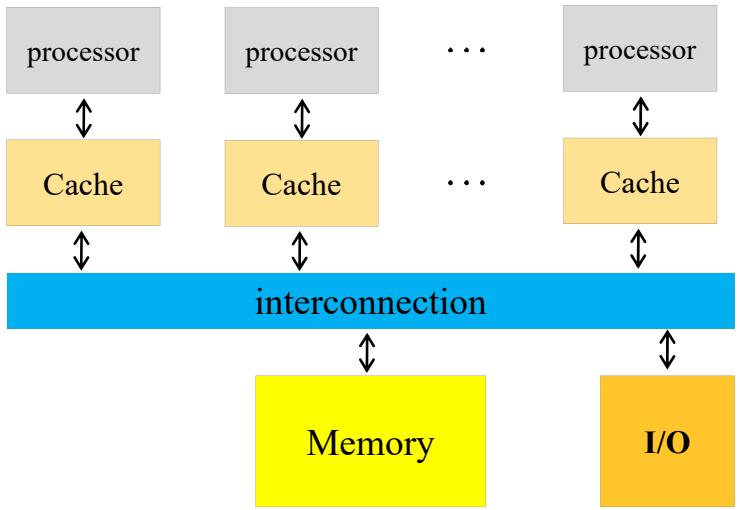


- Creation of multiple tasks (threads):
  - Automatically (for example, by the compiler)
  - Specified by the user (user needs to think *parallel*)

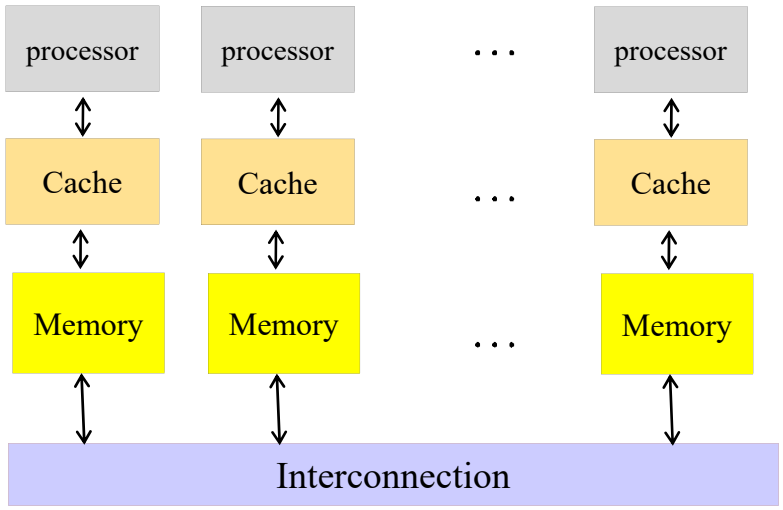
Multiprocessors

- **Idea:** create powerful computers by connecting many smaller ones
  good news: it works
  bad news: it is hard to write correct and efficient concurrent programs.

- Every CS/CoE professional has to deal with parallelism because Chip Multiprocessors are now the norm



SMP - Symmetric multiprocessors



Network connected MP

Speedup and efficiency (Section 6.2)

- For a given problem A, of size n, let $T_p(n)$ be the execution time on p processors, and $T_s(n)$ be the execution time (of the best algorithm for A) on one processor. Then,

  Speedup $S_p(n) = T_s(n) / T_p(n)$

  Efficiency $E_p(n) = S_p(n) / p$

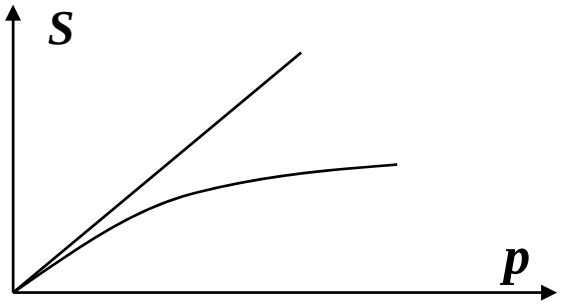
Speedup is between *0* and p, and efficiency is between *0* and *1*.

Linear speedup:

Speedup is linear in p

Minsky's conjecture:

Speedup is logarithmic in p



Speedup and efficiency

Amdahl's law:

If f is the fraction of the task that can be executed in parallel

$T_p = (1\text{-}f) * T_s + f * T_s / p$

Speedup $S_p = \dfrac{1}{(1-f) + \dfrac{f}{p}}$ $\xrightarrow{p \text{ is very large}}$ $= \dfrac{1}{(1-f)}$

Maximum speedup, assuming infinite parallelism

- **Scalability**
  - If can maintain the efficiency for larger p independently of the size of the problem, n, then we have **strong scalability**.
  - If we can maintain the efficiency for larger p only by increasing the size of the problem, then we have **weak scalability.**

Scaling Example 1

- Problem *Dot(n)* → computing the dot product of two vectors $\sum_{i=0}^{n-1} x(i) * y(i)$
- *Dot(1000)* on a single processor: T_s = 1000 (*time of add + time of multiply)*
- *Dot(1000)* on 10 processors (assuming t_{op} = *time of add = time of multiply*)
  - The 1000 multiplications can be done in parallel on the 10 processors
  - The 1000 additions cannot be done in parallel (accumulating 1000 values)
  - $T_{p=10}$ = 1000/10 × t_{op} +1000 × t_{op} = 1100 × t_{op}
  - Speedup, S_{10} = 2000/1100 = 1.82 → (efficiency = 18.2%)
- *Dot(1000)* on 100 processors
  - Time = 1000/100 × t_{op} + 1000 × t_{op} = 1010 × t_{op}
  - Speedup, S_{100} = 2000/1010 = 1.98 → (efficiency = 2%)
- Amdahl law gives the maximum possible speedup

  f for the above problem is 0.5 → max speedup = 2.

***Dot()* is not strongly scalable**

Scaling Example 2

- Problem *Mat(n)* → add two $n \times n$ matrices then sum the diagonals of the result
- *Mat(10)* on a single processor: $T_s = (100 + 10) \times t_{op}$
- *Mat(10)* on 10 processors
  - The addition of two matrices can be done in parallel
  - The summation of 10 diagonal elements cannot be done in parallel
  - $T_{p=10} = 100/10 \times t_{op} + 10 \times t_{op} = 20 \times t_{op}$
  - Speedup, $S_{10} = 110/20 = 5.5$ (efficiency = 55%)
- *Mat(10)* on 100 processors
  - $T_{p=100} = 100/100 \times t_{op} + 10 \times t_{op} = 11 \times t_{op}$
  - Speedup, $S_{100} = 110/11 = 10$ (efficiency = 10%)

Mat() is not strongly scalable

- Note: can use Amdahl law to find the maximum possible speedup
  - f for *Mat(10)* is 100/110 → max speedup = 11.

Scaling Example 2 (cont.)

- *Mat(100)* → same problem but when matrix size is 100 × 100.
  - Single processor: $T_s = (10000 + 100) \times t_{op}$
  - p = 10 processors
    - Speedup, S_{10} = 10100/1100 = 9.18 (91.8% efficiency)
  - p = 100 processors
    - Speedup, S_{100} = 10100/200 = 50.5 (50.5% efficiency)

***Mat()* is not strongly scalable**

- **However:**
  - *Mat(10)* on 10 processors → S_{10} = 5.5 – (55% efficiency)
  - *Mat (100)* on 100 processors → S_{100} =50.5 – (50.5% efficiency)
  - The efficiency of *Mat (n)* at *n=10* and *p=10* can be (almost) maintained at *p=100* if we increase *n* to *100*. Hence, *Mat(n)* is **weakly scalable**.

***Mat()* is weakly scalable**

Flynn's hardware taxonomy (Section 6.3)

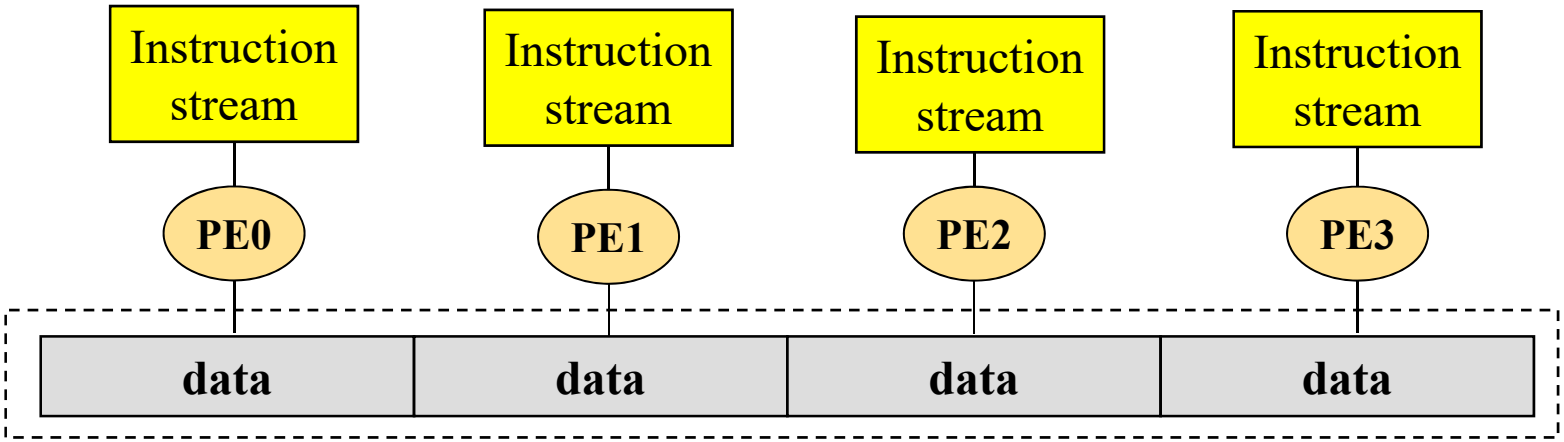
Looks at instructions and data parallelism. Oldest (1960's) and best known of many taxonomy proposals.

S M	I	S M	D

- S for single
- M for multiple
- I for instruction
- D for data.

- SISD is a sequential computer.
- SIMD: one stream of instructions applied to multiple data.
- MIMD: multiple streams of instructions executing on multiple data.
- MISD – need to be innovative to define it.

MIMD



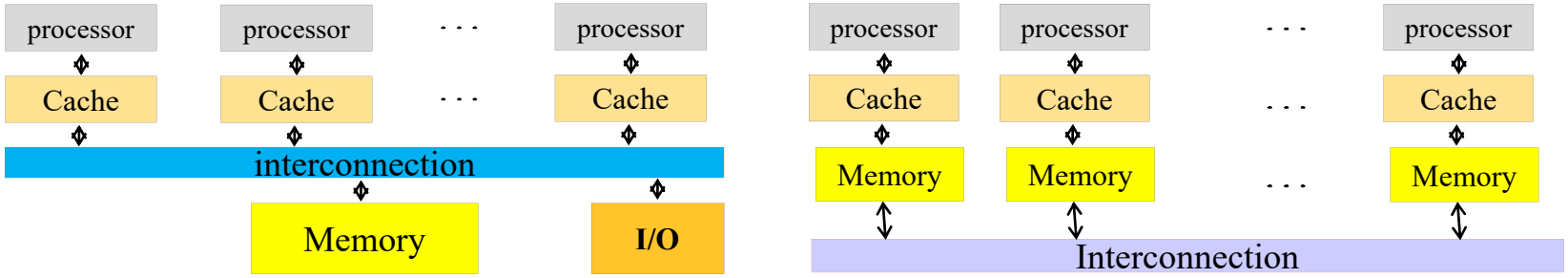
Multiple programs/threads executing on different data – However, if all PEs (processing elements) are to cooperate to solve the problem (as opposed to solving different problems), there should be interaction between the PEs.

Shared address space Vs **separate address spaces (an architecture concept)**

- The address space of an instruction stream executing on a processor consists of the "virtual" memory addresses that can be accessed from lw/sw instructions.
- Two instruction streams that do not share a memory address space can share information through *message passing*.

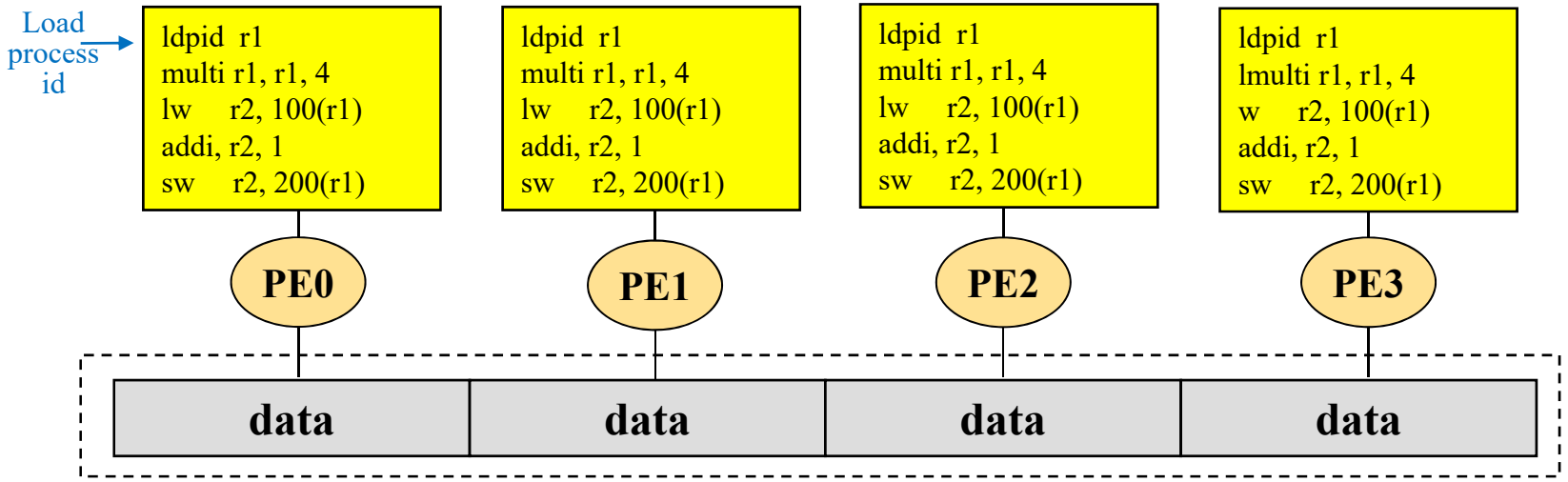
MIMD

- **Virtual addresses are mapped to physical memory locations**
  - The hardware memory system may have shared physical memory modules or distributed physical memory modules
  - Can have shared virtual address spaces on either a shared or distributed physical memory (same applies to separate virtual address spaces)



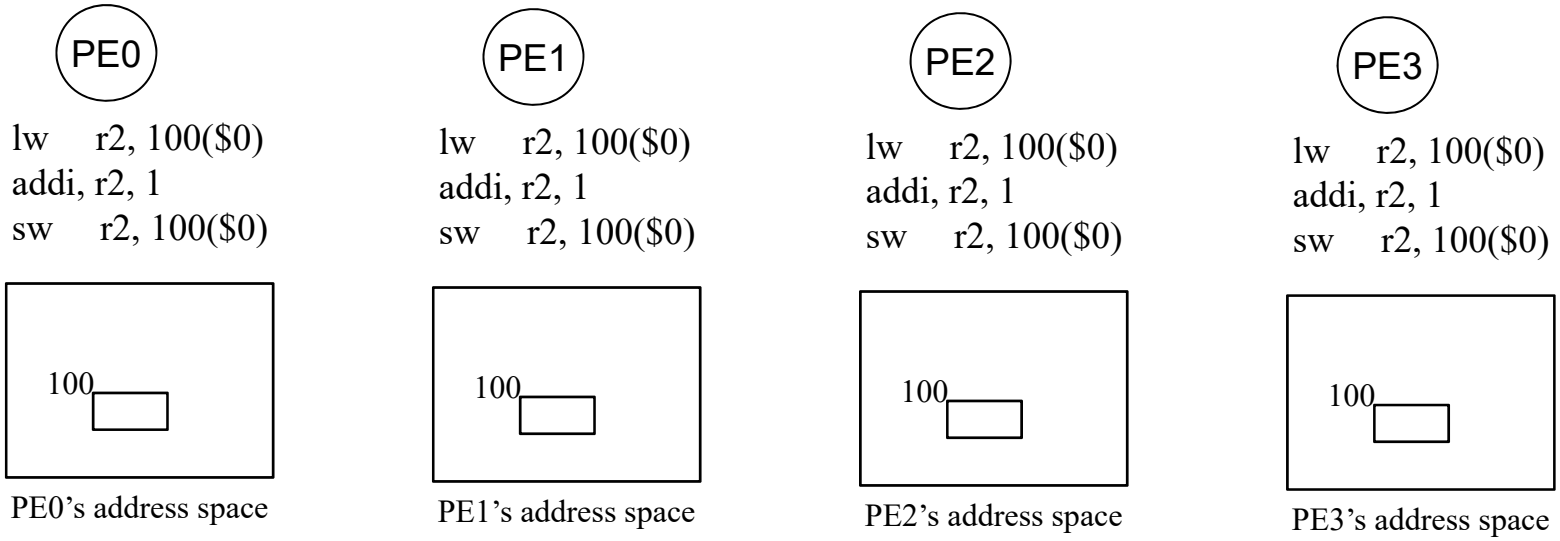
- **Uniform memory access, UMA** Vs **Non-uniform memory access, NUMA**
  - Does the delay for accessing a memory location depend on its address?.
- **Shared memory programming** Vs **distributed memory programming**
  - Variables can be shared (global) → shared memory programming
  - Variables are private (local) → distributed memory programming
  - Shared memory programming allows private as well as shared variables

The concept of SPMD

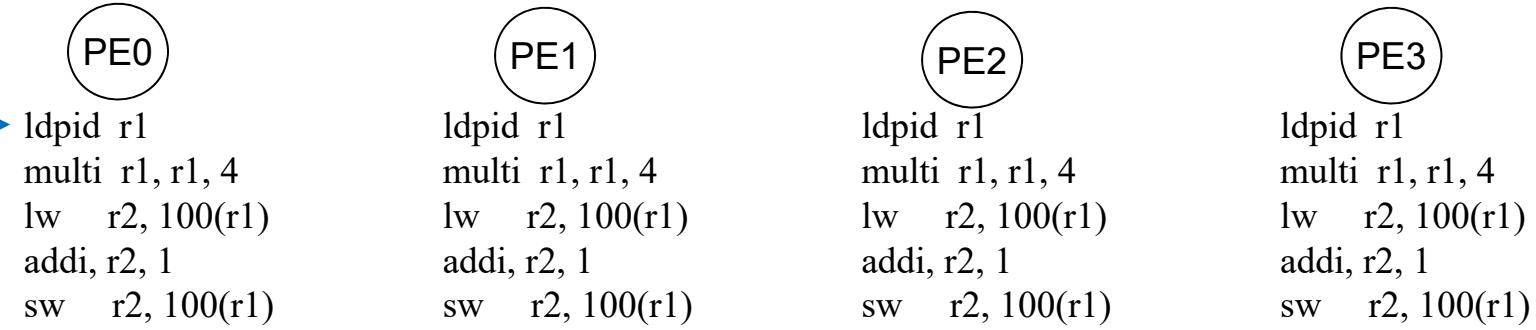


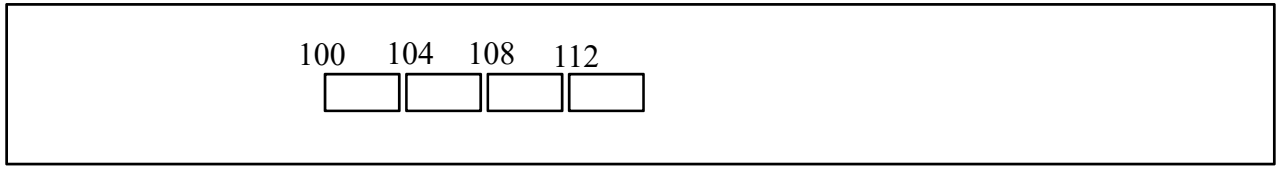
- **The concept of single Program Multiple Data (SPMD):**
  (applies to both distributed memory and shared memory MIMD programming)
  - User writes one program to be executed by all processors (threads).
  - How do you make the program do different things?

Shared Vs distributed address space

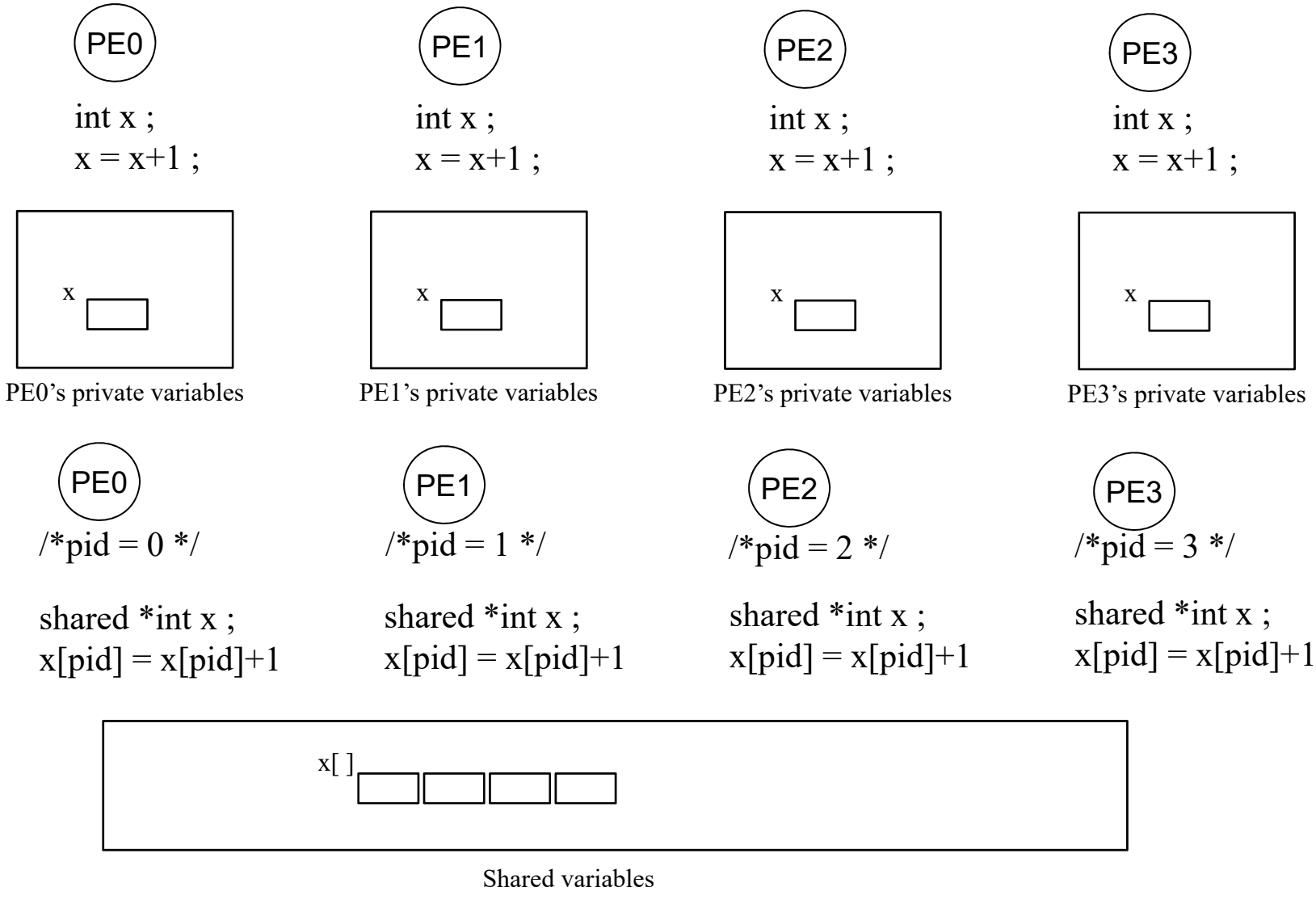


Load process id →



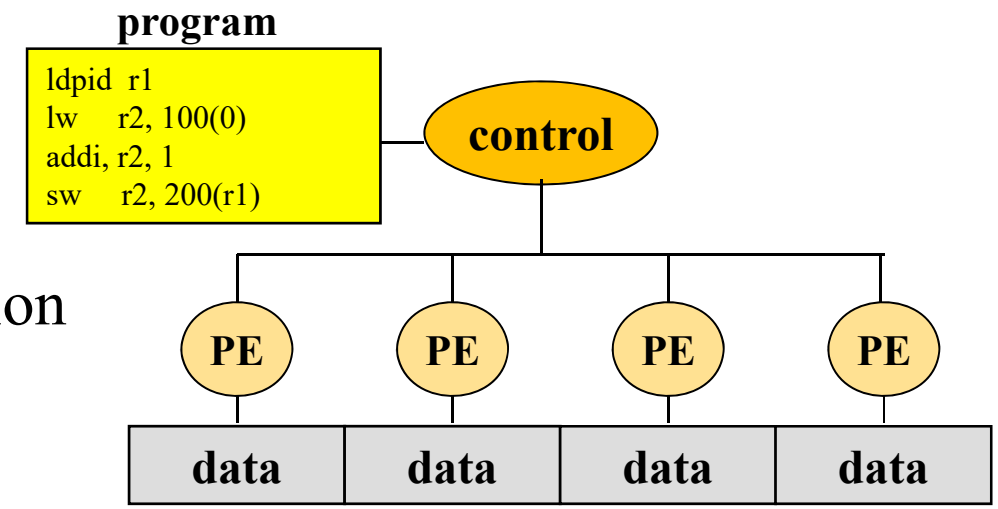


Programming with private and shared (global) variables



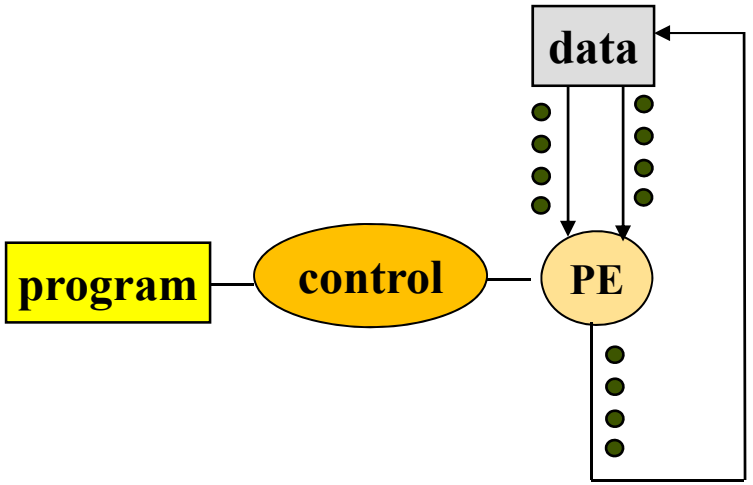
Note: in languages that allow shared variables, a variable not declared "shared" is private

SIMD (two flavors)



1) Synchronous, lockstep execution

All PEs execute the same instructions on different data

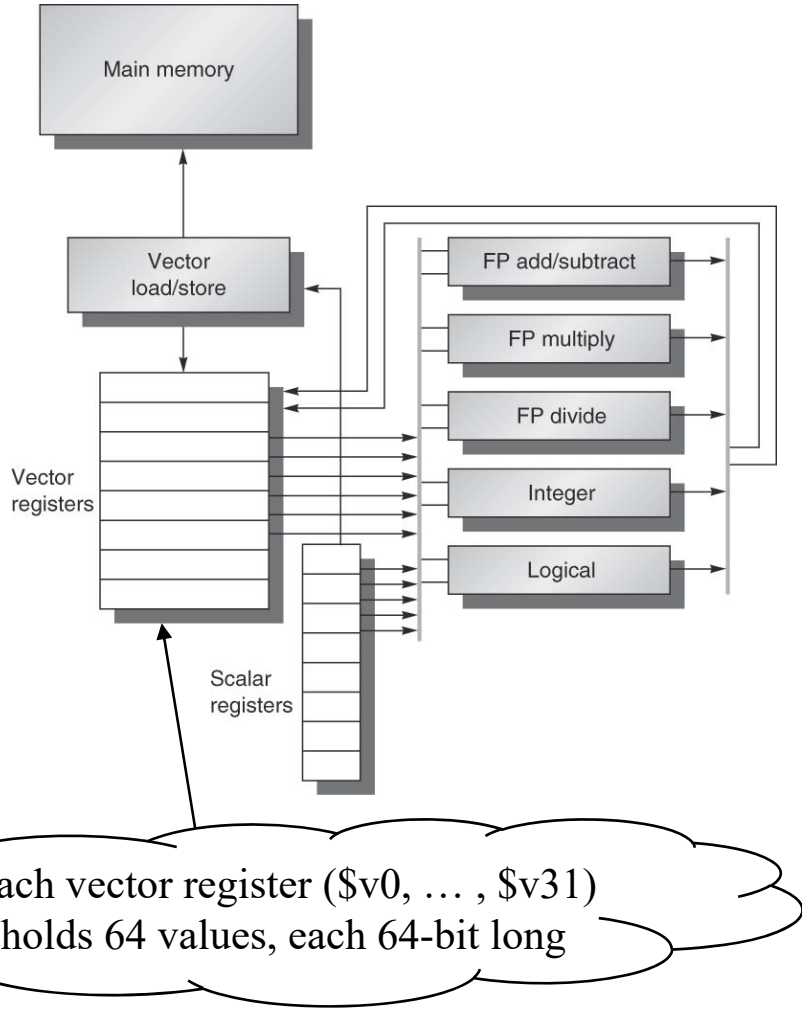


2) Vector processing

The same instruction is repeatedly executed on different data

Vector Processors

- **Example**: Vector extension to MIPS
  - The usual 32 integer registers ($0, … , $31)
  - 32 floating point registers ($f0, … , $f31)
  - 32 vector registers ($v0, … , $v31)
- Move vectors from/to memory
  - lv → an instruction to load a vector of data from memory into a vector registers
  - sv → an instruction to store a vector from a vector register to memory
- Vector instructions to stream data from vector registers to highly pipelined functional units
  - addv.d → add two vectors
  - addvs.d → add a scalar to each element of a vector



Significantly reduces instruction fetch and execution time

EX: compute $y(i) = a * x(i) + y(i), i = 0, \ldots, 63$

- Conventional MIPS code (assuming 64-bit architecture, i.e. a word = 8 bytes).

```
      l.d   $f0,0($sp)      ;load scalar a to $f0
      addi  $s2,$s0,512     ;64 elements (64*8=512 bytes)
loop: l.d   $f2,0($s0)      ;load x(i) into $f2
      mul.d $f2,$f2,$f0     ;multiply a and x(i)
      l.d   $f4,0($s1)      ;load y(i) into $f4
      add.d $f4,$f4,$f2     ;add y(i) to a x(i)
      s.d   $f4,0($s1)      ;store back into y(i)
      addi  $s0,$s0,8       ;increment index to x
      addi  $s1,$s1,8       ;increment index to y
      subu  $t0,$s2,$s0     ;# of elements left to process
      bne   $t0,$zero,loop  ;loop if not done
```

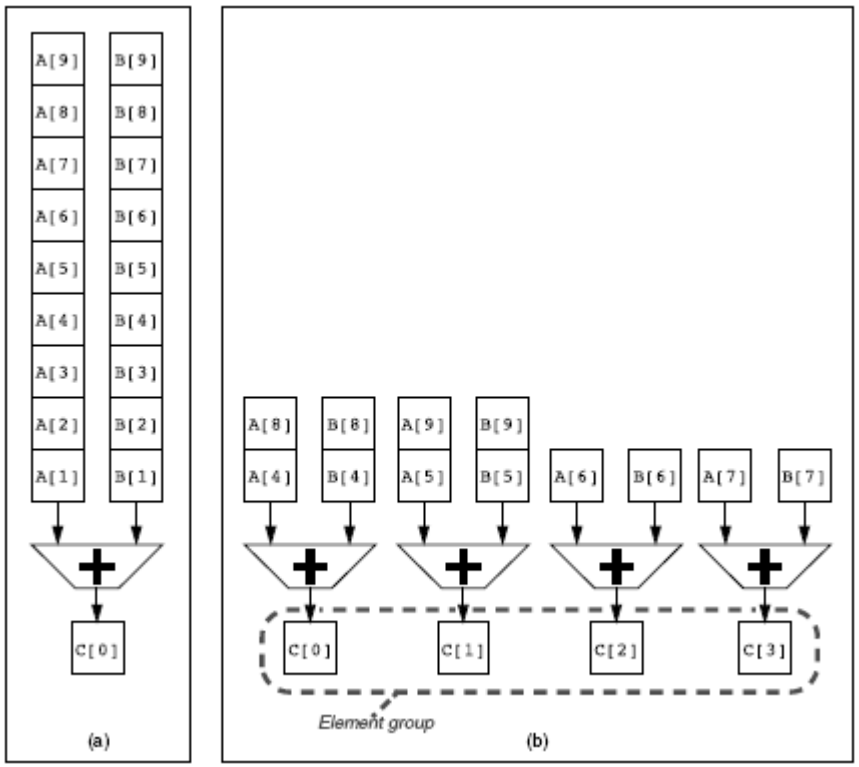
- Vector MIPS code

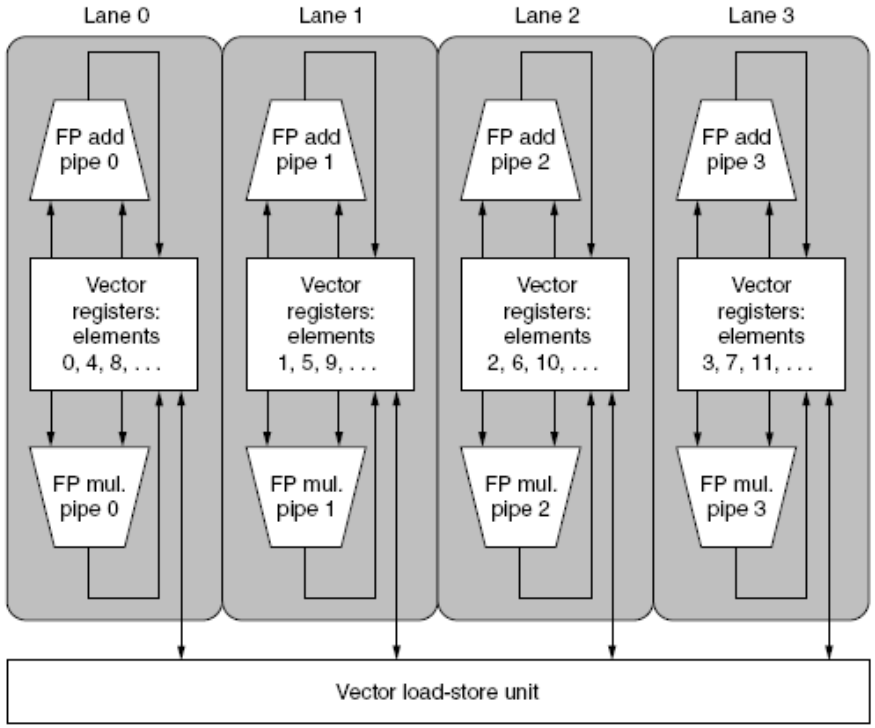
```
      l.d     $f0,0($sp)    ;load scalar a to $f0
      lv      $v1,0($s0)    ;load vector x (64 values) to $v1
      mulvs.d $v2,$v1,$f0   ;multiply vector x by scalar a
      lv      $v3,0($s1)    ;load vector y (64 values) to $v3
      addv.d  $v4,$v2,$v3   ;add two vectors
      sv      $v4,0($s1)    ;store back the result vector
```

Using multiple Lanes

Instead of using one pipelined functional unit for all the vector elements, multiple units can be used, in parallel.

EXAMPLE: 4 pipeline units can be used, each operating on 1/4th of the vector





Hardware Multi-threading (Sec. 6.4)

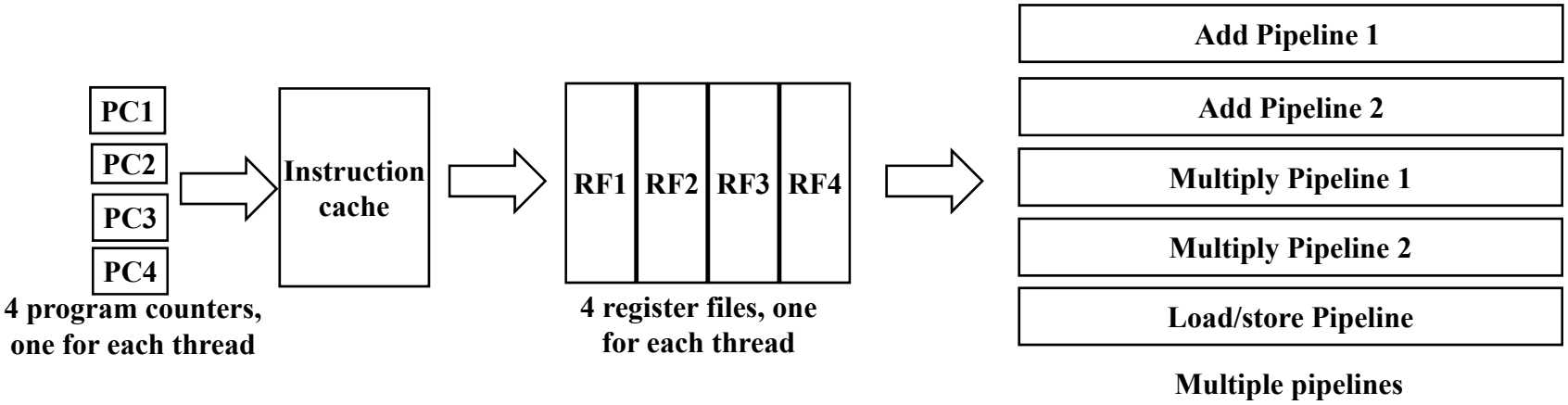
- Software-based thread context switching (Posix Threads)
  - Hardware traps on a long-latency operation
  - Software saves the context of the current thread, puts it on hold and starts the execution of another ready thread
  - Relatively large overhead (saving old context and loading new context)
  - Context = registers, PC, stack pointer, pointer to root page table, ….

- Hardware-based multithreading
  - Threads = user defined threads or compiler generated threads
  - Replicate registers (including PC and stack pointer)
  - Hardware-based thread-context switching (fast)

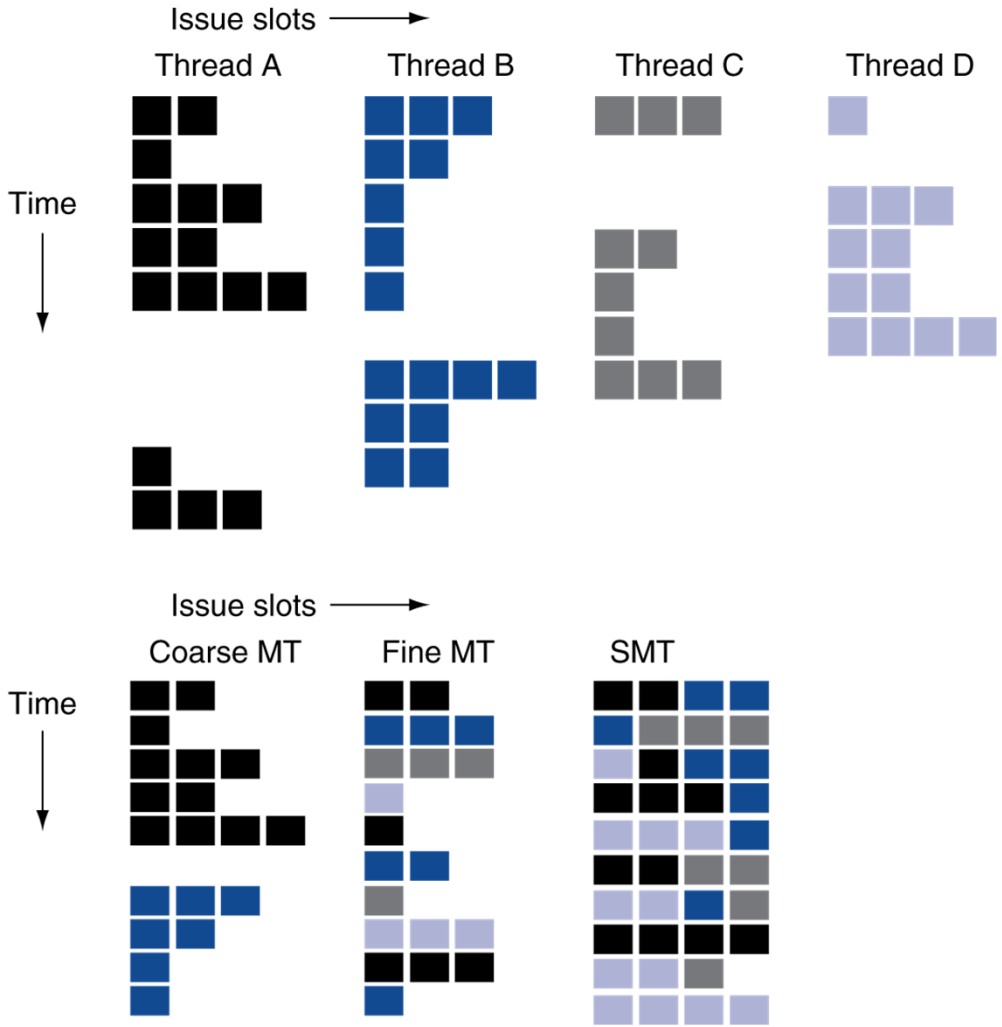
- Example: IBM Power5 and Pentium-4 supports hardware-based multi-threading

Scheduling multiple 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)
- SMT – Simultaneous Multi Threading
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when ready
  - Dependencies within each thread are handled separately



SMT Examples

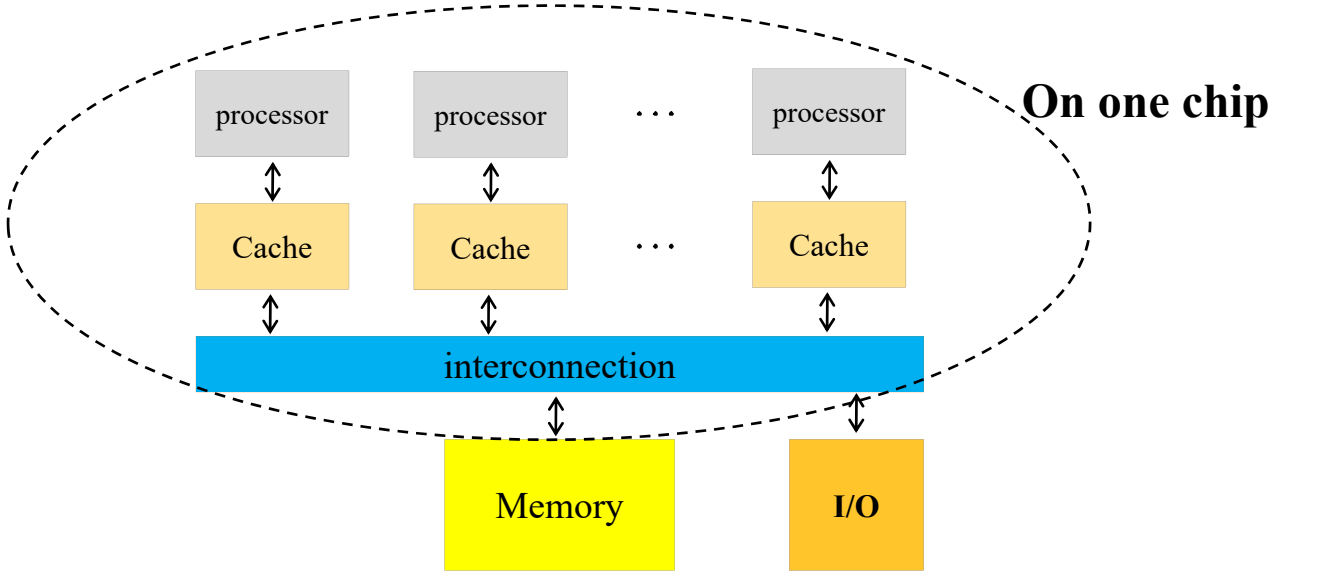


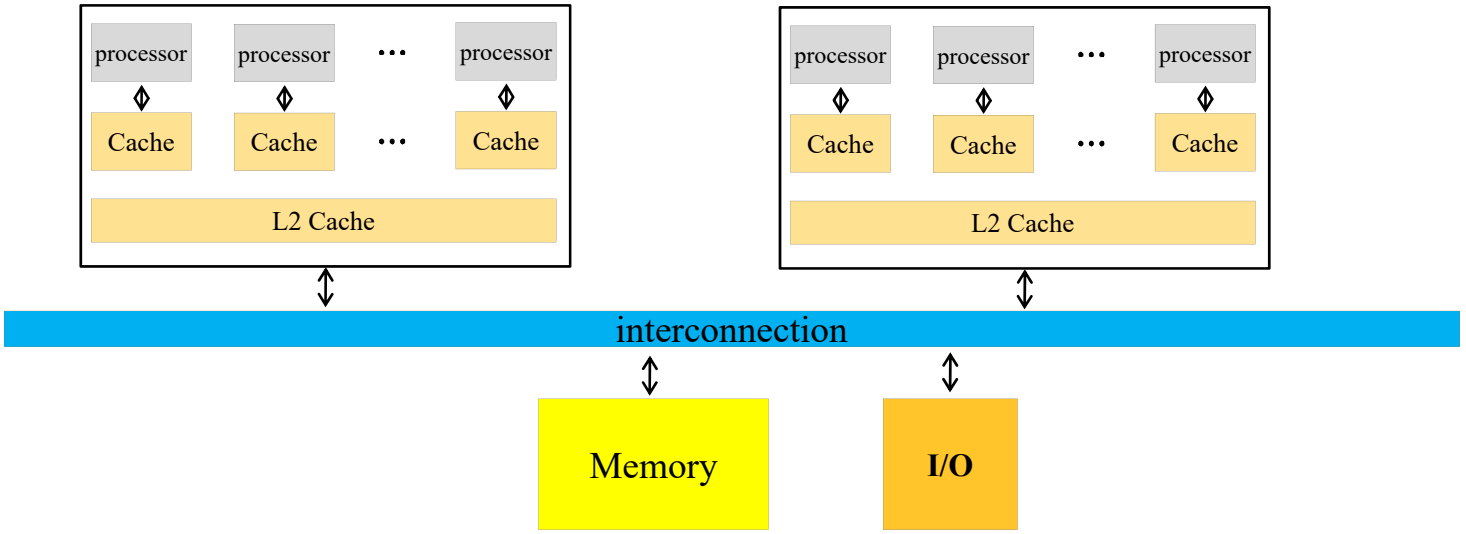
Single thread execution

This example assumes a 4-issue pipeline (can issue as many as 4 instructions every cycle)

Multithread execution

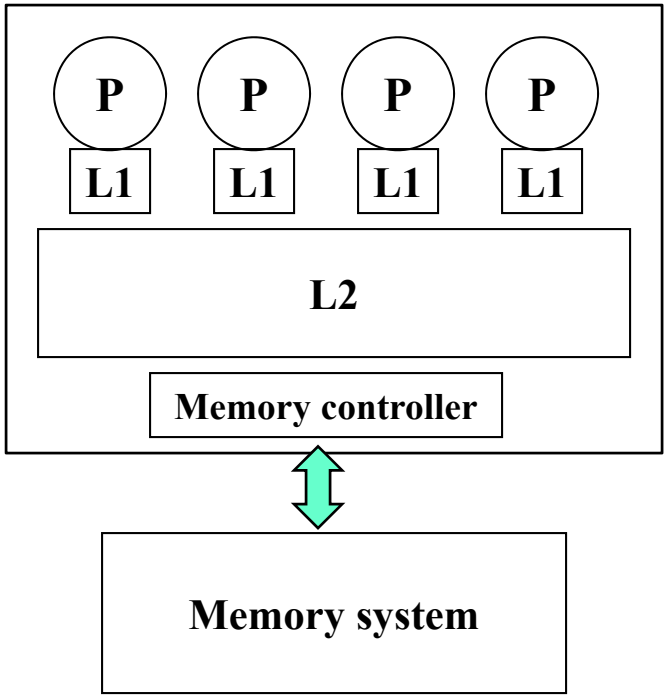
Shared memory systems (CMP, multicores, manycores) (sec. 6.5)





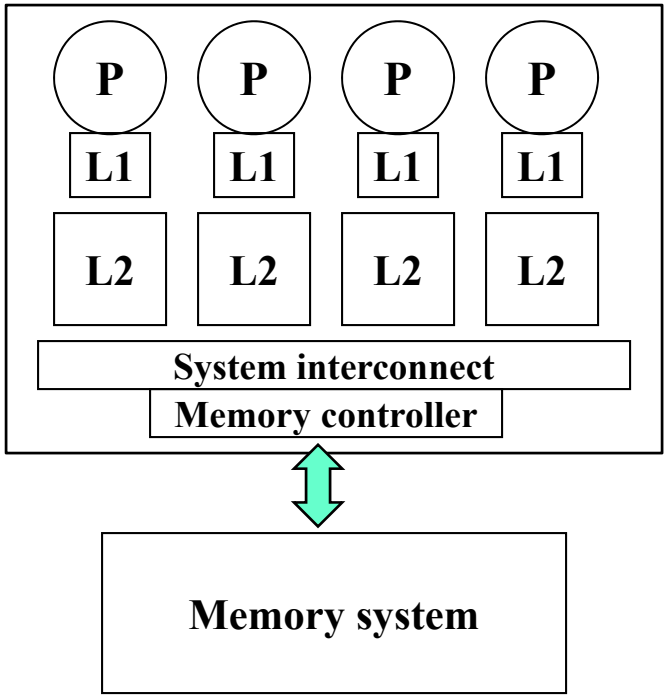
Chip Multiprocessors

Shared L2 systems



- Examples: Intel Pentium

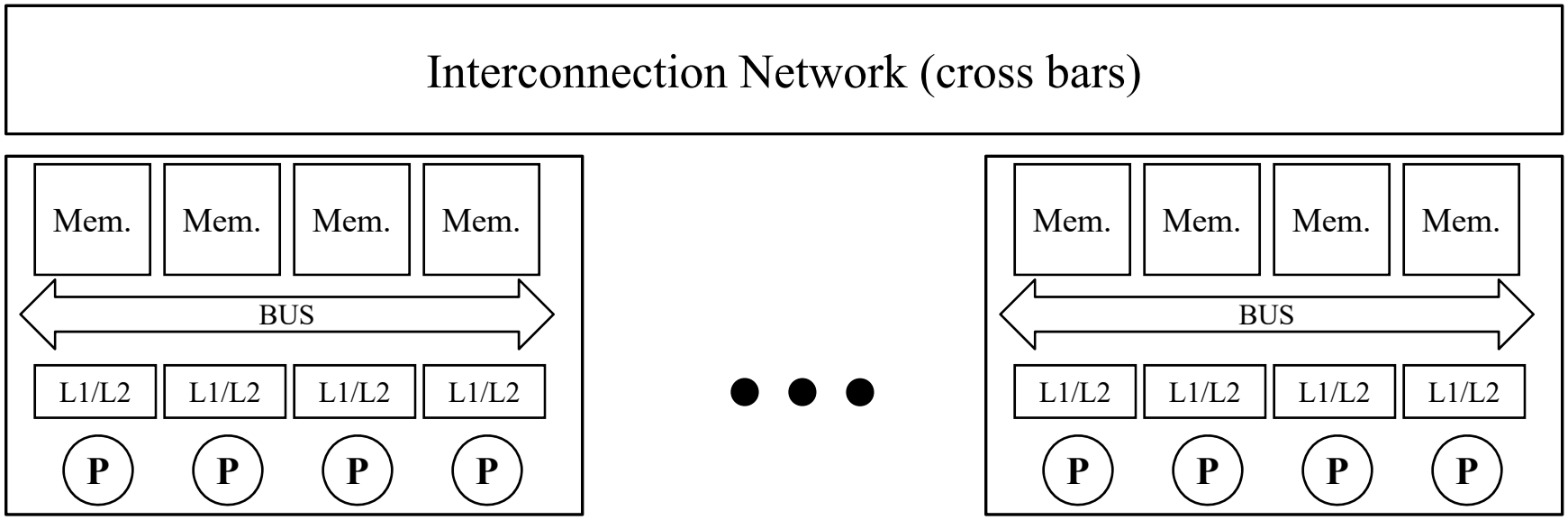
Private L2 systems



- Examples: AMD Opteron

Example: The Sun Fire E25 K

http://www.sun.com/servers/highend/sunfire_e25k/specs.xml



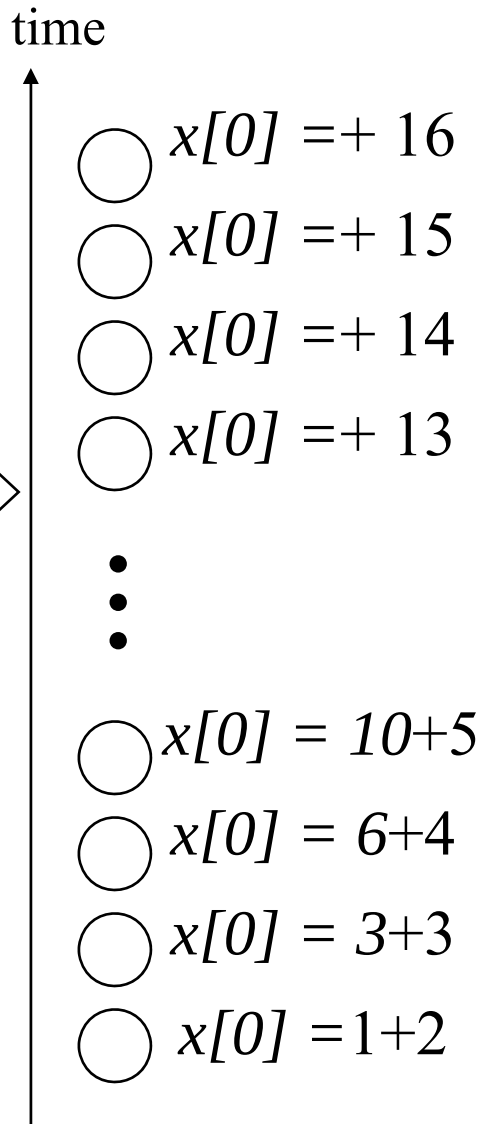
- Board = 4 SPARCS IV + 64 GB memory
- Up to 18 boards connected by crossbars
- 1.15 TB of Distributed shared memory

Thinking parallel

- The following computes the sum of *x[0]+…+x[15]* serially:

```
For (i = 1 ; i < 16  ;  i++)
  {
    x[0] = x[0] + x[i]
  }
```

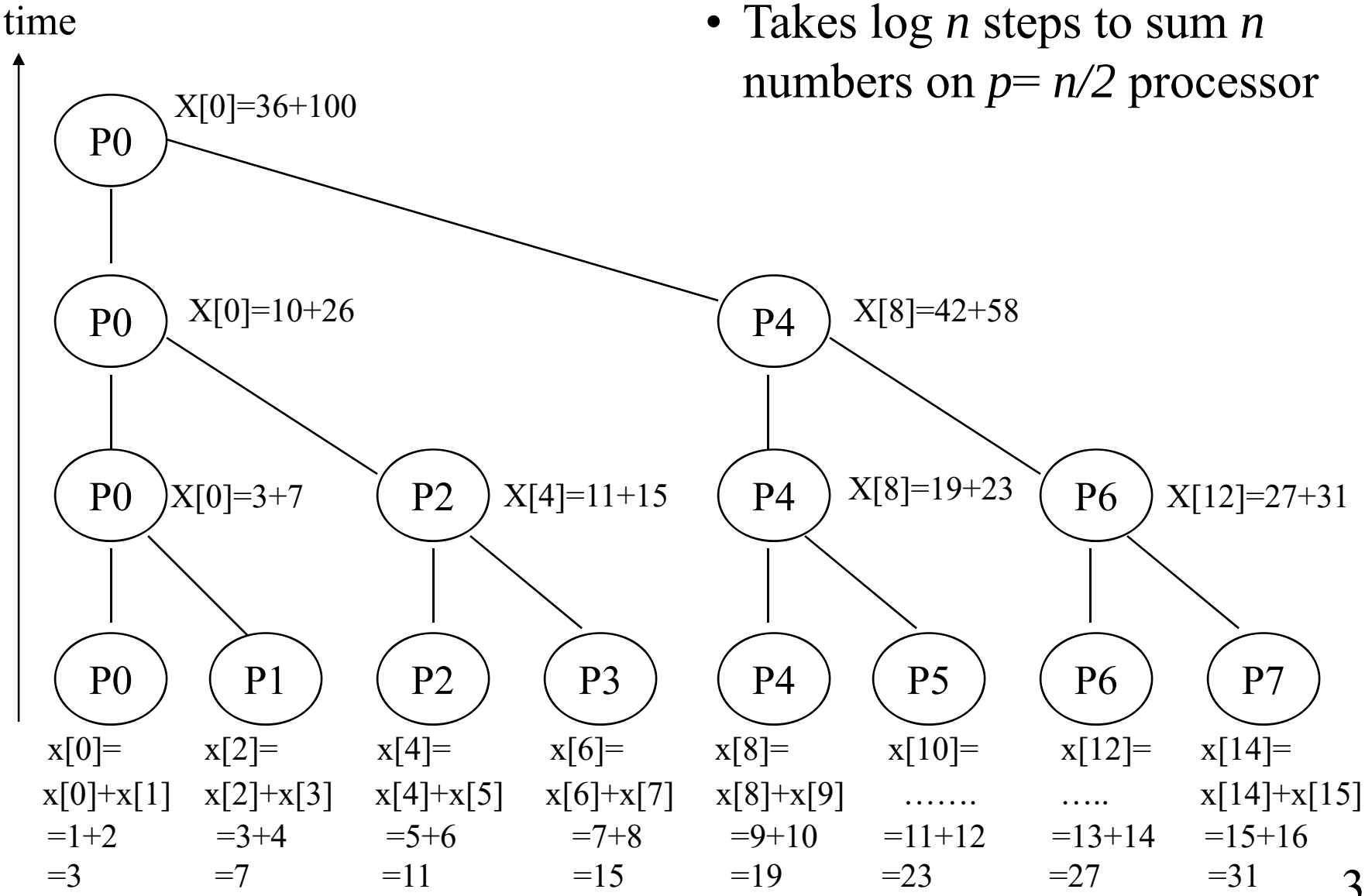
x[i] = i+1



- Takes *n-1* steps to sum *n* numbers on one processor

- Applies to associative and commutative operations (+, *, min, max, …)

Parallel sum algorithm (on 8 processors)



- Takes log *n* steps to sum *n* numbers on *p*= *n/2* processor

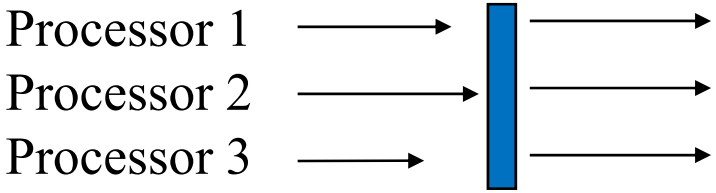
Example code on SMP

```
half = 8;  /* n=16 */
repeat {
    if (Pid < half) x[Pid] = x[Pid] + x[Pid+half];
    half = half/2;
} ;
until (half == 0);
```

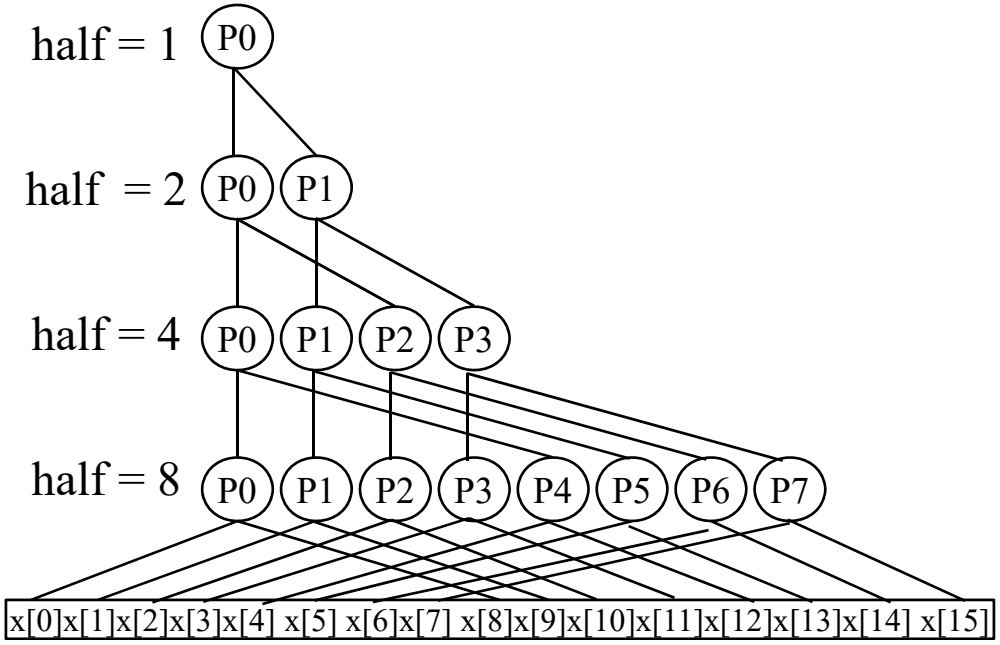
Pid is the processor ID

Should "half" be private or shared?

Potential for race conditions??



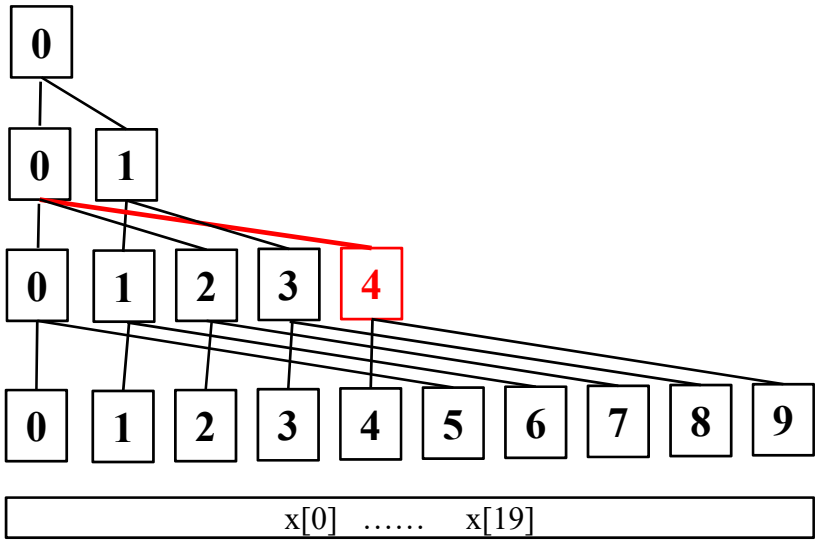
Barrier synchronization



Shared memory

Example: when *p = 10* (not a power of 2)

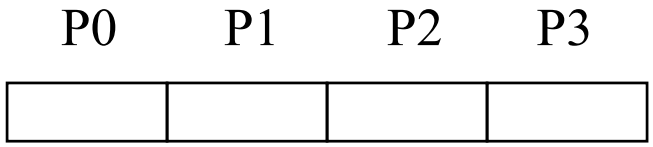
```
half = 10;  /* n=20 */
repeat
{
  if (half % 2 != 0 &&Pid == 0)  /*when half is odd; P0 gets the last element */
        x[0] = x[0] + x[half-1];
  if (Pid < half) x[Pid] = x[Pid] + x[Pid+half];
   half = half/2;
   barrier synch();
};
until (half == 0);
```



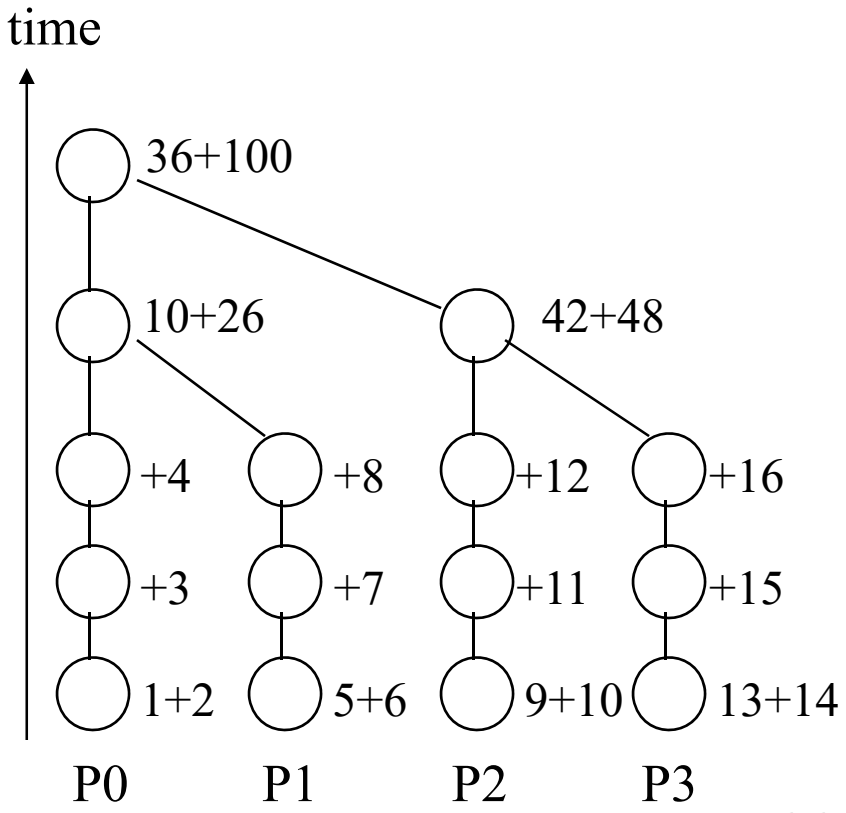
Now, we want to sum *n* elements on *p* processors, *n* >> *p*

Parallel sum of 16 elements on 4 processors

- Divide the array to be summed into 4 parts and assign one part to each processor



- Need 5 steps to sum *16* numbers on 4 processor
  - Speedup = 15/5 = 3
- Need 255+2 steps to sum 1024 numbers on 4 processors
  - Speedup = 1023/257 = 3.9
- How long does it take to sum n numbers on p processors?
  - Speedup = $\frac{n-1}{\frac{n}{p}-1+log\ p} \approx \frac{n}{\frac{n}{p}+log\ p}$



Parallel sum on a shared address space machine

- Assume *x[0] … x[9999]* are stored in shared memory.
- Assume *P* =16 processors, each with an identifier *Pid* (between 0 and 15)
- To sum the 10000 numbers, each processor executes *the following:*

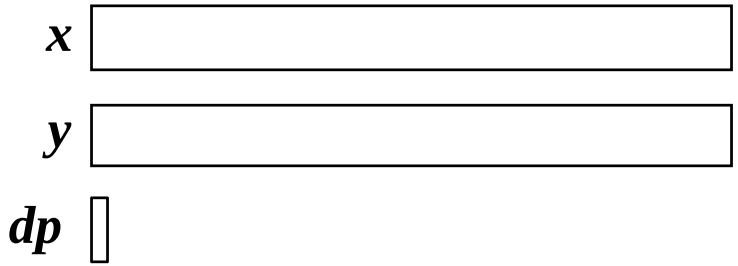
```
sum[Pid] = 0;
for ( i = 625 * Pid  ; i < 625  * (Pid +1)  ;  i++)
    sum[Pid]  = sum[Pid] + x[i];
half= 8 ;   /* P = 16 */
for (i=0 ; i < 4 ;  i++)
   { synchronize ;      /* a barrier */
     if(Pid < half ) sum[Pid] = sum[Pid] + sum[Pid + half ] ;
     half = half / 2; }
```

- *sum[ ]* and *x[ ]* are shared arrays,
- *half, Pid* and *i* are private variables (each processor has its own copy).
- Where will the global sum end up being?
- What if we want all processors to get a copy of the global sum?
- How would you change the program if *P* is not a power of two?
- Rewrite the program in terms of the # of processors and the size of *x*?

EX: Computing the dot product on shared memory

Example: dot product of two vectors, *x* and *y* (using a single thread)

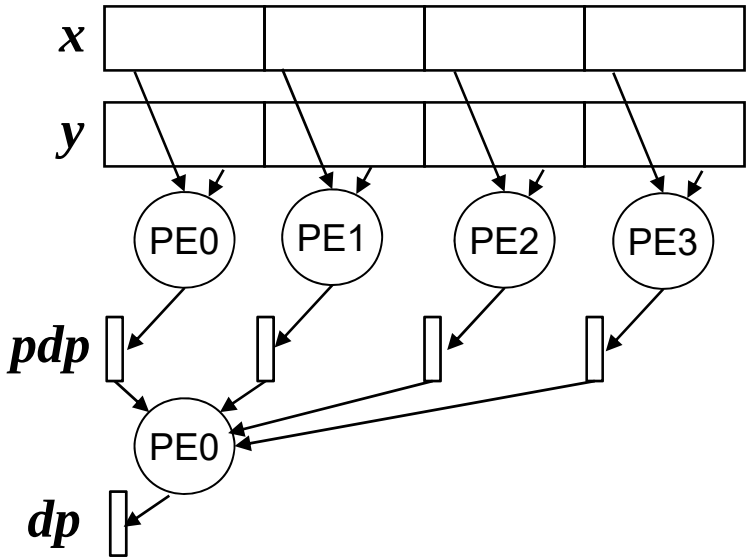
```
dp = 0 ;
for (i = 0 ;  i < n ;  i++)
    dp +=  x[i] * y[i]
```



Using 4 processors:

- Partition the arrays into 4 parts
- Each processor computes a partial sum
- One processor sums up the partial sums

  (could use binary tree reduction)

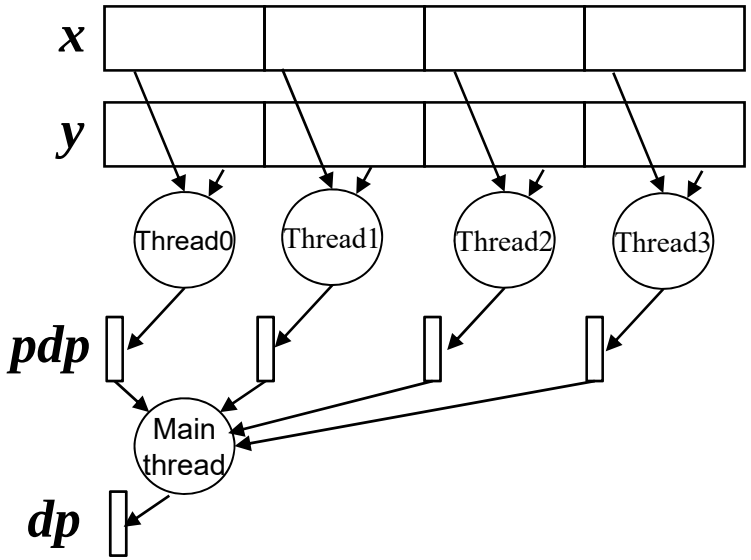


Multi-thread version of the dot product example

- Multi-threading was originally designed for Hiding Memory Latency
- With multicores, multiple threads will execute on multiple cores

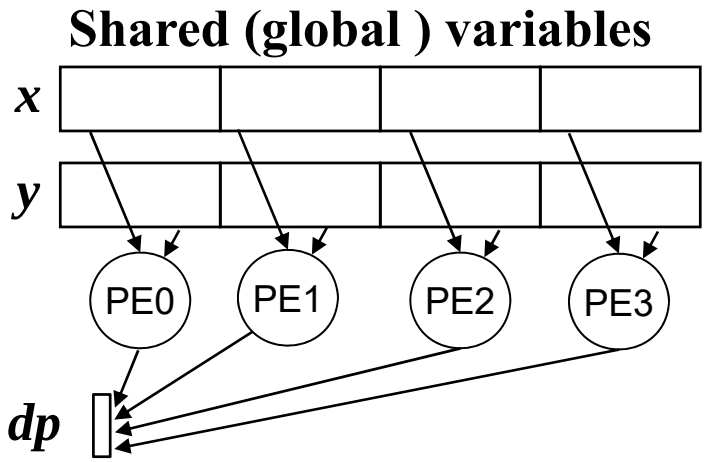
```
// x[] , y[], pdp[] and dp = 0 are all declared shared variables
for (k = 0; k < 4; k++)                    /* fork 4 threads */
    create_thread (partial_product, k, n);  /* k is used as a thread id */
Wait until all threads return ;             /* join threads */
for (k = 0; k < 4; k++)
    dp +=  pdp[k] ;


void partial_product (int k, int n);
{ int i ;          /* private variable */
  pdp[k] = 0 ;
  for (i = k*n/4;  i <  (k+1) * n/4 ;  i++)
      pdp[k] += x[i] * y[i] ;
  return ; }
```



Another version of the dot product example

```
// x[] , y[] and dp = 0 are all declared shared variables
for (k = 0; k < 4; k++)
    create_thread (partial_product, k , n);
Wait until all threads return ;
```




```
void partial_product (k, n);
{ int i, pdp = 0 ;  /* pdp is private -- each thread has its own copy */
  for (i = k*n/4;  i <  (k+1) * n/4 ;  i++)
        pdp += x[i] * y[i] ;
  pd += pdp ;
  return ;
}
```

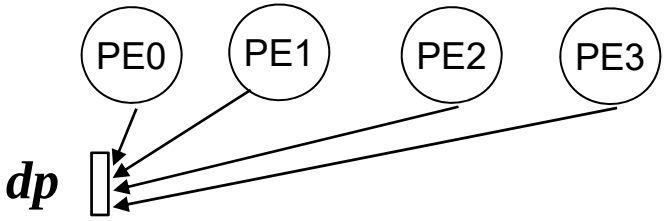
load *dp* from memory
Add *pdp* to *dp*
store *dp* to memory

Synchronization (race conditions)

What is the output of the following program??

```
dp = 0 ;
for (id = 0; id < 4; id++)
    create_thread (…, count , …);

void count ( );
{
   dp = dp + 1;
}
```



load *dp* from memory
Add *pdp* to *dp*
store *dp* to memory

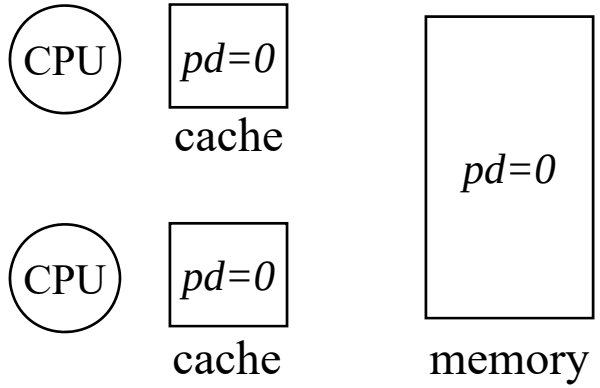
- A critical section is a section of code that can be executed by one processor at a time (to guarantee mutual exclusion)
- *locks* can be used to enforce mutual exclusion

```
get the lock ;
dp = dp + 1 ;
release the lock ;
```

Most parallel languages provide ways to declare and use locks and/or critical sections

Mutual Exclusion

- We need mutual exclusion in both parallel and serial programs (why?)
- Locks can be used to allow mutual exclusion, and hence provide a mechanism for exclusive access to shared data.
- Hardware support (in the form of atomic operations) is needed to implement locks
  - Atomic load-modify-store instructions,
  - Atomic swap instructions (swap the contents of a memory location with that of a register).
- In cache coherent systems, a cached memory location should be in the “Exclusive” state while executing an atomic operation on this location.



Implementing locks using atomic swap

- Atomic Swap interchanges a value in a register for a value in memory
  - loads the value from a memory location into the register
  - stores the value in register into the memory location

- Atomic swap can be used to implement locks:
  - The lock is represented by a variable, L
    - L=1 → locked
    - L=0 → not locked

```
Lock (L):
        Put 1 in Register, R
        Repeat
            Atomic Swap (R, L)
        Untill  (R = = 0)

Unlock:
        L = 0
```

Barrier synchronization

- A barrier synchronization between N threads can be implemented using a shared variable initialized to N.
- When a processor reaches the barrier, it decrements the shared variable by 1 and waits (in a busy wait loop) until the value of the variable is equal to zero before it leaves the barrier.

- Need locks???

- What if there is no shared variables (distributed memory machines)?

- Can you synchronize using special hardware?

The Pthread API

(see https://computing.llnl.gov/tutorials/pthreads/)

- Pthreads has emerged as the standard threads API (Application Programming Interface), supported by most vendors.
- The concepts discussed here are largely independent of the API and can be used for programming with other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well.
- Provides two basic functions for specifying concurrency:

```
#include <pthread.h>

int pthread_create (pthread_t  *thread_handle,
                    const pthread_attr_t  *attribute,
                    void (*thread_function)(void *),
                    void *arg);

int pthread_join (pthread_t  thread_handle,
                  void *ptr);
```

Mutual Exclusion

- Critical sections in Pthreads are implemented using mutex locks.
- Mutex-locks have two states: locked and unlocked. At any point of time, only one thread can lock a mutex lock. A lock is an atomic operation.
- A thread entering a critical section first tries to get a lock. It goes ahead when the lock is granted.

- **The API provides the following functions for handling mutex-locks:**

int *pthread_mutex_lock* (pthread_mutex_t *mutex_lock);

int *pthread_mutex_unlock* (pthread_mutex_t *mutex_lock);

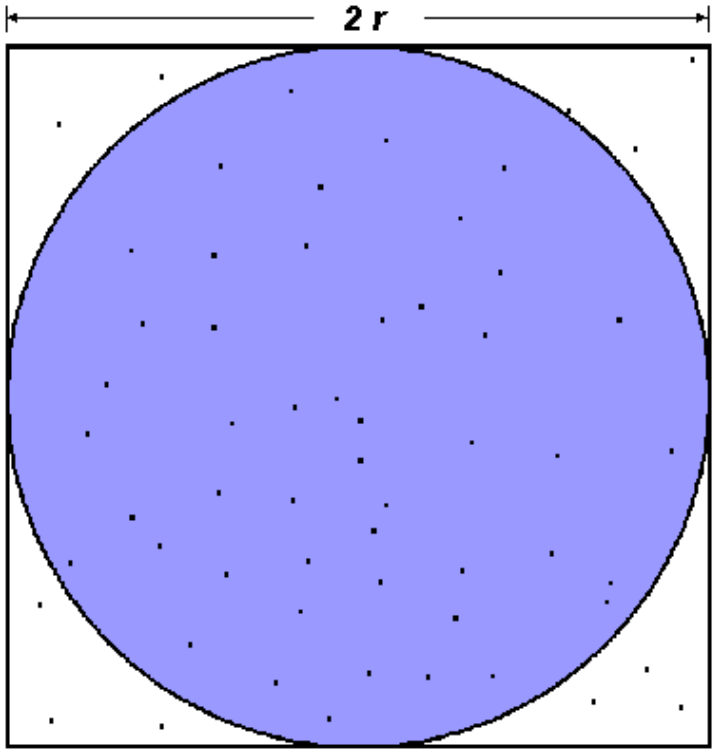
int *pthread_mutex_init* (pthread_mutex_t *mutex_lock,
const pthread_mutexattr_t *lock_attr);

Can replace by NULL

An Example (compute π)

The value of PI can be calculated in a number of ways. Consider the following method of approximating PI:

- Inscribe a circle in a square
- Randomly generate points in the square
- Determine the number of points in the square that are also in the circle
- Let A_c/A_s be the number of points in the circle divided by the number of points in the square
- PI ≈ 4 * (A_c/A_s)
- Note that the more points generated, the better the approximation



$$A_S = (2r)^2 = 4r^2$$

$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

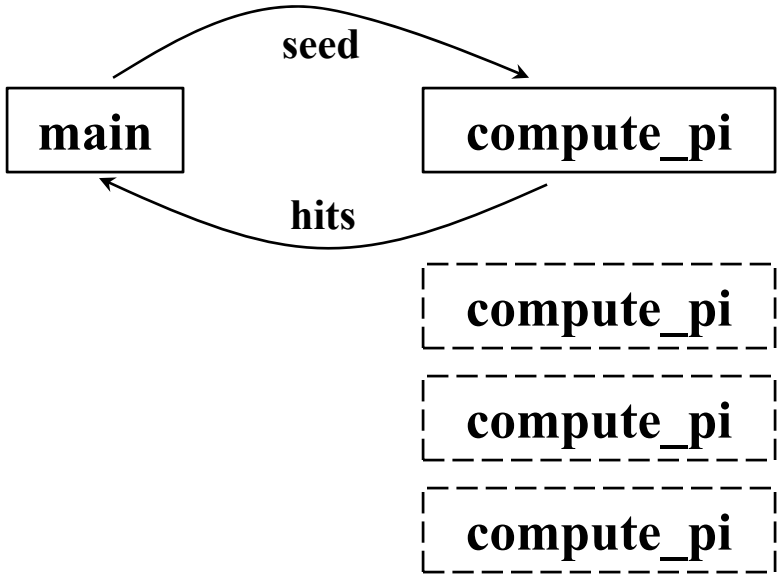
An Example (compute π)

```
#include <sys/time.h>
# define MAX_THREADS        64
void *compute_pi   ( void *);

int total_hits, sample_points, sample_points_per_thread, num_threads;

main ( )    {
…
}

void  *compute_pi (void *s) {
…
}
```



An Example (compute π)

```
struct arg_to_thread {int t_seed ; int hits ;}

main ( int argc, char argv[] )    {

    sample_points = atoi(argv[1]) ; /* first argument is the number of points */
    num_threads = atoi(argv[2]) ;   /* second argument is the number of threads*/

    pthread_t  p_threads[MAX_THREADS];
    pthread_attr_t  attr;
    pthread_attr_init (&attr);
    double computed_pi;
    struct arg_to_thread my_arg[MAX_THREADS] ;
```

An Example (compute π)

```
        total_hits =0;
        sample_points_per_thread = sample_points /num_threads;

        for (int i=0; i< num_threads; i++){
                my_arg[i].t_seed = i;          /* can chose any seed – here i is chosen*/
                pthread_create (&p_threads[i], &attr, compute_pi, &my_arg[i]);
        }

        for (i=0; i< num_threads; i++){
                pthread_join (p_threads[i], NULL);
                total_hits += my_arg[i].hits;
        }

        computed_pi = 4.0*(double) total_hits / ((double) (sample_points));
}
```

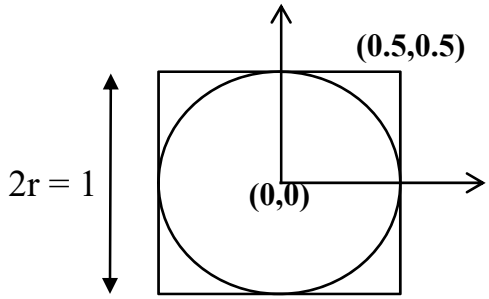
An Example (compute π)

```
void  *compute_pi (void *s) {
    struct arg_to_thread *local_arg ;
    int seed, i, local_hits ;
    double rand_no_x, rand_no_y;

    local_arg = s;
    seed=  (*local_arg).t_seed;
    local_hits =0;
    for (i=0 ;  i<sample_points_per_thread ;  i++) {
            rand_no_x = (double) (rand_r (&seed))/(double) RAND_MAX ;
            rand_no_y = (double) (rand_r (&seed))/(double) RAND_MAX ;
            if (((rand_no_x - 0.5) *(rand_no_x - 0.5) +
                    (rand_no_y - 0.5) * (rand_no_y - 0.5)) <0.25)
                    local_hits ++;          /* the generated sample is inside the circle*/
             seed *= i;
    }

    (*local_arg).hits = local_hits;
    pthread_exit (0);
}
```

Re-entrant function to generate a random number between 0 and RAND_MAX
Need to compile with
"gcc -D_REENTRANT –lpthread"



An Example (compute π)

```
void  *compute_pi (void *s) {
    struct arg_to_thread *local_arg ;
    int seed, i, local_hits ;
    double rand_no_x, rand_no_y;

    local_arg = s;
    seed=  (*local_arg).t_seed;
    local_hits =0;
    for (i=0 ;  i<sample_points_per_thread ;  i++) {
            rand_no_x = (double) (rand_r (&seed))/(double) RAND_MAX ;
            rand_no_y = (double) (rand_r (&seed))/(double) RAND_MAX ;
            if (((rand_no_x - 0.5) *(rand_no_x - 0.5) +
                    (rand_no_y - 0.5) * (rand_no_y - 0.5)) <0.25)
                    local_hits ++;          /* the generated sample is inside the circle*/
             seed *= i;
    }

    (*local_arg).hits = local_hits;
    pthread_exit (0);
}
```

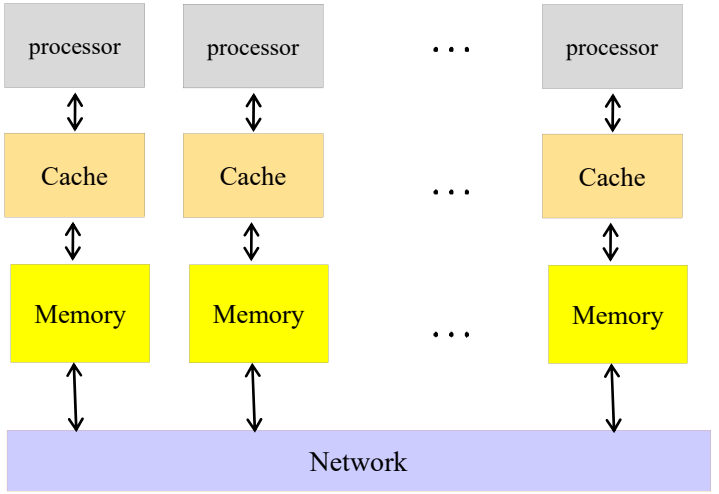
Re-entrent function to generate a random number between 0 and RAND_MAX
Need to compile with
"gcc -D_REENTRANT –lpthread"

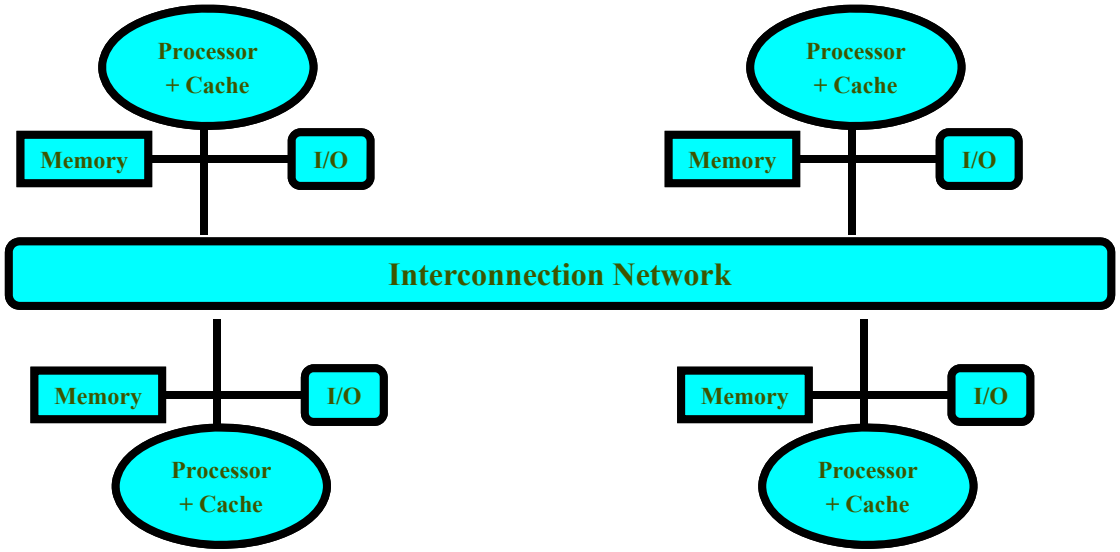
int *pthread_mutex_lock* (pthread_mutex_t *m_lock);
total_hits =+ local_hits;
int *pthread_mutex_ulock* (pthread_mutex_t *m_lock);

Allows the removal of "total_hits += my_arg[i].hits;" from main()

Multiprocessors connected by networks (Section 6.7)

- 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 (ex:Ethernet or a switch)
- Suitable for applications with independent tasks
  - Web servers, databases, simulations, …
- High availability, scalable, affordable
- Problem: Low interconnect bandwidth (compared to SMP)

- Grid Computing
  - computers interconnected by long-haul networks (ex: Internet)
  - Work units farmed out, results sent back
  - Can make use of idle time on PCs (ex: PITTGRID)

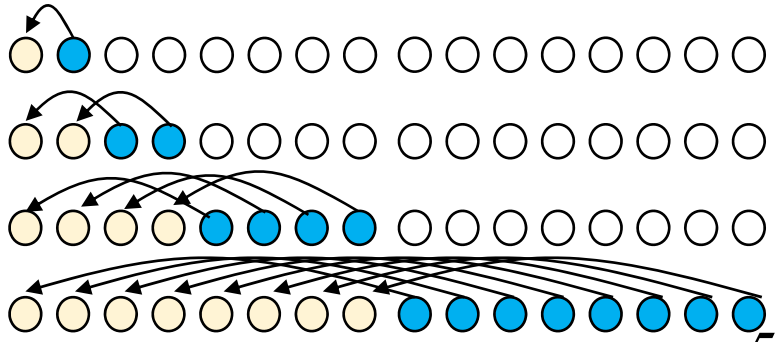
Programming a distributed address space machine

- Assume that 10000 values are stored in the local memories of 16 processors such that 625 values are stored in *x[0] … x[624]* in the local memory of each processor.
- All variables are local variables (each processor has its own copy) – no shared variables.
- The function "send(*m,p*)" sends a message containing the value of *m* to processor *p*.
- The function "receive(*m*)" receives a message and puts the received value in *m.*

```
sum = 0;
for (i=0 ; i < 625 ;  i++)
     sum = sum+ x[i];
half = 8;    /* P = 16  */
for (i=0 ; i < 4 ;  i++)
   { if (2*half > Pid >= half )  send (sum, Pid – half );
     if (Pid < half )  {  receive (remote_sum);
                          sum =+ remote_sum ; }
     half = half / 2; }.
```

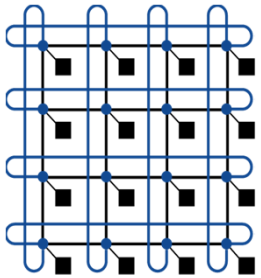
Compare with the shared memory program on slide 34.

- No shared variables.
- Where is the global sum?
- The distribution of the initial data to the local memories is done either by the programmer or by the compiler.

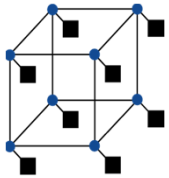


Interconnection network (Section 6.8)

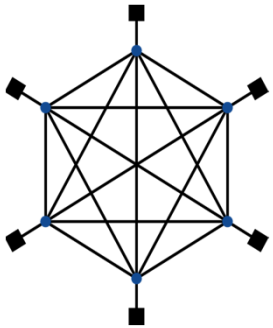
- To connect processors to memories or processors to processors



2D Mesh



N-cube (N = 3)

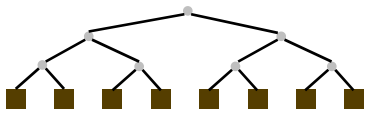


Fully connected

Bus

Ring

- Issues
  - Latency
  - Bandwidth
  - Cost (wires, switches, ports, …)
  - Scalability



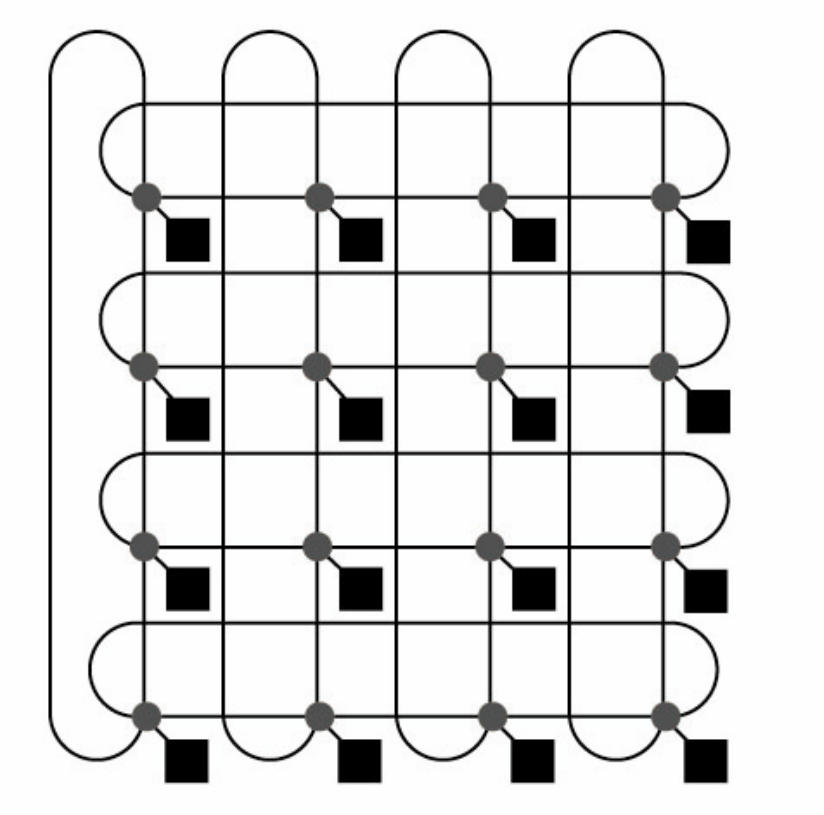
Tree

- Topology has been a focus of architects

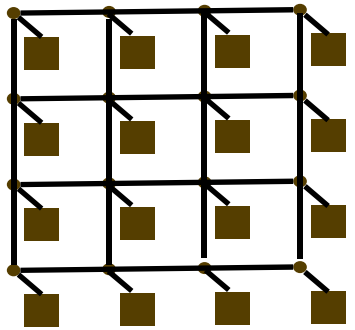
Evaluating Interconnection Network topologies

- *Diameter:* The distance between the farthest two nodes in the network.
- *Average distance:* The average distance between any two nodes in the network.
- *Node degree:* The number of neighbors connected to any particular node.
- *Bisection Width:* The minimum number of wires you must cut to divide the network into two equal parts.
- *Cost:* The number of links or switches (whichever is asymptotically higher) is a meaningful measure of the cost. However, a number of other factors, such as the ability to layout the network, the length of wires, etc., also factor in to the cost.

2-D torus



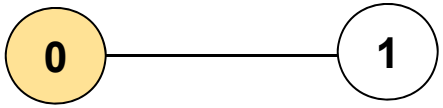
- Diameter??
- Bisection bandwidth??
- Routing algorithms
  - x-y routing
  - Adaptive routing
- 2D mesh (without the wrap-around connections)



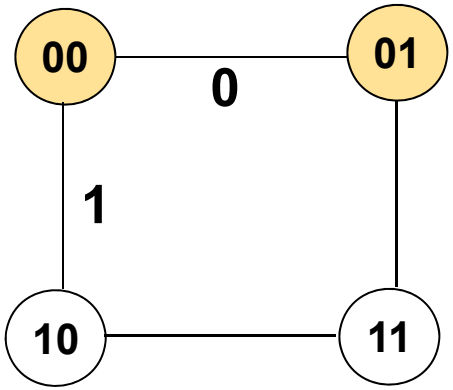
- **Variants**
  - 1-D (ring), 3-D.

Hypercube interconnections

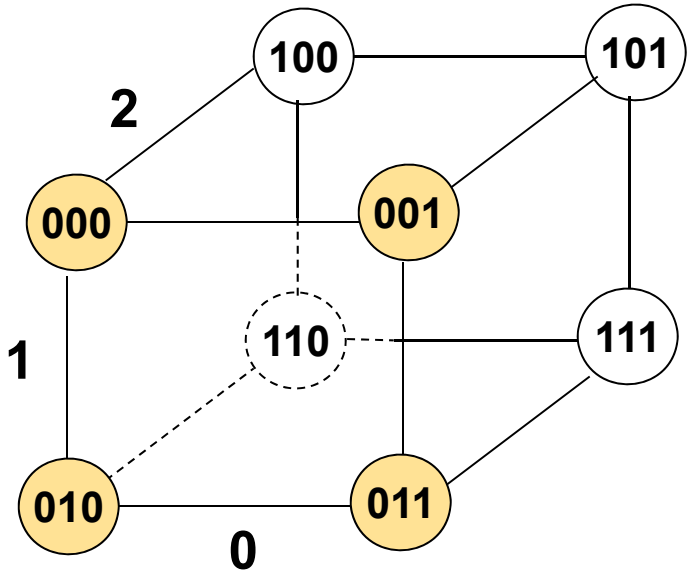
- An interconnection with low diameter and large bisection width.
- A q-dimensional hypercube is built from two (q-1)-dimensional hypercubes.



1-dimension
binary hypercube

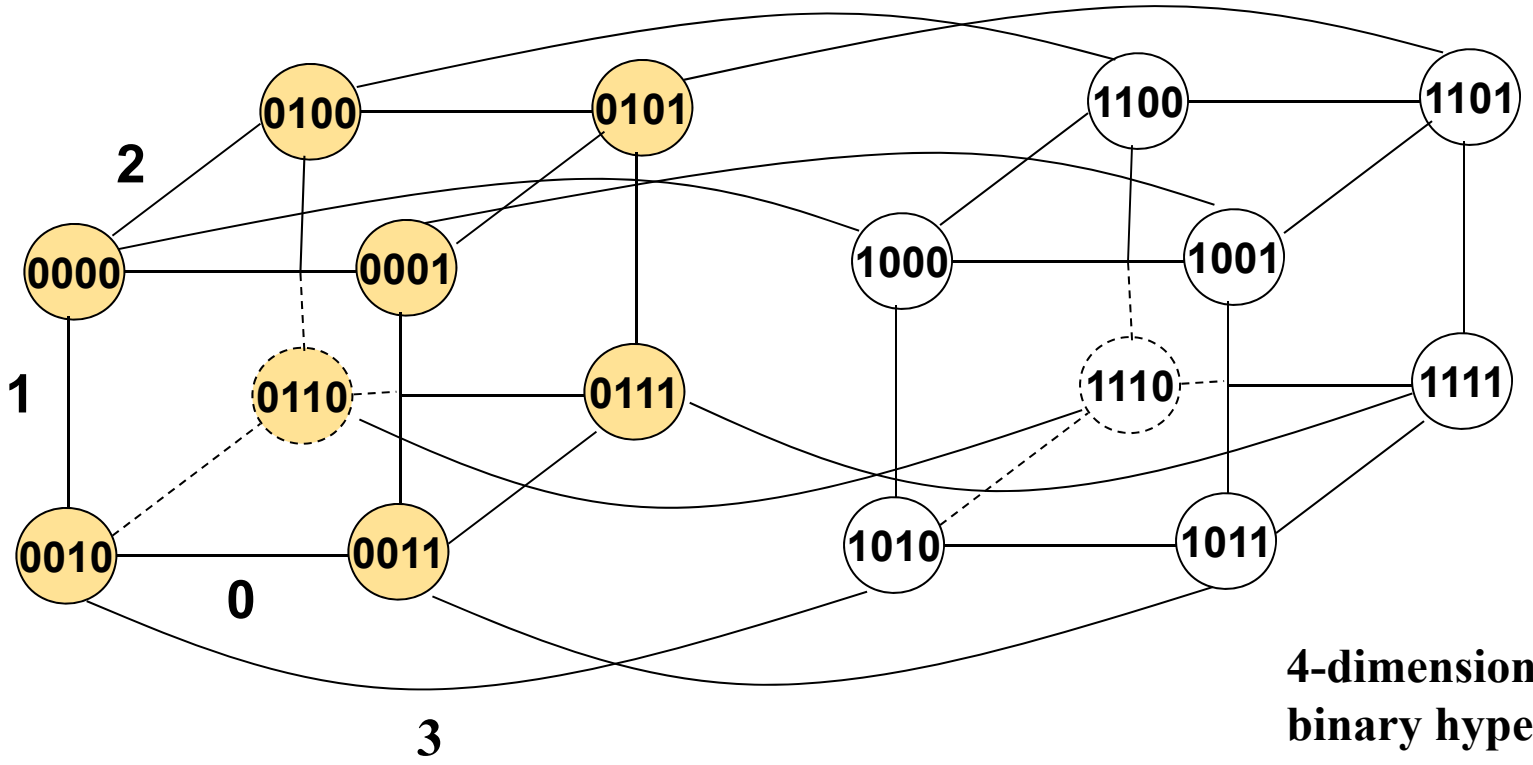


2-dimension
binary hypercube



3-dimension
binary hypercube

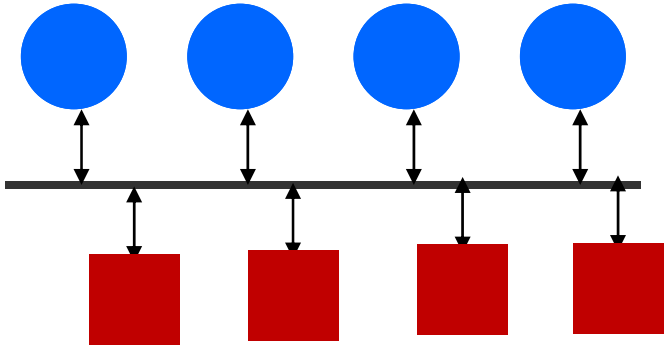
A 4-dimension Hypercube (16 nodes)

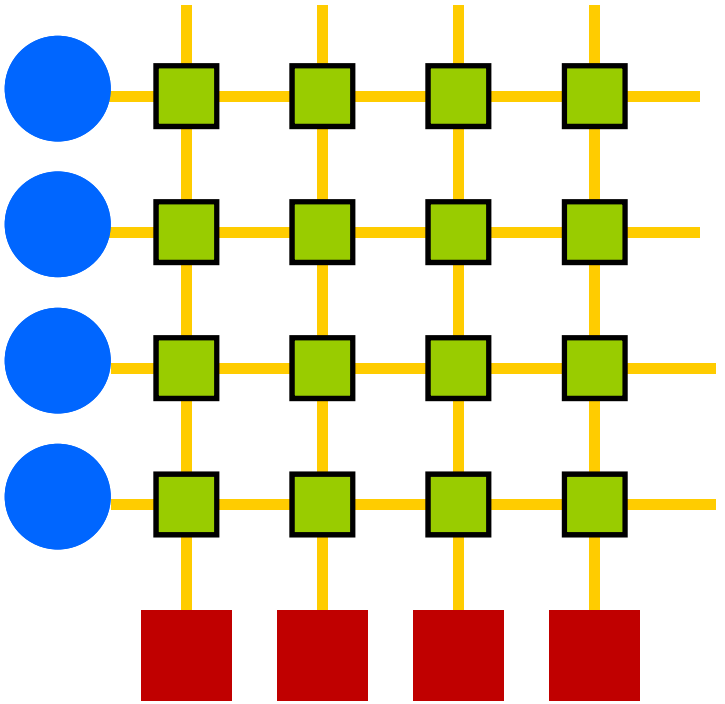


4-dimension binary hypercube

- Can recursively build a *q*-dimension network – has 2^q nodes

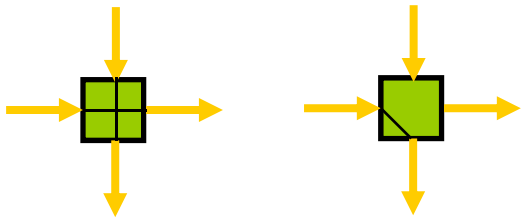
Centralized switching: Buses and crossbars



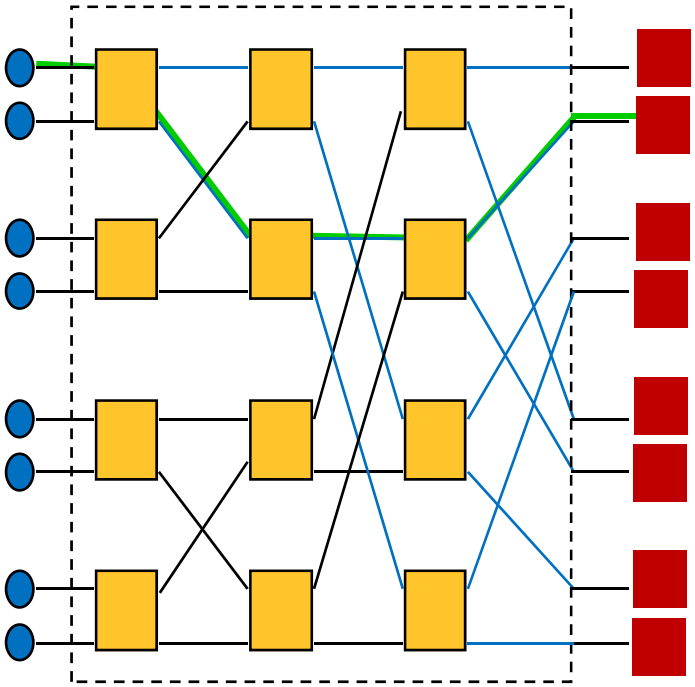


- Cost
- Latency
- Bandwidth
- Scalability

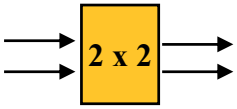
Each switch is a 2x2 switch that can be set to one of 2 settings

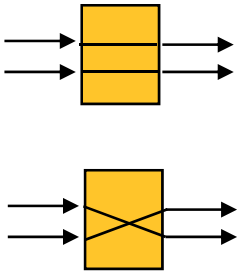


Centralized switching: Multistage networks



A 2x2 switch or router



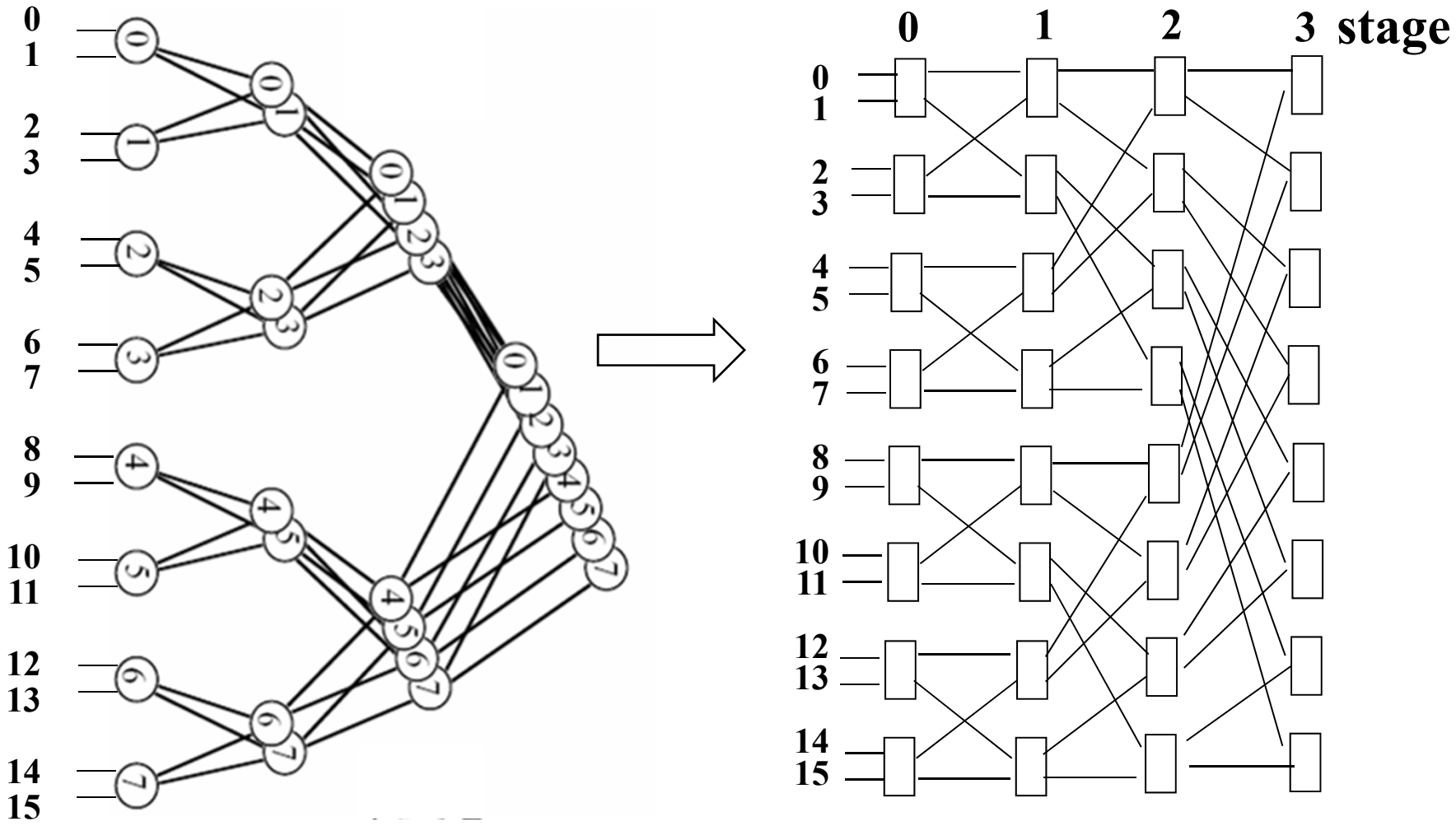


Circuit switching: circuits are established between inputs and outputs – arbitrate entire circuits.

Packet switching: packets are buffered at intermediate switches – arbitrate individual switches.

- **NxN Omega network:** log N stages, with N/2, 2x2 switches.
- **A blocking network**: some input-output permutations cannot be realized due to path conflicts.

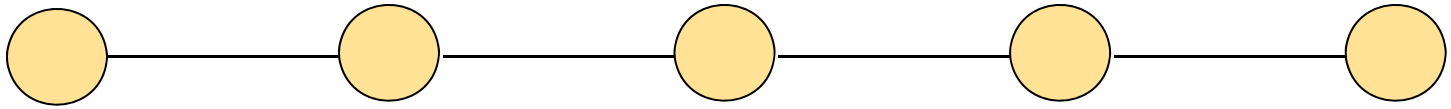
Fat tree networks



A fat tree networks using 2x2 bidirectional switches

Message latencies

- Ignoring congestion (queuing delays), the total time to transfer a message comprises of :
  - *Startup time* (t_s): Time overhead spent at sending and receiving nodes.
  - *Per-hop time* (t_h): a function of number of hops and includes factors such as switch and link latencies, network delays, etc.
  - *Transfer time* (t_w): This time depends on the bandwidth of links. For example, it takes t_w = 1 n.sec to send 1 bit on a 1 Gbits/sec link.



Hence, the time for a message of m bits to traverse one hop is $m\ t_w + t_h$

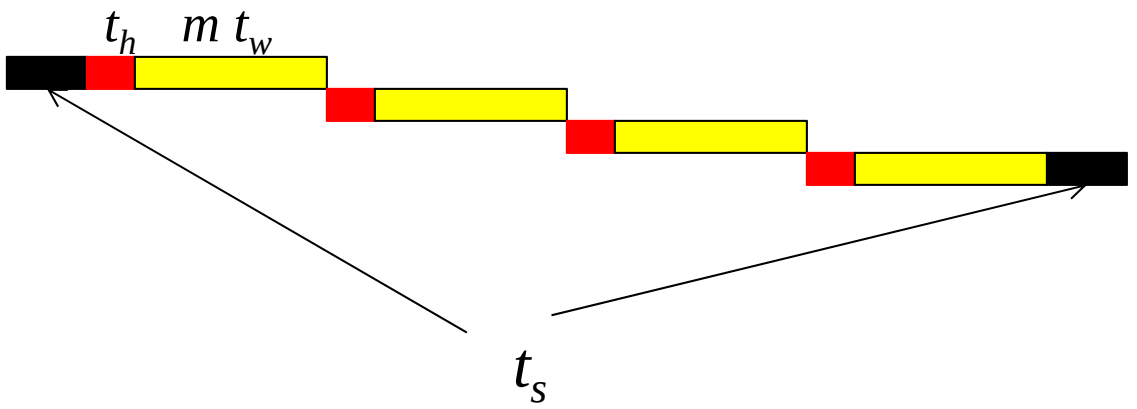
Example: if t_h = 500 n.sec., then a message with 1000 bits takes 1500 n.sec. to traverse a 1Gb/sec link.

Store-and-Forward switching

- A message traversing multiple hops is completely received at an intermediate hop before being forwarded to the next hop.
- The total communication cost for a message of size *m* to traverse *h* communication links is

$$t_{comm} = t_s + (mt_w + t_h)h$$

Example: *h*=4



Example: if t_h =500 n.sec and t_s = 400, then it takes 400+1500 * 4 = 6400 n.sec. for a message of 1000 bits to travel from a source to a destination that is 4 hops away on 1Gb/sec links.

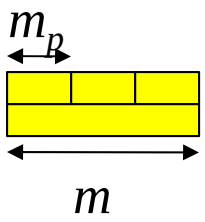
Packet switching

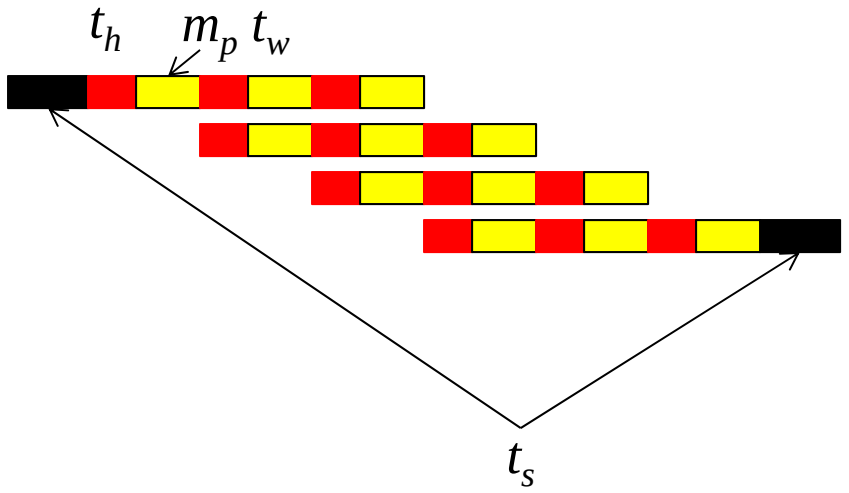
- Store-and-forward makes poor use of communication resources.
- Packet routing breaks messages into packets (say k packets of length $m_p = m/k$ each) and pipelines them through the network.
- Since packets may take different paths, each packet must carry routing information, error checking, sequencing, and other header information.
- The total communication time for packet routing is approximated by:

$$t_{comm} = t_s + (m_p t_w + t_h)h + (m_p t_w + t_h)(k-1)$$

Example:

$h=4$, $k=3$





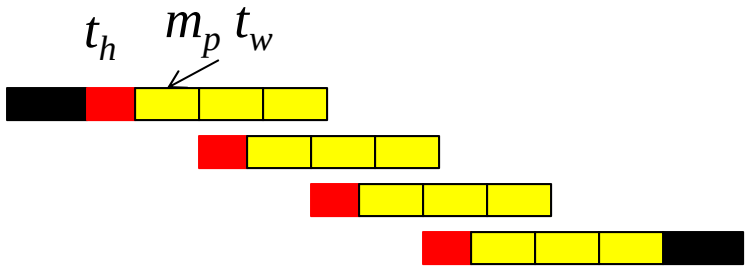
Cut-Through (worm-hole) switching

- Takes the concept of packet routing to an extreme by further dividing messages into basic units called flits.
- To minimize the header, force all flits to follow the same path.
- The first flit programs all intermediate routers. Subsequent flits then take the same route (minimal headers may be needed).
- Ignoring the headers in subsequent flits, the total communication time is approximated by

$$t_{comm} = t_s + (m_p t_w + t_h)h + m_p t_w(k-1)$$

$$= t_s + m\, t_w + h\, t_h + m_p t_w(h-1)$$

$$\approx t_s + m\, t_w + h\, t_h \qquad if\ m_p << m$$



Graphic Processing Units – GPU (Section 6.6)

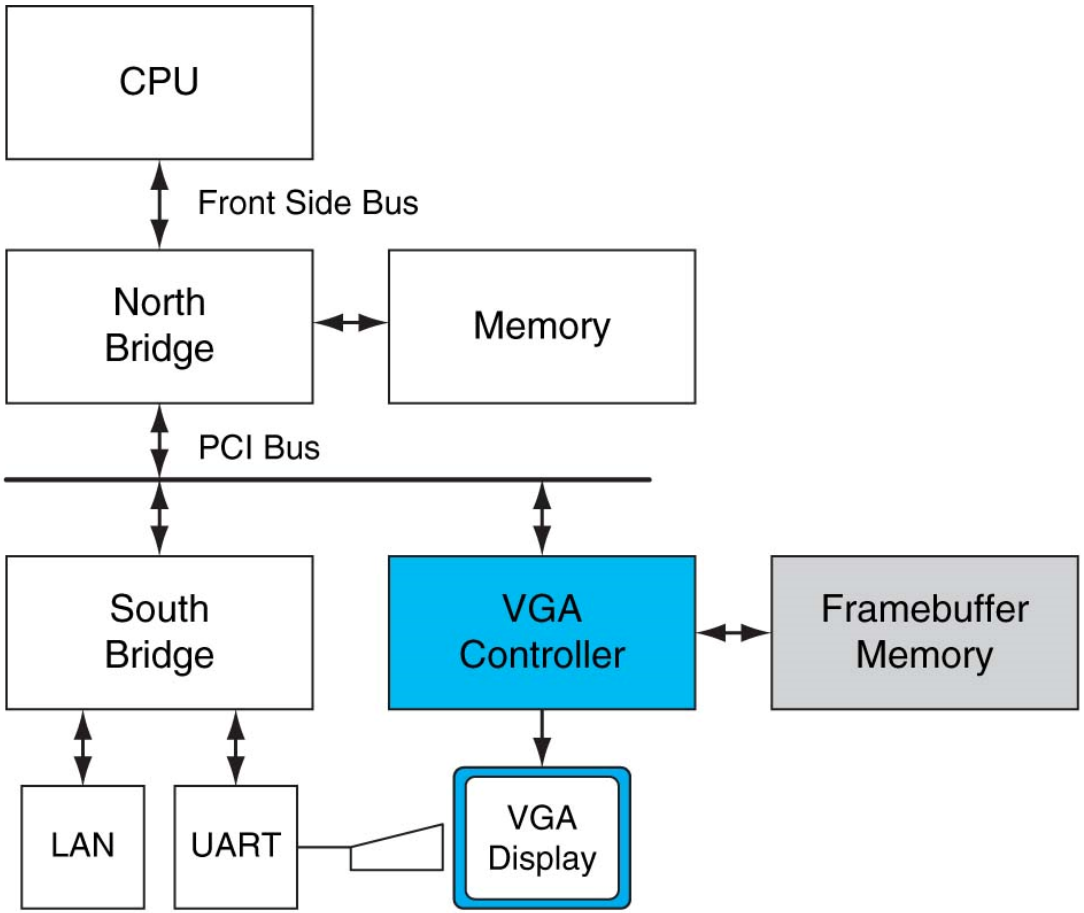
History of GPUs

- VGA (Video graphic array) in early 90's -- A memory controller and display generator connected to some (video) RAM
- By 1997, VGA controllers were incorporating some acceleration functions
- In 2000, a single chip graphics processor incorporated almost every detail of the traditional high-end workstation graphics pipeline
  - Processors oriented to 3D graphics tasks
  - Vertex/pixel processing, shading, texture mapping, rasterization

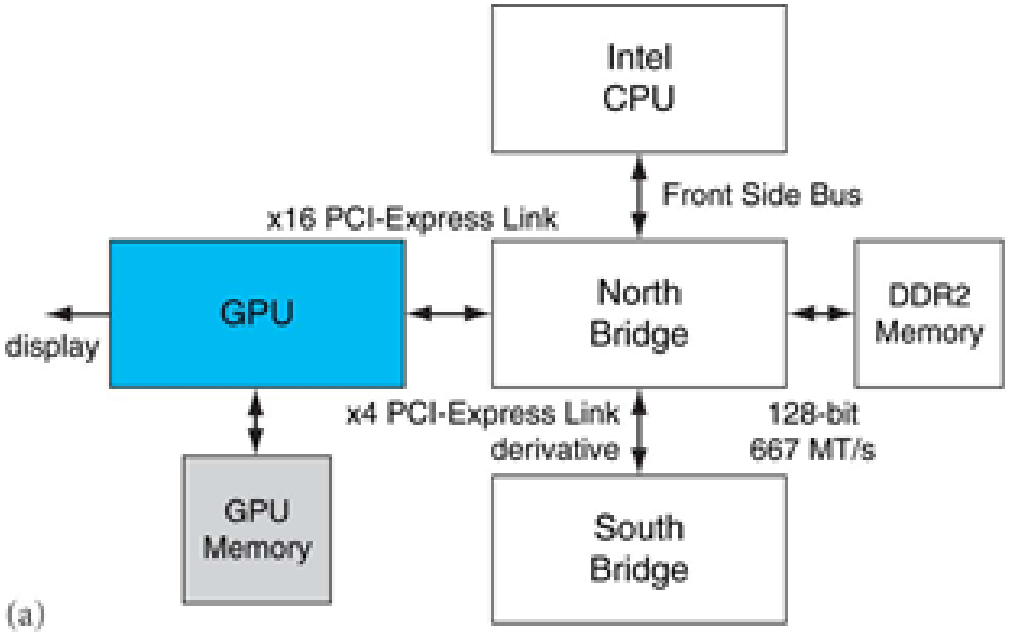
- More recently, processor instructions and memory hardware were added to support general-purpose programming languages

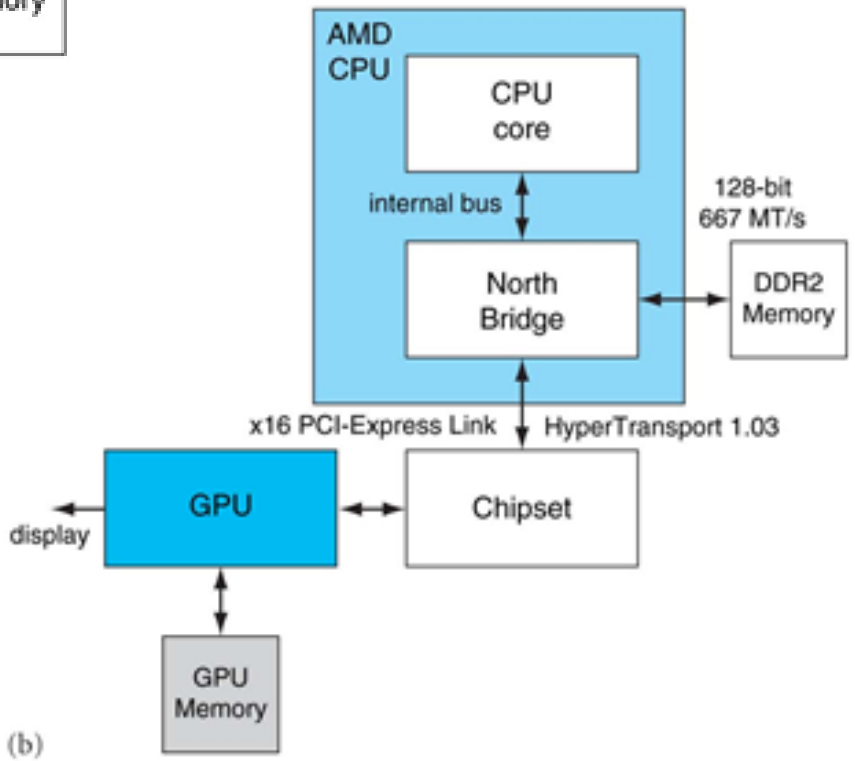
- OpenGL: A standard specification defining an API for writing applications that produce 2D and 3D computer graphics
- CUDA (compute unified device architecture): A scalable parallel programming model and language for GPUs based on C/C++

Historical PC architecture

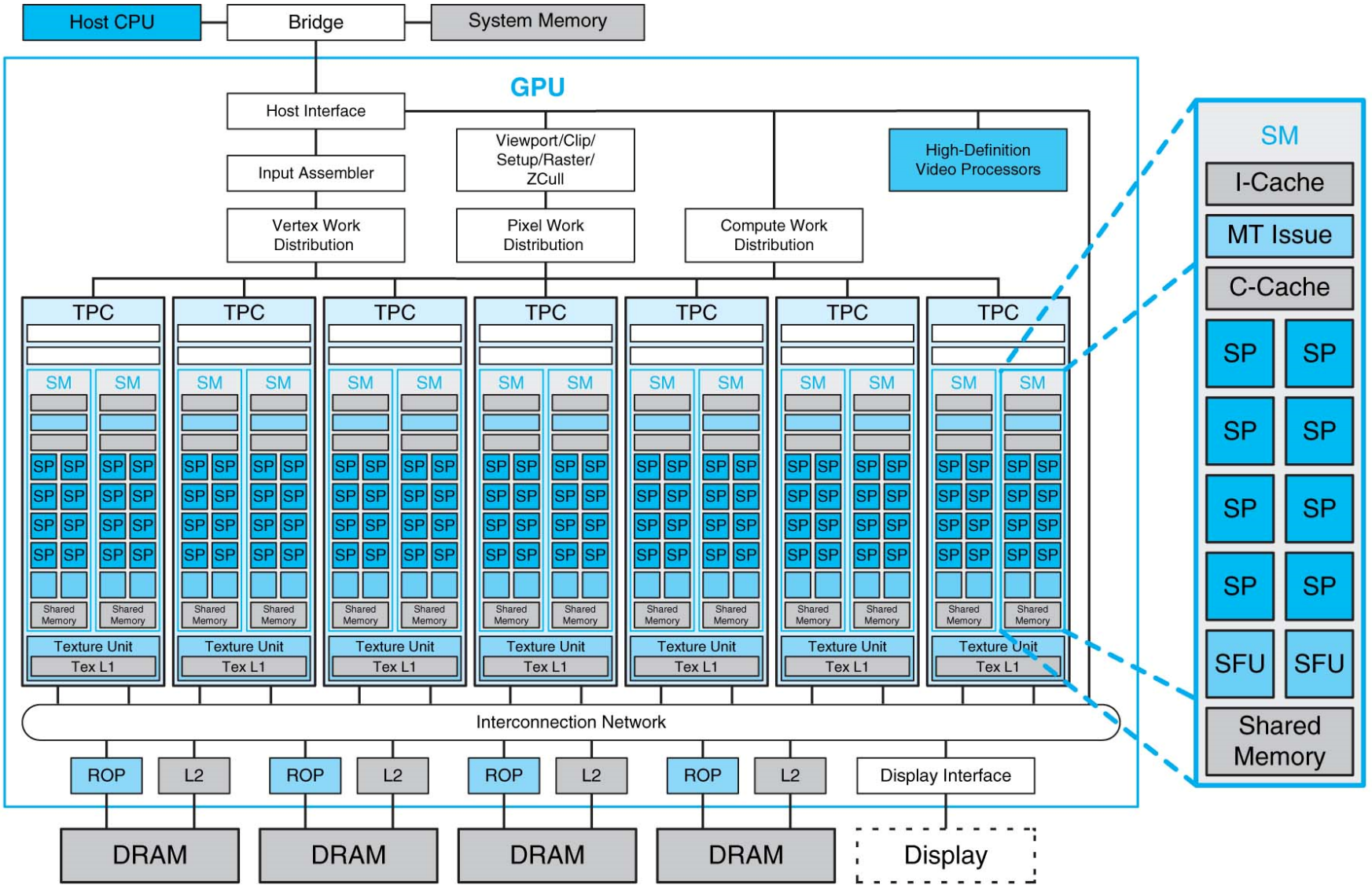


Contemporary PC architecture





Basic unified GPU architecture



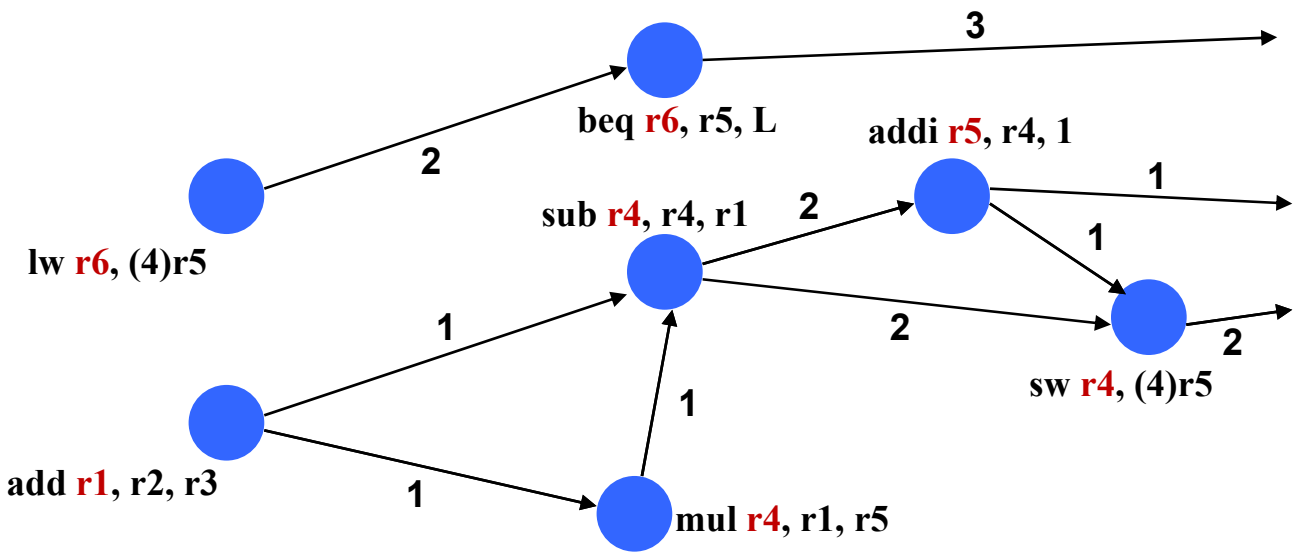
ROP = raster operations pipeline

Data dependence graphs

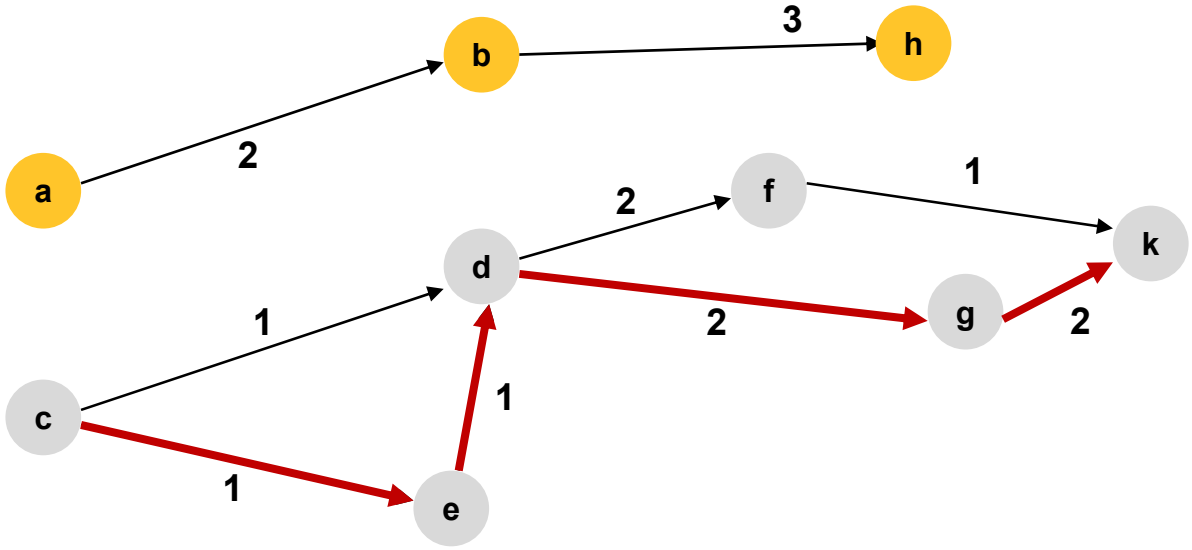
- Instructions consume values (operands) created by previous instructions

```
add r1, r2, r3
mul r4, r1, r5
```

- Given a sequence of instructions to execute, form a directed graph using producer-consumer relationships (not instruction order in the program)
  - Nodes are instructions
  - Edges represent dependences, possibly labeled with related information such as latency, etc.



Scheduling a thread with data dependence

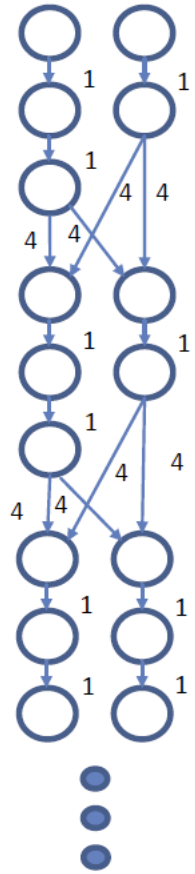


- Assuming two functional units.

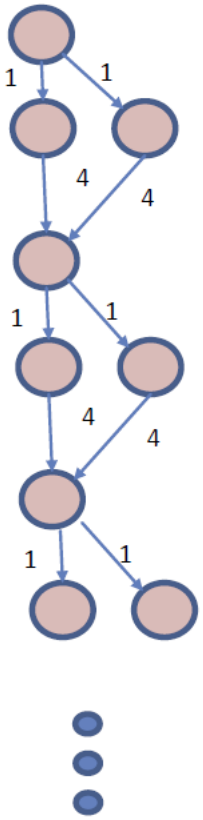
cycle	1	2	3	4	5	6	7	8	9
Unit 1	a		b		f	h			
Unit 2	c	e	d		g		k		

- What is the minimum execution time, given unlimited resources?

Homework: scheduling multiple threads



Thread A



Thread B

A	A
A	A
A	
B	
B	B
A	A
A	A
A	
B	
B	B
A	A
A	A
A	

Coarse grain MT

A	A
B	
A	A
B	B
A	
B	
A	A
B	B
A	A
A	
B	
B	B

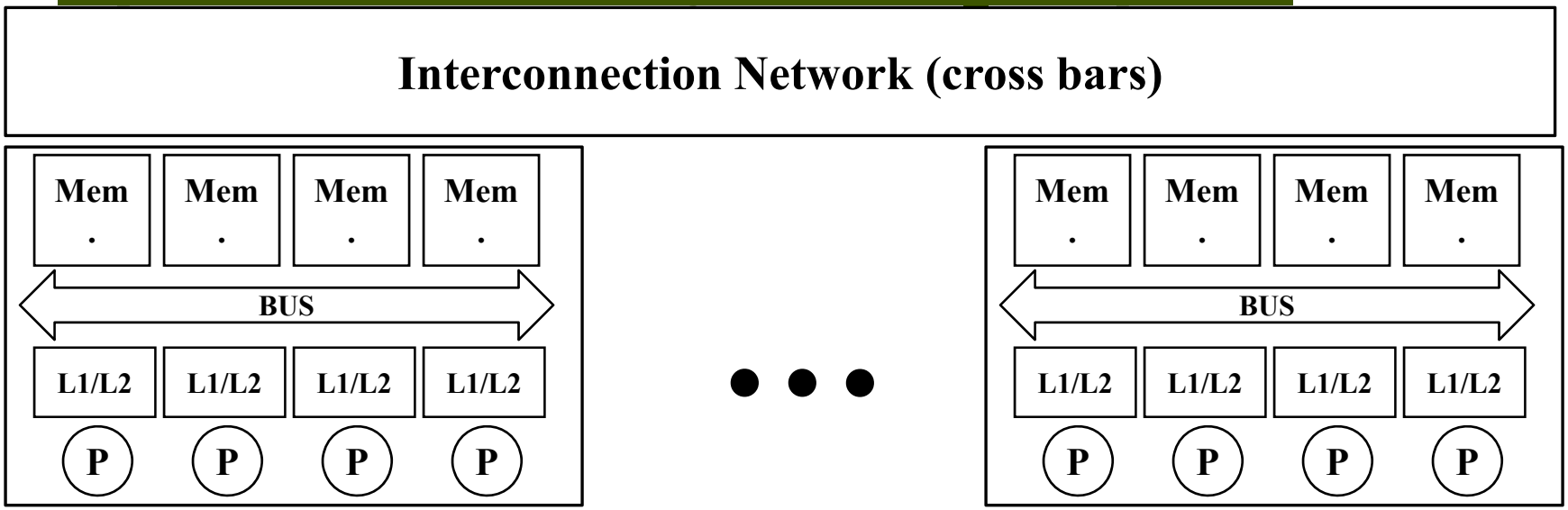
Fine grain MT

A	A
A	A
A	B
B	B
A	A
A	A
A	B
B	B
A	A
A	A
A	B
B	B

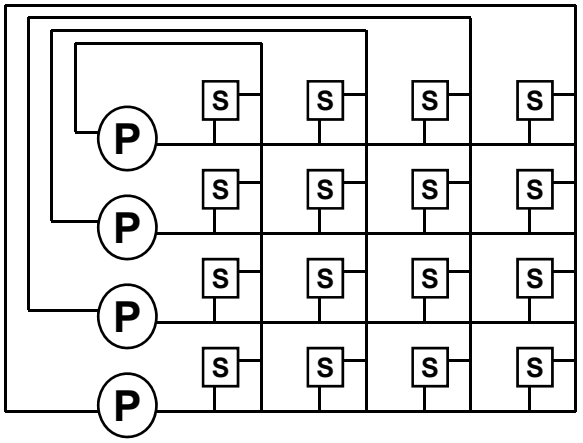
SMT

The Sun Fire E25 K (SMP)

http://www.sun.com/servers/highend/sunfire_e25k/specs.xml



- Board = 4 SPARCS IV + 64 GB memory
- Up to 18 boards connected by crossbars
- 1.15 TB of Distributed shared memory
- Snoopy memory coherence on each board
- Directory-based coherence among boards

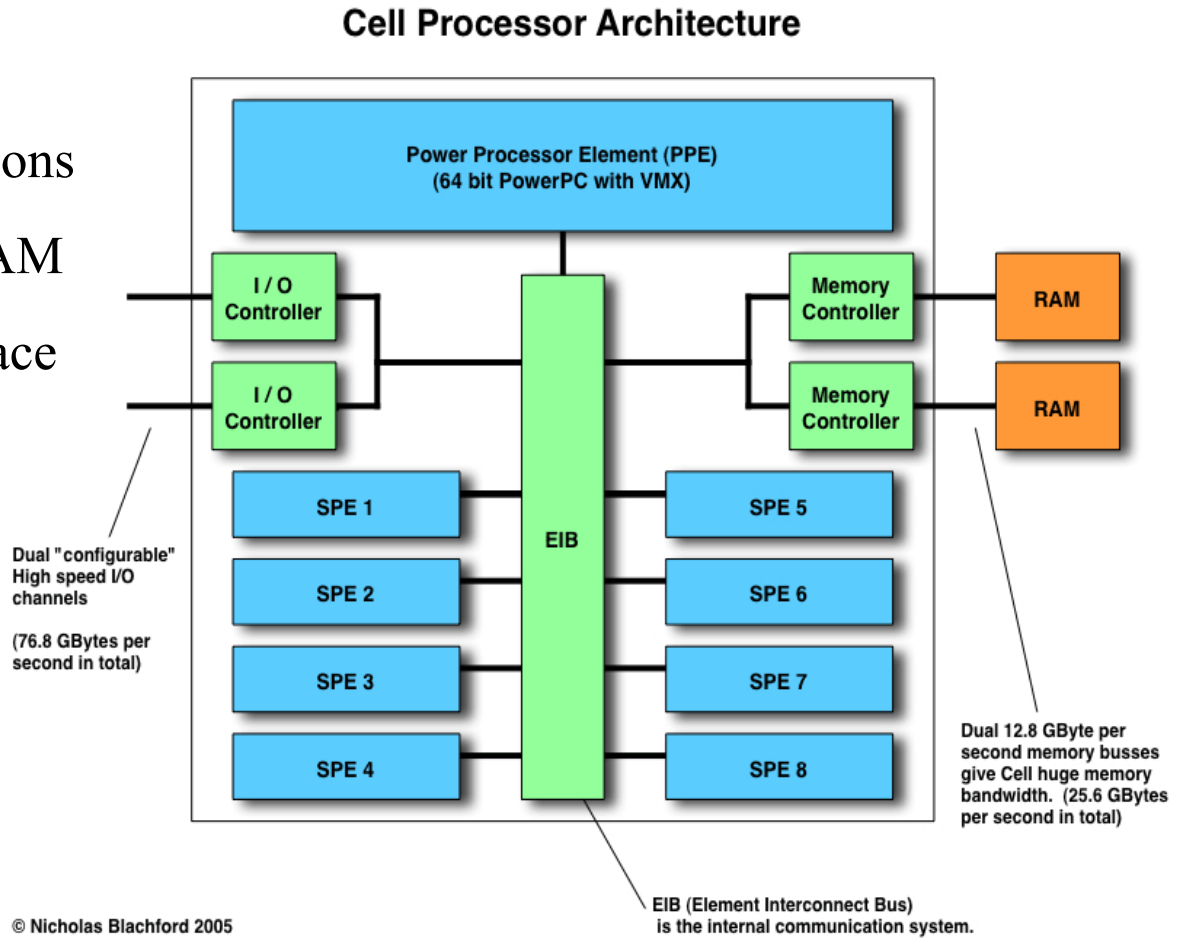


A 4x4 crossbar

The cell (heterogeneous chip design)

http://www.blachford.info/computer/Cell/Cell1_v2.html

- Targets video games
- SPE supports vector operations
- Each SPE has 256KB of RAM
- SPE has its own address space
- DMA transfers between memories



P threads (POSIX)

- A thread is a light weight process (has its own stack and execution state, but shares the address space with its parent).
- Hence, threads have local data but also can share global data.