

# Lecture 3

Programming with threads

## Announcements

- Office hours:  
Mondays 3p to 4p, Thursdays 4p to 5p
- Friday's Section:  
Moved to 2:00 to 2:50, CSB 004
- Special session for those who cannot make section  
time: 3:00 to 3:20, CSB 004
- You must receive authorization from Professor  
Baden to take the quiz at the alterative time

## Assignment #0

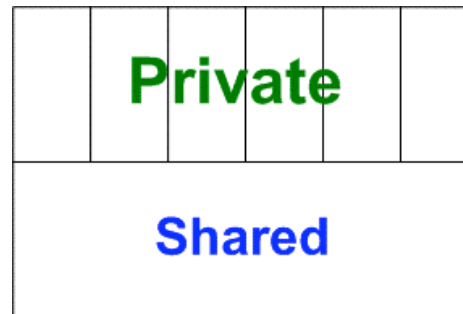
- Complete this by Friday to establish a stable platform for running threaded code
- A1 will be posted later on Friday

# SPMD execution model

- Most parallel programming is implemented under the Same Program Multiple Data programming model = SPMD
- Other names for this model are “loosely synchronous” or “bulk synchronous”
- Programs execute as a set of  $P$  processes or threads
  - We specify  $P$  when we run the program
  - Each process/thread is usually assigned to a different physical processor
- Each process or thread
  - is initialized with the same code
  - has an associated *rank*, a unique integer in the range  $0:P-1$
  - executes instructions at its own rate
- Processes communicate via messages or shared memory, threads through normally through shared memory

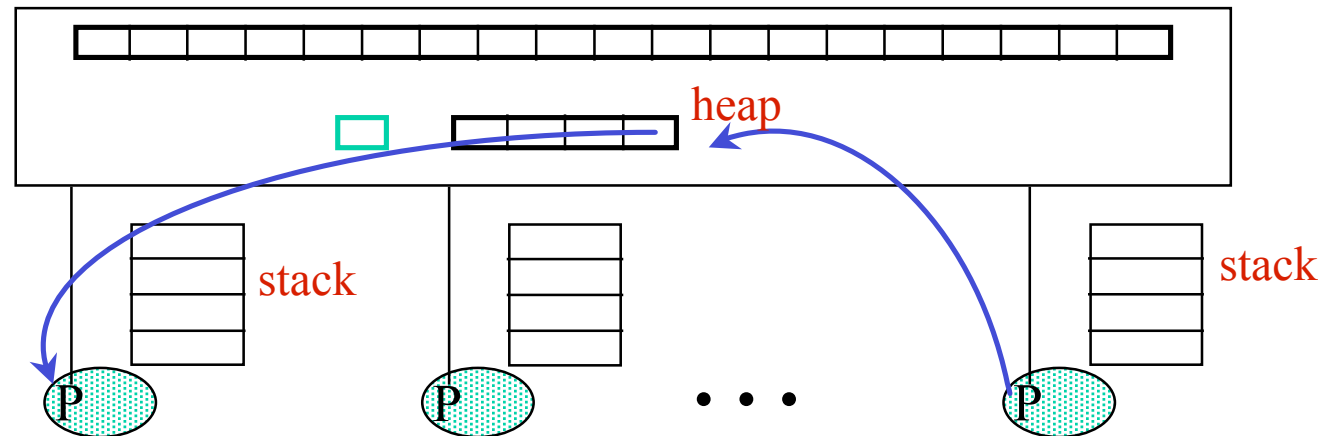
# Shared memory programming with threads

- A collection of concurrent instruction streams, called *threads*
- Each thread has a unique thread ID
- A new storage class: shared data
- A thread is similar to a procedure call with notable differences
  - A procedure call is “synchronous:” a return indicates completion
  - A spawned thread executes asynchronously until it completes
  - Both share global storage with caller
  - Synchronization is needed when updating shared state



# Why threads?

- Processes are “heavy weight” objects scheduled by the OS
  - Protected address space, open files, and other state
- A thread, AKA a lightweight process (LWP) is sometimes more appropriate
  - Threads share the address space and open files of the parent, but have their own stack
  - Reduced management overheads
  - Kernel scheduler multiplexes threads



# Practical issues

- Thread creation is faster than process creation (real time)
- Moving data in shared memory is cheaper than passing a message through shared memory

<https://computing.llnl.gov/tutorials/threads>

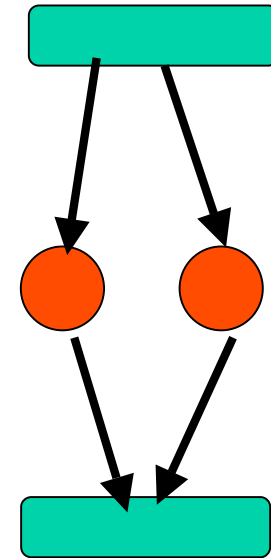
|                           | CPU/<br>node | Fork<br>( $\mu$ s) | Create<br>( $\mu$ s) | MPI Shared<br>Mem<br>(GB/sec) | Mem -<br>CPU<br>(GB/sec) |
|---------------------------|--------------|--------------------|----------------------|-------------------------------|--------------------------|
| AMD 2.4 GHz Opteron       | 8            | 41.1               | 0.66                 | 1.2                           | 5.3                      |
| IBM 1.9 GHz POWER5 p5-575 | 8            | 64.2               | 1.75                 | 4.1                           | 16                       |
| INTEL 2.4 GHz Xeon        | 2            | 55.0               | 1.64                 | 0.3                           | 4.3                      |
| Intel 1.4 GHz Itanium 2   | 4            | 54.5               | 2.03                 | 1.8                           | 6.4                      |

# Threads in practice

- A common interface is the POSIX Threads “standard” (pthreads): IEEE POSIX 1003.1c-1995
  - Beware of non-standard features
- Another approach is to use program annotations via openMP

# Programming model

- Start with a single root thread
- Fork-join parallelism to create concurrently executing threads
- Threads may or may not execute on different processors, and might be interleaved
- Scheduling behavior specified separately



# OpenMP programming

- Simpler interface than explicit threads
- Parallelization handled via annotations
- See <http://www.openmp.org>
- Parallel loop:

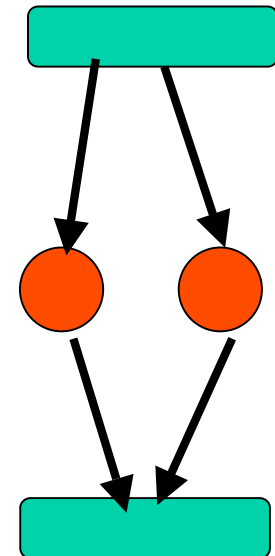
```
#pragma omp parallel private(i) shared(n)
{
    #pragma omp for
    for(i=0; i < n; i++)
        work(i);
}
```

# Parallel Sections

```
#pragma omp parallel // Begin a parallel construct
{ // form a team
  // Each team member executes the same code
  #pragma omp sections // Begin work sharing
  {
    #pragma omp section // A unit of work
    {x = x + 1;}

    #pragma omp section // Another unit
    {x = x + 1;}

  } // Wait until both units complete
} // End of Parallel Construct; disband team
// continue serial execution
```



# Race conditions

- Consider the statement, assuming  $x == 0$   
 $x = x + 1;$

- Generated code
  - $r1 \leftarrow (x)$
  - $r1 \leftarrow r1 + \#1$
  - $r1 \rightarrow (x)$

- Possible interleaving with two threads

P1  
 $r1 \leftarrow x$

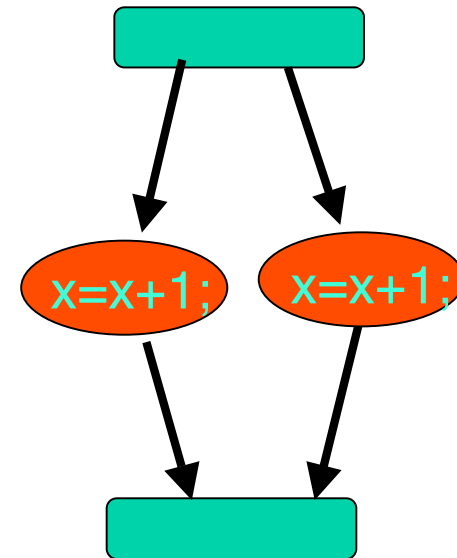
$r1 \leftarrow r1 + \#1$

$x \leftarrow r1$

P2  
 $r1 \leftarrow x$

$r1 \leftarrow r1 + \#1$

$x \leftarrow r1$



*$r1(P1)$  gets 0*  
 *$r2(P2)$  also gets 0*  
 *$r1(P1)$  set to 1*  
 *$r1(P1)$  set to 1*  
*P1 writes its R1*  
*P2 writes its R1*

# Race conditions

- A *Race* condition arises when the timing of accesses to shared memory can affect the outcome
- We say we have a *non-deterministic* computation
- Sometimes we can use non-determinism to advantage, but we avoid it usually
- For the same input we want to obtain the same results from operations that do not have side effects  
(like I/O and random number generators)
- Memory consistency and cache coherence are necessary but not sufficient conditions for ensuring program correctness
- We need to take steps to avoid race conditions through appropriate program synchronization

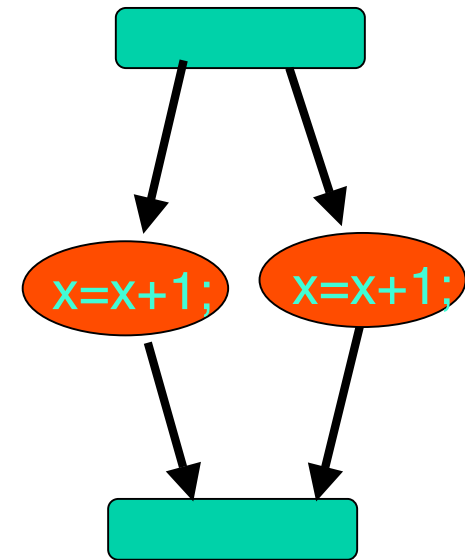
## Mutual exclusion

- Each process samples and increments the shared variable  $x$
- The code performing the operation is a *critical section*
- Only one thread at a time may access this code
- We use *mutual exclusion* to implement the critical section
- A critical section is non-parallelizing computation.. sensible guidelines?

# Critical Sections

- Only one thread at a time may run the code in a critical section
- *Mutual exclusion* to implement critical sections

```
#pragma omp parallel // Begin a parallel construct
{
  #pragma omp sections // Begin worksharing
  { //
    #pragma omp critical // Critical section
    {x = x + 1}
    #pragma omp critical // Another critical section
    {x = x + 1}
    ... // More Replicated Code
  }
  #pragma omp barrier // Wait for all members to arrive
  // Wait until both units of work complete
}
```



## How does mutual exclusion work?

- A simple solution is to use a mutex variable
- E.g. provided by pthreads
- Locks may be CLEAR or SET
- Lock() waits if the lock is set, else sets the lock
- Unlock clears the lock if set

```
Mutex mtx;  
mtx.lock();  
    CRITICAL SECTION  
mtx.unlock();
```

# Coding with pthreads

```
#include <pthread.h>
#include <iostream>
using namespace std;
void *Hello(void *tid) {
    cout << "Hello from " << (int) tid << endl;
    pthread_exit(NULL);
}
int main(int argc, char *argv[ ]){
    int NT = 3, status;
    pthread_t th[NT];
    for(int t=0;t<NT;t++)
        assert(!pthread_create(&th[t],NULL, Hello, (void *)t));
    for(int t=0;t<NT;t++)
        assert(!pthread_join(th[t], (void **) &status));
    pthread_exit(NULL);
}
```

```
% g++ t.C -lpthread
% a.out
Hello from thread 0
Hello from thread 1
Hello from thread 2
```

# Computing a sum in parallel

- Also see: dotprod\_mutex.c in the LLNL tutorial

Globals:

```
pthread_mutex_t mutex_sum;  
int *x, global_sum, N, NT;
```

Main:

```
for (int i=0; i < N; i++) x[i] = i;  
global_sum = 0;  
assert(!pthread_mutex_init(&mutex_sum, NULL));  
pthread_t thrd[NT];  
for(int t=0;t<NT;t++)  
    pthread_create(&thrd[t], NULL, summ, (void *)t);  
    //Join threads...  
cout << "The sum of 0 to " << N-1 << " is: " << sum << endl;
```

# The computation

```
void *summ(void *arg){
    int TID = (int)arg;
    int i0 = TID*(N/NT), i1 = i0 + (N/NT);
    double mysum = 0;
    for ( i=i0; i<i1; i++)
        mysum += x[i] ;
    pthread_mutex_lock (&mutex_sum);
    global_sum += mysum;
    pthread_mutex_unlock (&mutex_sum);
    pthread_exit((void*) 0);
}
```

```
pthread_create(&thrd[t], NULL,
               summ, (void *)t));
```

```
g++ sum.C -lpthread
% a.out
%The sum of 0 to 2047 is: 2096128
```

# Implementation issues

- Hardware support
  - Test and set: atomically test a memory location and then set it
  - Cache coherence protocol provides synchronization
- Scheduling issues
  - Busy waiting or spinning
  - Yield process
  - Pre-emption by scheduler

# Correctness and synchronization

```
% a.out 5  
# threads: 5  
Total sum is 2  
The sum of 0 to 4 is: 30
```

```
int sum = 0;    // Global  
void *sumIt(void *arg){ // Thread  
    int TID = (int)arg;  
    pthread_mutex_lock (&mutex_sum);  
    sum += 2*(TID+1);  
    pthread_mutex_unlock (&mutex_sum);  
    if (TID == 0)  
        cout << "Total sum is " << sum << endl;  
    pthread_exit((void*) 0); }
```

# Barrier synchronization

- Why was the sum incorrectly reported?
- We read a location updated by other threads that had not had the chance to produce their contribution (true dependence)
- Don't overwrite the values used by other processes in the current iteration until they have been consumed (anti-dependence)
- A barrier can be built with locks

# Building a linear time barrier with locks

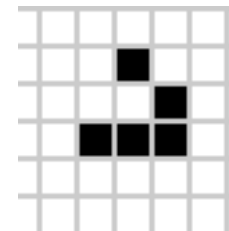
```
Mutex arrival=UNLOCKED, departure=LOCKED;  
int count=0;
```

```
void Barrier( )  
    arrival.lock( );           // atomically count the  
    count++;                  // waiting threads  
    if (count < n$proc) arrival.unlock( );  
    else departure.unlock( );   // last processor  
                                // enables all to go  
  
    departure.lock( );  
    count--;                   // atomically decrement  
    if (count > 0) departure.unlock( );  
    else arrival.unlock( );     // last processor resets state
```

# A First application

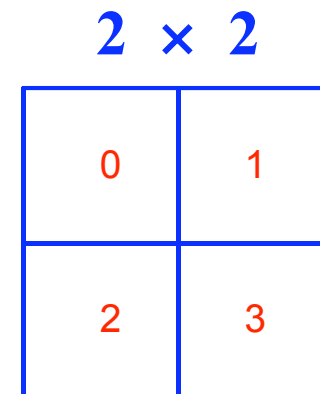
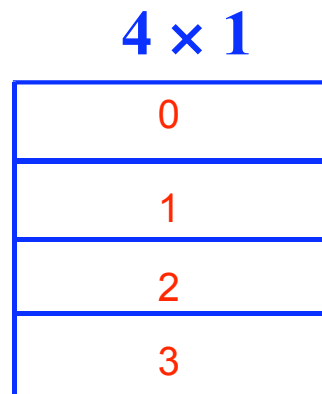
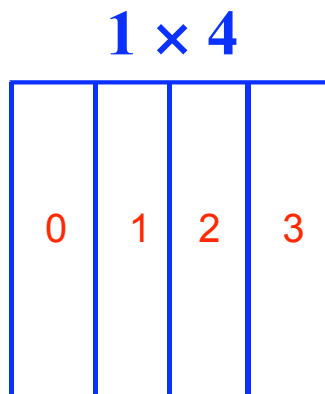
- Playing Conway's Game of Life  
[http://en.wikipedia.org/wiki/Conway's\\_Game\\_of\\_Life](http://en.wikipedia.org/wiki/Conway's_Game_of_Life)

```
for t in 0:T-1
  forall (i,j) in 0:N-1 × 0:N-1
    nNeigh = number of neighbors of Worldt[i,j]
    if World[i,j] AND (nNeigh == 2 or nNeigh ==3) then
      WorldNext[i,j] = LIVE
    else
      WorldNext[i,j] = (nNeigh == 2)
    end if
  end forall
end for
World = WorldNext;
```



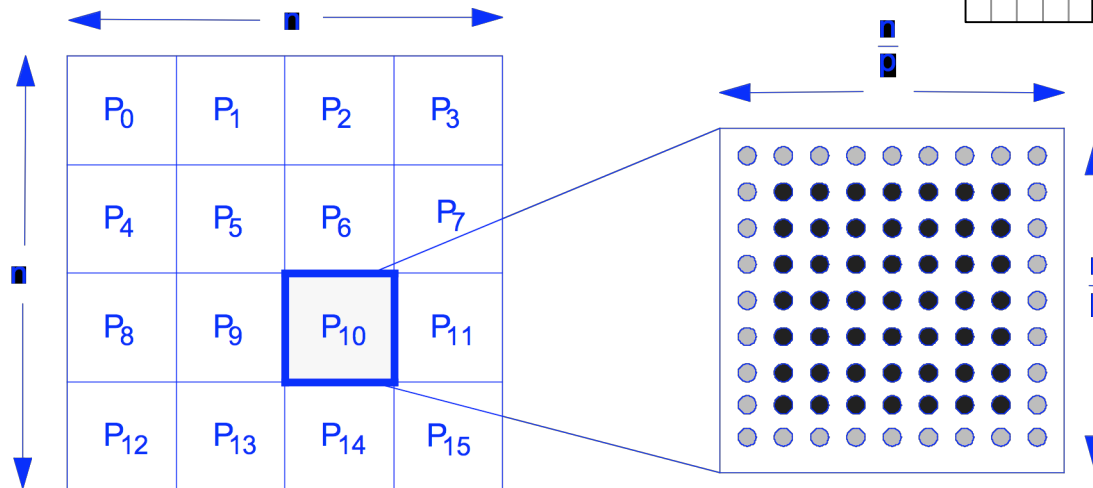
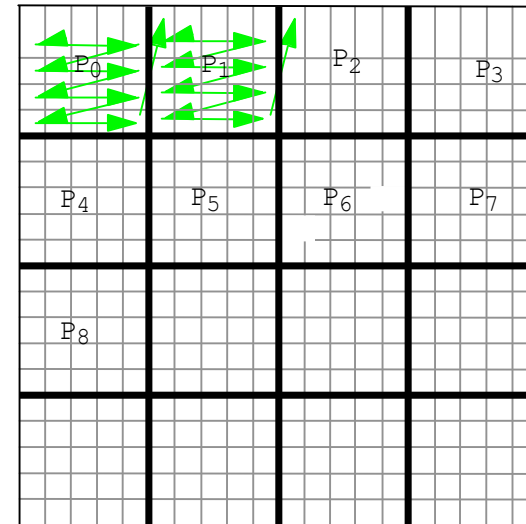
# Partitioning in multiple dimensions

- Splits up the data over processors
- Different partitioning according to the *processor geometry*
- For  $P$  processors geometries are of the form  $p_0 \times p_1$ , where  $P = p_0 p_1$
- For  $P=4$ , 3 possible geometries



# Data access

- Off processor values surround each local subproblem
- Non-contiguous data
- Inefficient to access values on certain faces/edges



# Threaded implementation

mymin = 1 + (\$TID + n/NT),    mymax = mymin + NT -1;

**while** (population) **do**

    myPop = population = 0;

**\$omp barrier**

**for** i = mymin **to** mymax **do**

**for** j = 1 **to** n **do**

            Wnew[i,j] = ...

**if** Wnew[i,j] myPop++;

**end for**

**end for**

**\$omp critical**

        { population += myPop }

**\$omp barrier**

    W[:,:] = WNew[:,:]

**\$omp barrier**

**end while**

- Pseudo code uses OpenMP notation
- Code handles loop decomposition
- Don't read locations updated by other processes in the previous iteration until they have been produced (true dependence)
- Don't overwrite values used by other processes in the current iteration until they have been consumed (anti-dependence)

# Computing the number of inhabitants

- What is the running time?
- Can we do better?

```
$omp critical  
{ population += myPop }
```