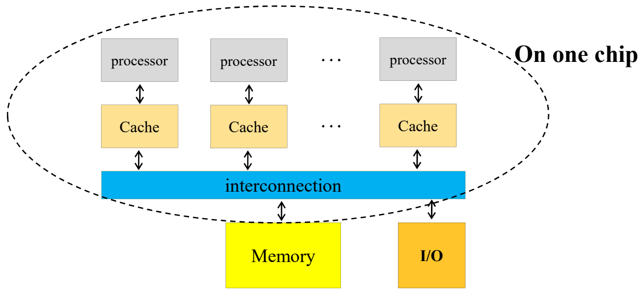
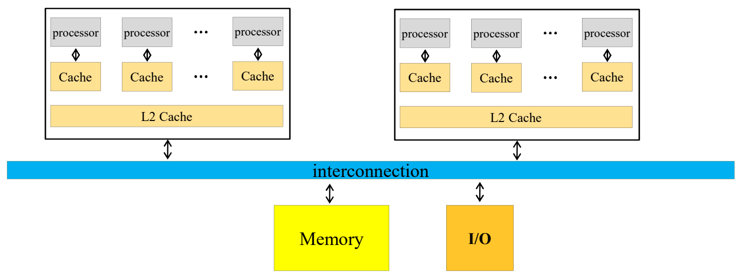
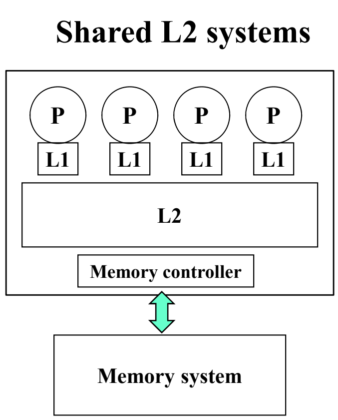
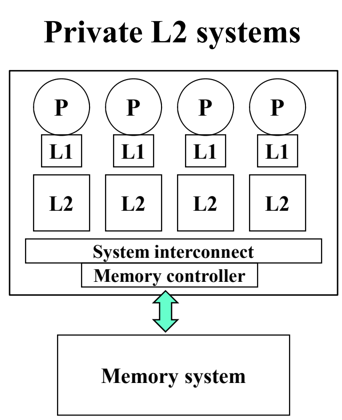
Shared memory systems (CMP, multicores, manycores) (sec. 6.5)





Chip Multiprocessors

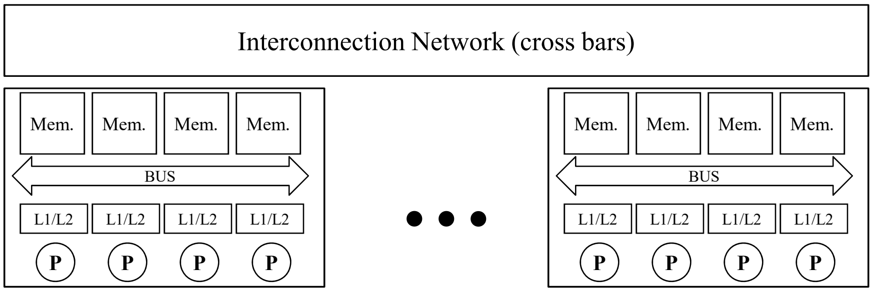




- Examples: Intel Pentium
- Examples: AMD Opteron

Example: The Sun Fire E25 K

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

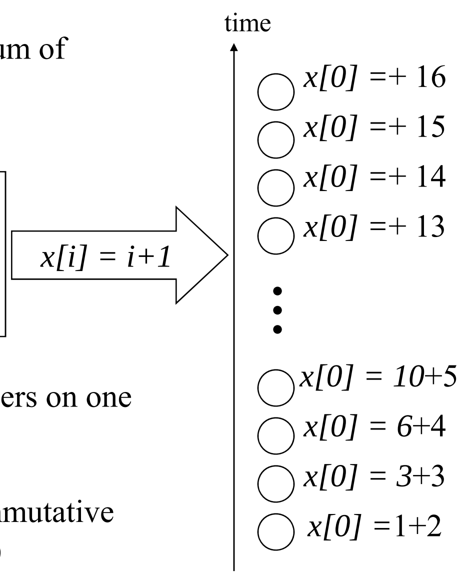


- Board = 4 SPARCS IV + 64 GB memory
- Up to 18 boards connected by crossbars
- 1.15 TB of Distributed shared memory

Thinking parallel

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

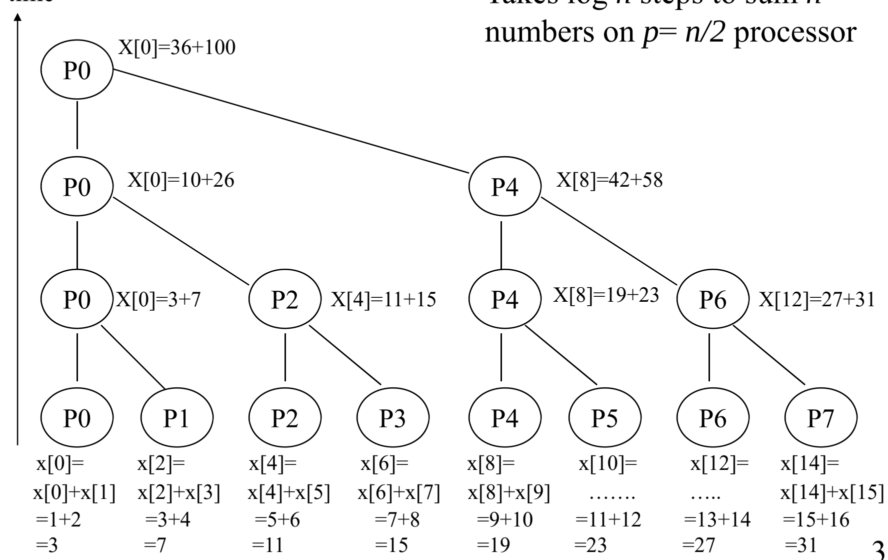
```
For (i = 1 ; i < 16  ;  i++)
  {
     x[0] = x[0] + x[i]
  }
```



- 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



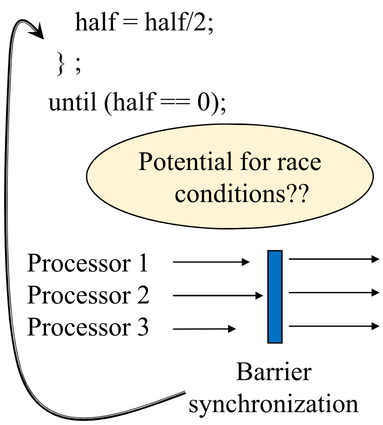
Example code on SMP

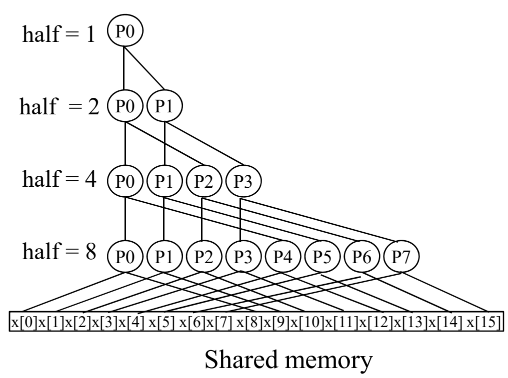
```
half = 8;  /* n=16 */
repeat {
   if (Pid < half) x[Pid] = x[Pid] + x[Pid+half];
   half = half/2;
 } ;
until (half == 0);
```

Pid is the processor ID

Should "half" be private or shared?

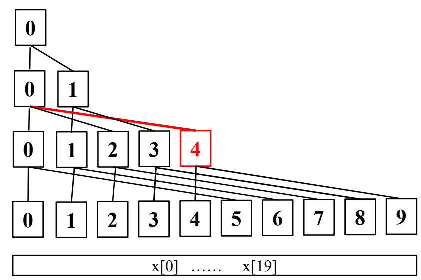
Potential for race conditions??





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

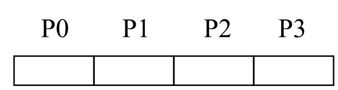
```
n=20 ; half = n / 2 ;
repeat
{
   if (Pid < half) x[Pid] = x[Pid] + x[Pid+half];
   if (n % 2 != 0 && Pid == 0)  /*when n is odd; P0 gets the last element */
        x[0] = x[0] + x[n-1];
   n = half ;
   half = half / 2;
   barrier synch();
};
until (half == 0);
```

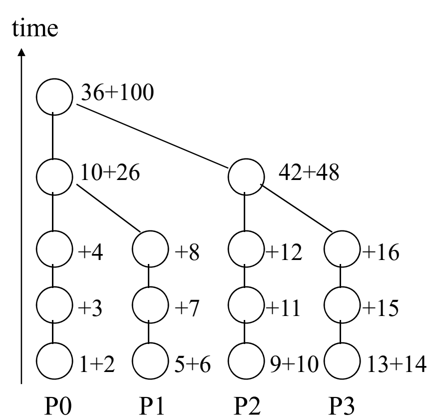


Now, we want to sum *n* elements on *p* processors, *n* >> *p*

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:*

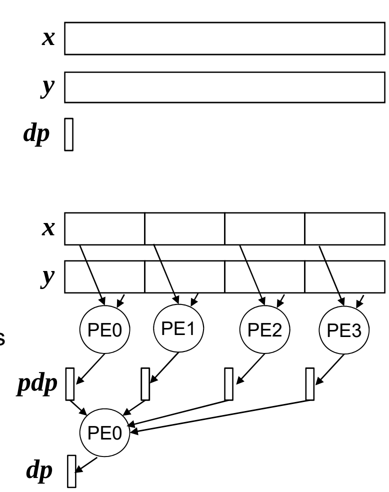
```
sum[Pid] = 0;
for ( i = 625 * Pid  ; i < 625  * (Pid +1)  ;  i++)
    sum[Pid]  = sum[Pid] + x[i];
half= 8 ;   /* P = 16 */
for (i=0 ; i < 4 ;  i++)
   { synchronize ;      /* a barrier */
     if(Pid < half ) sum[Pid] = sum[Pid] + sum[Pid + half ] ;
     half = half / 2; }
```

- *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]
```

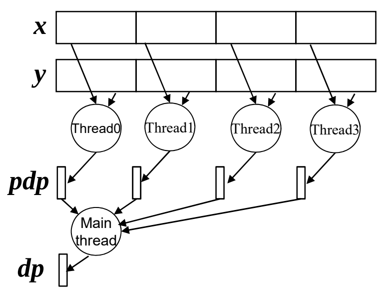


Using 4 processors:
- Partition the arrays into 4 parts
- Each processor computes a partial sum
- One processor sums up the partial sums
  (could use binary tree reduction)

Multi-thread version of the dot product example

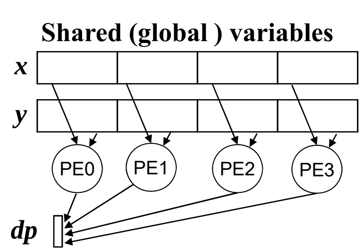
- Multi-threading was originally designed for Hiding Memory Latency
- With multicores, multiple threads will execute on multiple cores

```
// 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 ;
```

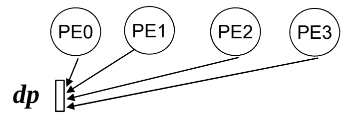



```
void partial_product (k, n);
{ int i, pdp = 0 ;  /* pdp is private -- each thread has its own copy */
  for (i = k*n/4;  i <  (k+1) * n/4 ;  i++)
       pdp += x[i] * y[i] ;
  pd += pdp ;
  return ;
}
```

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 *1* to *dp*
store *dp* to memory

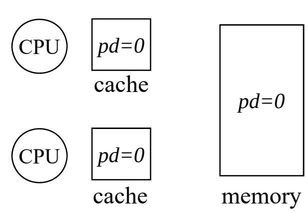
- A critical section is a section of code that can be executed by one processor at a time (to guarantee mutual exclusion)
- *locks* can be used to enforce mutual exclusion

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:**

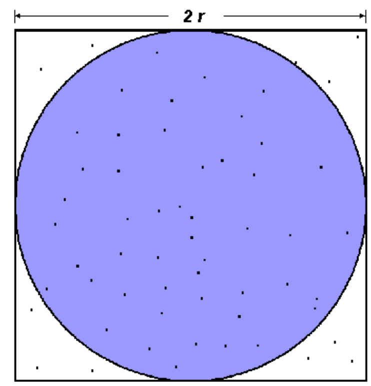
```
int pthread_mutex_lock ( pthread_mutex_t *mutex_lock);

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:

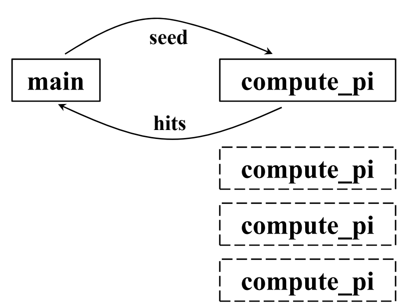
- Inscribe a circle in a square
- Randomly generate points in the square
- Determine the number of points in the square that are also in the circle
- Let A_c/A_s be the number of points in the circle divided by the number of points in the square
- PI $\approx 4 * (A_c/A_s)$
- Note that the more points 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] ;
```

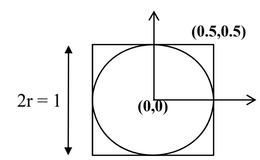

An Example (compute π)

```
    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()