Thinking parallel

- The following computes the sum of \( x[0]+...+x[15] \) serially:

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

- 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 \( n/2 \) processor

\[
\]
Example code on SMP

```c
half = 4;
repeat
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
    half = half/2;
until (half == 1);
```

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

```c
half = 5;
repeat
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
    barrier synch();
/* when half is odd; P0 gets the last element */
    if (half % 2 != 0 && Pn == 0)
        sum[0] = sum[0] + sum[4];
    half = half/2;
until (half == 1);
```

Now, we want to sum \( n \) elements on \( p \) processors, \( n \gg p \)
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
- 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?
- What is the speedup?

Programming a shared address space machine

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

```c
Sum[0] = 0;
for (i = 625 * Pn ; i < 625 * (Pn +1) ; i++)
    Sum[0] = Sum[0] + A[i];
Half = 8 ; /* P = 16 */
for (i=0 ; i < 4 ; i++)
    { synchronize ; /* a barrier */
    if(Pn < Half ) Sum[0] = Sum[0] + Sum[Half ];
    Half = Half / 2; }
```

- \( Sum[ ] \) and \( A[ ] \) are shared arrays,
- \( Half, Pn \) and \( i \) are private variables (each processor has its own copy).
- Where will the global sum end up being?
- 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 \( A \)?
Multithread programming for shared memory

Originally designed for Hiding Memory Latency

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

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

Multi-thread version of the dot product example

Instead of calling the function `partial_product()`, create a new thread that will execute that function.

```c
dp = 0;
for (k = 0; k < 4; k++)
    /* fork 4 threads */
    create_thread(partial_product, k*n/4, n/4);
Wait until all threads return;  /* join threads */
for (k = 0; k < 4; k++)
    dp += pdp[k];

void partial_product (a , b);
{
pdp[k] = 0;
for (i = a; i < a+b; i++)
    pdp[k] += x[i] * y[i] ;
return ;
}
```

Shared (global) variables

![Diagram of vectors and dot product](image)
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.

The Pthread API

(see [https://computing.llnl.gov/tutorials/pthreads/](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:

```c
#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);
```
Another version of the dot product example

dp = 0 ;
for (k = 0; k < 4; k++)
    create_thread (partial_product, k*n/4, n/4);
Wait until all threads return ;

void partial_product (a , b);
{ int pdp = 0 ;  /* each thread has its own copy of pdp */
    for (i = a ;  i < a+b ;  i++)
        pdp += x[i] * y[i] ;
    pd += pdp ;
    return ;
}

Synchronization (race conditions)

What is the output of the following program??

dp = 0 ;
for (id = 0; id < 4; id++)
    create_thread (..., sum_computed_values, ...);

void sum_computed_values ();
{ pdp = compute the value () ;
    dp += pdp ;
}

➢ 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
Multithreading: shared variables

- A variable declared within the thread function is private to that thread (cannot be accessed by other threads)
- A variable declared outside the thread function is shared and is accessible by all threads
- The race condition explained in slide 36 results if two threads simultaneously update a shared variable.
- Moreover, if threads execute on different CPUs, then a shared variable can be cached by multiple CPUs. This creates the cache coherence problems talked about in Section 5.

Example:
Initially pd = 0
Thread 1 executes pd=pd+1
Thread 2 executes pd=pd+1

This is why we need to keep the caches coherent

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);
```
Mutual Exclusion

- We need mutual exclusion in both parallel and serial programs (why?)
- Hardware support is needed to implement locks
- The load/store operation should be indivisible if we want to implement locks
- May use atomic swap operations (see next slide).
- Locks can be used to allow exclusive access to shared data.
- In a bus-connected system, atomic operations can be implemented by not releasing the bus before finishing the operation.
- A processor that wants the lock spins waiting for the bus release.

Mutual exclusion

- Basic Facility Required
  - Atomically retrieve and change a value (Read and Modify Instructions)
  - Atomic Reads (Load) and Atomic Writes (Stores) are not sufficient
- Example: Atomic Exchange
  - Interchanges a value in register for a value in memory
    - loads the value in memory into the register
    - stores the value in register into the memory
  - Use atomic exchange to acquire and release a lock in memory (Lock)

```
// Assume (Lock = 0) means Lock is free
Lock:
    Put 1 in Register, Reg
    Atomic Swap (Reg,Lock)
    if (Reg = 1) then try again
Unlock:
    Lock = 0
```
Barrier synchronization

- A barrier synchronization can be implemented using a shared variable initialized to N, the number of processors.
- When a processor reaches the barrier, it decrements the shared variable by 1 and waits 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?

An Example (compute $\pi$)

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 $R$ be the number of points in the circle divided by the number of points in the square
- $\pi \approx 4 R$
- Note that the more points generated, the better the approximation
#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) {
  ...
}

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*/

  int i ;
  pthread_t p_threads[MAX_THREADS];
  pthread_attr_t attr;
  double computed_pi;
  struct arg_to_thread my_arg[MAX_THREADS] ;

  pthread_attr_init (&attr);
An Example (compute $\pi$)

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

for (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 $\pi$)

```c
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”
An Example (compute $\pi$)

```c
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
```